Lecture Notes in Artificial Intelligence 5663

Edited by R. Goebel, J. Siekmann, and W. Wahlster

Subseries of Lecture Notes in Computer Science

Renate A. Schmidt (Ed.)

Automated Deduction – CADE-22

22nd International Conference on Automated Deduction
Montreal, Canada, August 2-7, 2009
Proceedings

Volume Editor

Renate A. Schmidt
School of Computer Science
The University of Manchester
Manchester, UK
E-mail: schmidt@cs.man.ac.uk

Library of Congress Control Number: Applied for

CR Subject Classification (1998): I.2.3, I.2, F.4.1, F.3, F.4, D.2.4

LNCS Sublibrary: SL 7 – Artificial Intelligence

ISSN 0302-9743
ISBN-10 3-642-02958-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-02958-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12717996 06/3180 5 4 3 2 1 0

Preface

This volume contains the proceedings of the 22nd International Conference on Automated Deduction (CADE-22). The conference was hosted by the School of Computer Science at McGill University, Montreal, Canada, during August 2–7, 2009. CADE is the major forum for the presentation of research in all aspects of automated deduction. Within this general topic the conference is devoted to foundations, applications, implementations and practical experiences. CADE was founded in 1974 when it was held in Argonne, USA. Since then CADE has been organized first on a bi-annual basis mostly and since 1996 on an annual basis, in 2001, 2004, 2004, 2006 and 2008 as a constituent of IJCAR.

This year the Program Committee selected 32 technical contributions out of 77 initial submissions. Of the selected papers 27 were regular papers and 5 were system papers. Each paper was refereed by at least three reviewers on its significance, technical quality, originality, quality of presentation and relevance to the conference. The refereeing process and the Program Committee meeting were conducted electronically via the Internet using the EasyChair conference management system. The program included three invited lectures by distinguished experts in the area: *Instantiation-Based Automated Reasoning: From Theory to Practice* by Konstantin Korovin (The University of Manchester, UK), *Integrated Reasoning and Proof Choice Point Selection in the Jahob System: Mechanisms for Program Survival* by Martin Rinard (Massachusetts Institute of Technology, USA), and *Building Theorem Provers* by Mark Stickel (SRI International, USA). In addition, the conference included a two-day program of a diverse range of workshops and tutorials. Two system competitions were held during the conference: *The CADE ATP System Competition (CASC)* organized by Geoff Sutcliffe, and *The Satisfiability Modulo Theories Competition (SMT-COMP)* organized by Clark Barrett, Morgan Deters, Albert Oliveras and Aaron Stump.

The papers in these proceedings cover a diversity of logics, extending from classical propositional logic, first-order logic and higher-order logic, to non-classical logics including intuitionistic logic, modal logic, temporal logic and dynamic logic. Also covered are theories, extending from various theories of arithmetic to equational theories and algebra. Many of the papers are on methods using superposition, resolution, SAT, SMT, instance-based approaches, tableaux and term rewriting but also hierarchical reasoning and the inverse method, or combinations of some of these. The most salient issues include, for example, termination and decidability, completeness, combinations, interpolant computation, model building, practical aspects and implementations of fully automated theorem provers. Considerable impetus comes from applications, most notably analysis and verification of programs and security protocols, and the provision and support of various automated reasoning tasks.

The CADE-22 Program Committee was part of the Herbrand Award Committee, which additionally consisted of the previous award winners of the last ten years and the Trustees of CADE Inc. The committee has decided to present the Herbrand Award for Distinguished Contributions to Automated Reasoning to Deepak Kapur in recognition of his seminal contributions to several areas of automated deduction including inductive theorem proving, term rewriting, unification theory, integration and combination of decision procedures, lemma and loop invariant generation, as well as his work in computer algebra, which helped to bridge the gap between the two areas.

I would like to thank the many people without whom the conference would not have been possible. First, I would like to thank all authors who submitted papers, all participants of the conference as well as the invited keynote speakers, the tutorial speakers, the workshop organizers and the system competition organizers for their contributions. I am very grateful to the members of the Program Committee and the external reviewers for carefully reviewing and selecting the papers. We are all indebted to Andrei Voronkov for providing EasyChair and his support during the discussion phase of the submissions. I also thank the Trustees of CADE Inc. for their advice and support.

Special thanks go to the members of the local organization team in the School of Computer Science at McGill University for their tremendous amount of effort, especially Maja Frydrychowicz, who did outstanding work. Moreover, I am extremely grateful to Aaron Stump, the Workshop Chair, Carsten Schürmann, the Publicity Chair, and of course Brigitte Pientka, who as Conference Chair was involved in almost every aspect of the organization of the conference.

Finally, it is my pleasure to acknowledge the generous support by the School of Computer Science and the Faculty of Science at McGill University, and Microsoft Research.

May 2009 Renate Schmidt

Conference Organization

Program Chair

Renate Schmidt	The University of Manchester

Program Committee

Alessandro Armando	Università di Genova
Franz Baader	Technische Universität Dresden
Peter Baumgartner	NICTA, Canberra
Bernhard Beckert	Universität Koblenz-Landau
Nikolaj Bjørner	Microsoft Research
Maria Paola Bonacina	Università degli Studi di Verona
Alessandro Cimatti	Istituto per la Ricerca Scientifica e Tecnologica, Trento
Silvio Ghilardi	Università degli Studi di Milano
Jürgen Giesl	RWTH Aachen
Rajeev Goré	The Australian National University
Reiner Hähnle	Chalmers University of Technology
John Harrison	Intel Corporation
Miki Hermann	École Polytechnique
Ullrich Hustadt	University of Liverpool
Katsumi Inoue	National Institute of Informatics, Japan
Tommi Junttila	Helsinki University of Technology
Deepak Kapur	University of New Mexico
Alexander Leitsch	Technische Universität Wien
Christopher Lynch	Clarkson University
Claude Marché	INRIA Saclay, Parc Orsay Université
William McCune	University of New Mexico
Aart Middeldorp	Universität Innsbruck
Hans de Nivelle	Uniwersytet Wrocławski
Albert Oliveras	Universitat Politècnica de Catalunya
Lawrence Paulson	University of Cambridge
Brigitte Pientka	McGill University
David Plaisted	University of North Carolina at Chapel Hill
Michaël Rusinowitch	LORIA and INRIA, Lorraine
Renate Schmidt	The University of Manchester
Carsten Schürmann	IT-Universitetet i København
Aaron Stump	The University of Iowa
Geoff Sutcliffe	University of Miami
Cesare Tinelli	The University of Iowa

Andrei Voronkov	The University of Manchester
Christoph Weidenbach	Max-Planck-Institut für Informatik

Conference Chair

Brigitte Pientka	McGill University

Workshop and Tutorial Chair

Aaron Stump	The University of Iowa

Publicity Chair

Carsten Schürmann	IT-Universitetet i København

Local Organization

Maja Frydrychowicz	McGill University
Brigitte Pientka	McGill University

External Reviewers

Stefan Andrei
Roger Antonsen
Carlos Areces
Gilles Barthe
Armin Biere
Thorsten Bormer
Ana Bove
Marco Bozzano
Kai Brünnler
Richard Bubel
John Burgess
Roberto Carbone
Franck Cassez
Thomas Chatain
Jacek Chrząszcz
Frank Ciesinski
Aaron Coble
Sylvain Conchon
Leonardo de Moura
Stéphane Demri
Derek Dreyer
Jori Dubrovin
Stephan Falke
William Farmer
Germain Faure
Arnaud Fietzke
Jean-Christophe Filliâtre
Anders Franzén
Achille Frigeri
Alexander Fuchs
Carsten Fuhs
Alfons Geser
Laura Giordano
Amit Goel
Alberto Griggio
Joe Hendrix
Anders Starcke Henriksen
Thomas Hillenbrand
Timothy Hinrichs
Matthias Horbach
Joe Hurd
Koji Iwanuma
Manfred Jaeger
Kevin Jones
Vladimir Klebanov
Konstantin Korovin

Miyuki Koshimura
Alexander Krauss
Sava Krstić
Evgeny Kruglov
Martin Lange
Giacomo Lenzi
Stéphane Lescuyer
Tomer Libal
Pablo López
Salvador Lucas
Thomas Lukasiewicz
Filip Marić
Sebastian Mödersheim
Georg Moser
Hidetomo Nabeshima
Enrica Nicolini
Thomas Noll
Claudia Obermaier
Duckki Oe
Greg O'Keefe
Jens Otten
Andrei Paskevich
Rafael Peñaloza
Lorenzo Platania
Marc Pouzet
Silvio Ranise
Sandip Ray
Andrew Reynolds
Alexandre Riazanov
Christophe Ringeissen
Enric Rodríguez-Carbonell
Albert Rubio
Philipp Rümmer
Andrey Rybalchenko
Gernot Salzer
Viktor Schuppan
Roberto Sebastiani
Luciano Serafini
Bariş Sertkaya
Jakob Grue Simonsen
Michael Stevens
Umberto Straccia
Lutz Straßburger
Boontawee Suntisrivaraporn
Naoyuki Tamura
René Thiemann
Dmitry Tishkovsky
Stefano Tonetta
Dmitry Tsarkov
Tarmo Uustalu
Arild Waaler
Uwe Waldmann
Geoffrey Washburn
Daniel Weller
Patrick Wischnewski
Harald Zankl
Hans Zantema
Hantao Zhang
plus other anonymous reviewers

Sponsoring Institutions

School of Computer Science, McGill University
Faculty of Science, McGill University
Microsoft Research

Table of Contents

Session 1. Invited Talk

Session 2. Combinations and Extensions

Session 3. Minimal Unsatisfiability and Automated Reasoning Support

Session 4. System Descriptions

Session 5. Invited Talk

Session 6. Interpolation and Predicate Abstraction

Session 7. Resolution-Based Systems for Non-classical Logics

Session 8. Termination Analysis and Constraint Solving

Session 9. Invited Talk

Session 10. Rewriting, Termination and Productivity

Session 11. Models

Session 12. Modal Tableaux with Global Caching

Session 13. Arithmetic

Integrated Reasoning and Proof Choice Point Selection in the Jahob System – Mechanisms for Program Survival

Martin Rinard

Department of Electrical Engineering and Computer Science
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
32 Vassar Street, 32-G744
Cambridge, Massachusetts 02139
rinard@mit.edu

Abstract. In recent years researchers have developed a wide range of powerful automated reasoning systems. We have leveraged these systems to build Jahob, a program specification, analysis, and verification system. In contrast to many such systems, which use a monolithic reasoning approach, Jahob provides a general *integrated reasoning* framework, which enables multiple automated reasoning systems to work together to prove the desired program correctness properties.

We have used Jahob to prove the full functional correctness of a collection of linked data structure implementations. The automated reasoning systems are able to automatically perform the vast majority of the reasoning steps required for this verification. But there are some complex verification conditions that they fail to prove. We have therefore developed a proof language, integrated into the underlying imperative Java programming language, that developers can use to control key choice points in the proof search space. Once the developer has resolved these choice points, the automated reasoning systems are able to complete the verification. This approach appropriately leverages both the developer's insight into the high-level structure of the proof and the ability of the automated reasoning systems to perform the mechanical steps required to prove the verification conditions.

Building on Jahob's success with this challenging program verification problem, we contemplate the possibility of verifying the complete absence of fatal errors in large software systems. We envision combining simple techniques that analyze the vast majority of the program with heavyweight techniques that analyze those more sophisticated parts of the program that may require arbitrarily sophisticated reasoning. Modularity mechanisms such as abstract data types enable the sound division of the program for this purpose. The goal is not a completely correct program, but a program that can survive any remaining errors to continue to provide acceptable service.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 1–16, 2009.

1 Introduction

Data structure consistency is a critical program correctness property. Indeed, it is often directly related to the meaningful survival of the program. As long as a program preserves the integrity of its core data structures, it is usually able to execute through errors to continue to provide acceptable (although not necessarily perfect) service to its users [1,2,3,4,5,6,7].

We have developed a general program specification and verification system, Jahob, and used Jahob to verify, for the first time, the full functional correctness of a collection of linked data structure implementations [8,9,10]. This verification constitutes a key step towards the goal of statically ensuring data structure consistency — part of the verification process is ensuring that individual data structure implementations preserve invariants that capture key internal data structure consistency constraints.

1.1 Integrated Reasoning

To verify the full functional correctness of linked data structure implementations, Jahob must work with sophisticated properties such as transitive closure, lambda abstraction, quantified invariants, and numerical relationships involving the sizes of various data structure components. The diversity and complexity of the resulting verification conditions makes the use of a single monolithic reasoning system counterproductive. Instead, Jahob uses *integrated reasoning* — it implements a general framework that enables arbitrary reasoning systems to interoperate to prove the complex verification conditions that arise in this context [8,10]. The success of integrated reasoning depends on two techniques:

- **Splitting:** Jahob splits verification conditions into equivalent conjunctions of subformulas and processes each subformula independently. Splitting enables Jahob to take a single formula that requires many kinds of reasoning, divide the formula up into subformulas (each of which requires only a single kind of reasoning), then apply multiple different reasoning systems as appropriate to solve the subformulas. The result is that Jahob is able to leverage the combined power of an arbitrary number of reasoning systems to solve complex formulas.
- **Formula Approximation:** In general, each reasoning system will have its own restrictions on the set of formulas that it will accept as input. Several formula approximation techniques make it possible to successfully deploy a diverse set of reasoning systems together within a single unified reasoning framework. These approximation techniques accept higher-order logic formulas and create equivalent or semantically stronger formulas accepted by specialized decision procedures, provers, and other automated reasoning systems.

  Our approximation techniques rewrite equalities over complex types such as functions, apply beta reduction, and express set operations using first-order quantification. They also soundly approximate constructs not directly

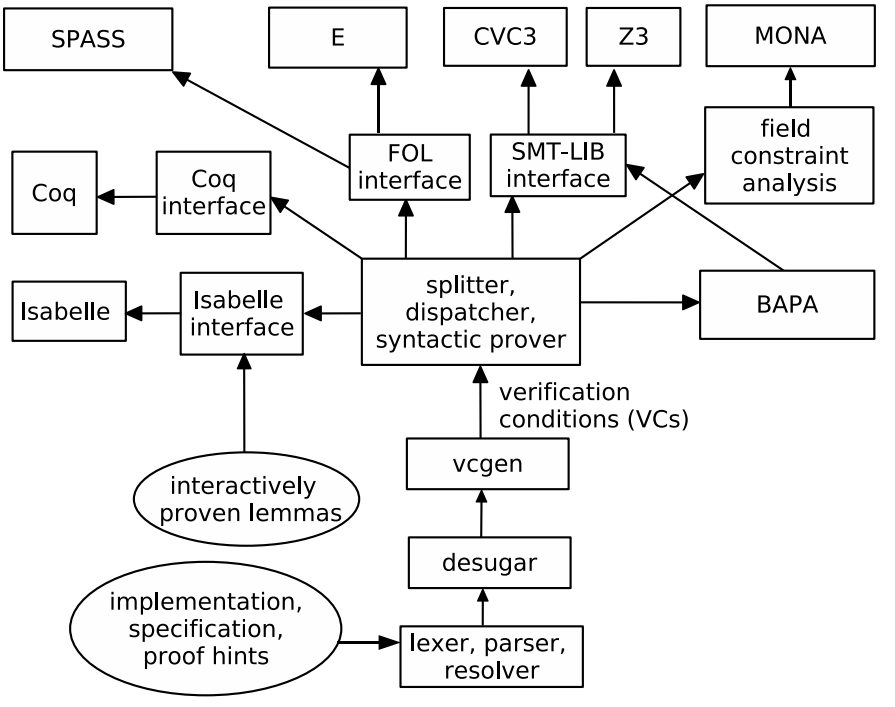


Fig. 1. Integrated Reasoning in the Jahob System

supported by a given specialized reasoning system, typically by replacing problematic constructs with logically stronger and simpler approximations.

Together, these techniques make it possible to productively apply arbitrary collections of specialized reasoning systems to complex higher-order logic formulas. Jahob contains a simple syntactic prover, interfaces to first-order provers (SPASS [11] and E [12]), an interface to SMT provers (CVC3 [13,14] and Z3 [15,16]), an interface to MONA [17,18], an interface to the BAPA decision procedure [19,20], and interfaces to interactive theorem provers (Isabelle [21] and Coq [22]) (see Figure 1). The interactive theorem prover interfaces make it possible for Jahob to, when necessary, leverage human insight to prove arbitrarily complex formulas requiring arbitrarily sophisticated reasoning.

The reason integrated reasoning is so appropriate for our program verification tasks is the diversity of the generated verification conditions. The vast majority of the formulas in these verification conditions are well within the reach of even quite simple decision procedures. But there are always a few complex formulas that require sophisticated techniques. Integrated reasoning makes it possible for Jahob to apply multiple reasoning systems as appropriate: simple, fast solvers for simple formulas, and arbitrarily sophisticated (and therefore arbitrarily unscalable) solvers for complex formulas, and, if absolutely necessary, interactive theorem proving.

In recent years the automated reasoning community has developed a range of very powerful reasoning systems. We have found integrated reasoning to be an effective way to productively apply the combined power of these reasoning systems to verify complex and important program correctness properties. One particularly important aspect of integrated reasoning is its ability to leverage arbitrarily specialized reasoning systems that are designed to operate within arbitrarily narrow domains. The tight focus of these reasoning systems makes them irrelevant for the vast majority of the reasoning steps that Jahob must perform. But they are critical in enabling Jahob to deal effectively with the full range of properties that arise in the verification of sophisticated linked data structure implementations (and, we expect, will arise in other ambitious program verification efforts). Jahob's integrated reasoning technique makes it possible for these kinds of reasoning systems to make a productive contribution within a larger program analysis and verification system.

1.2 Proof Choice Point Selection

In our experience, automated reasoning systems encounter difficulties proving verification conditions when the corresponding proof search spaces have key choice points (such as quantifier instantiations, case splits, selection of induction hypotheses, and lemma decompositions) that the reasoning systems are unable to resolve in any relevant amount of time.

We have therefore developed a proof language that enables developers to resolve these key choice points [9]. Because this proof language is integrated into the underlying imperative programming language (Java), it enables developers to stay within a familiar conceptual framework as they work with choice points to shape the overall structure of the path through the proof search space. The automated reasoning systems in Jahob are then able to leverage this high-level guidance to perform all of the remaining proof steps. This approach appropriately leverages the complementary capabilities of the developers and the automated reasoning systems. Developers usually have the insight required to effectively shape the high-level structure of the correctness proof; automated reasoning systems excel at the detailed symbolic manipulation required to leverage the guidance to complete the proof.

Given the substantial reasoning capabilities of current reasoning systems and the rate at which these reasoning capabilities are improving, we expect technologies (such as the Jahob integrated proof language) that are designed to leverage high-level developer insight into the proof process to become increasingly viable. And because these techniques can significantly reduce the effort required to obtain a given result, we expect them to be increasingly deployed across a wide range of automated reasoning scenarios.

1.3 Program Survival

Data structure consistency provides, by itself, no guarantee that the program as a whole will execute correctly. Indeed, given the difficulty of obtaining full program correctness specifications (and, more fundamentally, the fact that no

one can state precisely what most large software systems should do), we expect full program verification to remain beyond reach for the foreseeable future.

But it is possible, however, to systematically identify all of the errors that could cause a program to fail outright. And because standard techniques such as testing can deliver programs that operate reliably on almost all inputs, the elimination of fatal errors is a key step towards obtaining software systems that can survive the inevitable remaining errors to continue to deliver acceptable service.

We have developed a set of (in most cases quite simple) dynamic techniques that, together, enable programs to survive virtually all errors [1,2,3,4,5,6,7]. The experimental results show that these techniques are surprisingly effective in enabling programs to survive otherwise fatal errors so that they can continue to provide effective service to users.

1.4 Static Verification of Survival Properties

In this paper we explore the possibility of using program analysis and verification technology to statically guarantee the absence of fatal errors in software systems. A recurring theme is that relatively simple and scalable techniques that can operate effectively across large parts of the program should be sufficient to verify the absence of fatal errors in almost all of the program.

But there will always be (for reasons of efficiency and desired functionality) complex parts of the program that are difficult if not impossible to analyze using these techniques. For these parts of the program we advocate the use of sophisticated heavyweight program analysis and verification techniques that potentially involve the developer, in some cases quite intimately, in the verification process. We see two keys to the success of this approach:

- **Modularity:** It is practical to apply heavyweight reasoning techniques only to a feasibly sized region of the program. Modularity mechanisms (such as abstract data types) that restrict the regions of the program to which these techniques must be applied are essential to the success of this approach.
- **Developer Interaction:** We have found both integrated reasoning and proof choice point selection to play critical roles in enabling successful data structure consistency proofs. Given the quite sophisticated properties that will inevitably arise, we believe that both of these techniques will also prove to be critical to the verification of other survival properties.

For any particular program correctness property, there is a temptation to develop specification languages and verification techniques that are tailored to that property. We have chosen, in contrast, to use a general purpose specification and verification approach. The disadvantage of this general approach is that for any specific property it can require more developer specification and verification effort than an approach tailored for that property. But the range of properties that our general approach can address justifies the engineering effort required to build a powerful and versatile reasoning system that the developer can use to specify and verify virtually any desired property. In particular, the system can usually smoothly extend to handle unforeseen extensions of the original properties.

Tailored approaches, in contrast, usually have difficulty supporting unforeseen extensions, in which case the developer is left with no good specification and verification alternative at all.

We focus on the static verification of several properties that are especially relevant to program survival: data structure consistency, the absence of memory errors such as null pointer dereferences and out of bounds accesses, the absence of memory leaks, and control flow properties such as the absence of infinite loops. In all of these cases, with the exception of memory leaks, we expect a dual analysis approach to be effective in making it possible to statically verify the desired property. In particular, we expect simple and scalable techniques to suffice for the vast majority of the program, with standard encapsulation mechanisms such as abstract data types providing the modularity properties required to apply heavyweight techniques successfully to those feasibly small regions of the program that require such techniques. While the current focus is on verifying complex properties in targeted parts of the program, in the longer run we expect the construction of effective scalable analyses for the simpler properties that must hold across large regions of the program to prove to be the more challenging problem.

2 Dynamic Techniques

Before considering static verification techniques for survival properties, we first review our experience with a collection of dynamic survival techniques. We obtained these techniques by analyzing the different ways that programs could fail, then developing a simple technique that enables the program to survive each class of failures.

In comparison with the static techniques considered in this paper, the great advantage of these dynamic techniques is their simplicity — instead of relying on potentially quite heavyweight program analysis and verification technology, the techniques simply change the semantics of the underlying model of computation to completely eliminate the targeted class of failures. The insight here is that standard models of computation are unforgiving and brittle — the unstated assumption is that because programs should be perfect, the developer should bear all of the responsibility for making the program execute successfully. Our new models of computation, in contrast, acknowledge that although developers make mistakes, these mistakes need not be fatal in the context of a more forgiving model of computation that works harder to make the program succeed. Our experimental results show that the resulting nonstandard models of computation can be quite effective in practice in enabling programs to execute effectively through situations that would cause them to fail with standard brittle models of computation.

2.1 Failure-Oblivious Computing

Programs that use failure-oblivious computing check each memory access (either statically or dynamically) for memory errors. On write memory errors (for example, an out of bounds write or write via a null pointer), the program simply

discards the write. On read memory errors, the program simply makes up a value to return as the result of the read [4]. In effect, this technique changes the semantics of the underlying programming language to make *every* memory access that the program can ever execute succeed. Experimental results indicate that, in practice, this technique enables programs to successfully survive otherwise fatal memory errors. And it provides an absolute guarantee that a memory error will never terminate the execution. Discarding out of bounds writes also helps eliminate data structure corruption errors that would otherwise occur on out of bounds writes. The standard way to apply failure-oblivious computing is to use a compiler to introduce the memory error checks (although it is, in principle, possible to apply the technique at the binary or even hardware level [23]).

Note that throwing exceptions on memory errors in the hope of invoking a developer-provided exception handler that can recover from the error is usually totally ineffective. In practice, the exception usually propagates up to the top-level exception handler, which terminates the program. And in general, it can be difficult for developers to provide effective exception handlers for unanticipated exceptions. Most memory errors are, of course, unanticipated — after all, if the developer had thought a memory error could occur, he or she would have written different code in the first place.

2.2 Loop Termination

Infinite loops threaten the survival of the program because they deny parts of the program access to a resource (the program counter) that they need to execute successfully. The following bounded loop technique completely eliminates infinite loops [24]:[1]

- **Training:** Execute the program for several training runs. For each loop, record the *observed maximum* — i.e., the maximum number of iterations the loop performed during any training run.
- **Iteration Bound:** Select a *slack factor* (typically a number around several thousand) and, for each loop, compute an *iteration bound* — i.e., the slack factor times the observed maximum for that loop. If the loop never executed during the training runs, simply use the maximum iteration bound for loops that did execute during training runs.
- **Iteration Bound Enforcement:** If necessary, transform the program to add an explicit loop counter to each loop. During production runs, use the loop counter to exit the loop whenever the number of iterations exceeds the iteration bound. If the loop exits after executing more iterations than the

[1] Of course, any program that is designed to execute for an unbounded period of time must have at least one unbounded loop (or other form of unbounded execution such as unbounded recursion). For example, many event-driven programs have an event-processing loop that waits for an event to come in, processes the event, then returns back to the top of the loop to wait for the next event. It is relatively straightforward for the developer to identify these loops and for the system to not apply the infinite loop termination technique to these loops.

observed maximum number of iterations from the training runs but fewer iterations than the iteration bound, update the iteration bound to be the slack factor times the number of iterations from the current execution of the loop.

2.3 Cyclic Memory Allocation

Like infinite loops, memory leaks are a form of unbounded resource consumption that enable one component to exhaust resources required for the program to survive. It is possible to completely eliminate memory leaks by allocating a fixed size buffer for each memory allocation site, then allocating objects cyclically out of that buffer [24]. Specifically, the nth object allocated at each site is allocated in slot n mod s, where s is the number of objects in the buffer. If the program only accesses at most the last s objects allocated at the corresponding site, this transformation does not affect the correctness of the program.

The potential disadvantage of cyclic memory allocation is that it may allocate multiple live objects in the same slot. The result is that writes to one object will overwrite other objects allocated in that same slot. Our experimental results indicate that overlaying live objects in this way typically causes the program to gracefully degrade rather than fail [24]. We attribute this property, in part, to the fact that the transformation (typically) preserves the type safety of the program.

2.4 Data Structure Repair

In the presence of failure-oblivious computing corrupted data structures are, by themselves, incapable of causing an outright program failure. They can, however, significantly disrupt the execution of the program and make it difficult for the program to execute acceptably. Data structure repair, which is designed to restore important consistency properties to damaged data structures, can significantly improve the ability of the program to deliver acceptable results [1,6,7]. We have found that this result holds even when the repair is unable to completely reconstruct information originally stored in the data structure (typically because the data structure corruption error destroyed the information before the repair was invoked).

3 Data Structure Consistency

Our experience with data structure repair and failure-oblivious computing shows that data structure consistency can be a critical component of meaningful program survival. We propose a multistage approach to the static verification of data structure consistency. The first stage is to obtain fully verified implementations of standard abstract data types. This goal has been largely achieved in Jahob program verification system [9,8,10]. The second stage is to verify properties that involve multiple abstract data types. Building on verified abstract data type implementations, the Hob project made significant progress towards this goal [25,26,27,28], but challenges still remain.

3.1 Jahob

Jahob is a general program verification system with a powerful specification language based on higher-order logic. Because the specifications take the form of specialized comments, it is possible to use standard Java implementation frameworks to execute specified programs. Given these specifications, the Jahob implementation processes the program to generate verification conditions. It then uses its underlying integrated reasoning system to prove these verification conditions.

3.2 The Jahob Specification Language

Jahob specifications use primitive types (such as integers and booleans), sets, and relations to characterize the abstract state of the data structure. A verified abstraction function establishes the correspondence between the concrete values that exist when the program executes (the implementation directly manipulates these values) and the abstract state in the specification (which exists only for verification purposes). Method contracts, class invariants, and annotations within method bodies use classical higher-order logic to express the desired properties of the data structure interface and implementation.

Specification Variables. In addition to concrete Java variables, Jahob supports *specification variables*, which do not exist during program execution but are useful to specify the behavior of methods without revealing the underlying data structure representation. In addition to other purposes, developers use specification variables to identify the abstract state of data structure implementations. Abstraction functions specify the connection between the concrete Java variables and the corresponding specification variables.

Method Contracts. A method contract in Jahob contains three parts: 1) a precondition, written as a requires clause, stating the properties of the program state and parameter values that must hold before a method is invoked; 2) a frame condition, written as a modifies clause, listing the components of the state that the method may modify (the remaining components remain unchanged); and 3) a postcondition, written as an ensures clause, describing the state at the end of the method (possibly defined relative to the parameters and state at the entry of the method). Jahob uses method contracts for assume/guarantee reasoning in the standard way. When analyzing a method m, Jahob assumes m's precondition and checks that m satisfies its postcondition and the frame condition. Dually, when analyzing a call to m, Jahob checks that the precondition of m holds, assumes that the values of state components from the frame condition of m change subject only to the postcondition of m, and that state components not in the frame condition of m remain unchanged. Public methods omit changes to the private state of their enclosing class and instead use public specification variables to describe how they change the state.

Class Invariants. A *class invariant* can be thought of as a boolean-valued specification variable that Jahob implicitly conjoins with the preconditions and postconditions of public methods. The developer can declare an invariant as private or public (the default annotation is private). Typically, a class invariant is private and is visible only inside the implementation of the class. Jahob conjoins the private class invariants of a class C to the preconditions and postconditions of methods declared in C. To ensure soundness in the presence of callbacks, Jahob also conjoins each private class invariant of class C to each reentrant call to a method m declared in a different class D. This policy ensures that the invariant C will hold if $D.m$ (either directly or indirectly) invokes a method in C. To make an invariant F with label l hold less often than given by this policy, the developer can write F as $b \rightarrow I$ for some specification variable b. To make F hold more often, the developer can use assertions with the shorthand (theinv l) that expand into F.

Loop Invariants. The developer states a loop invariant of a while loop immediately after the while keyword using the keyword invariant (or inv for short). Each loop invariant must hold before the loop condition and be preserved by each iteration of the loop. The developer can omit conditions that depend only on variables not modified in the loop — Jahob uses a simple syntactic analysis to conclude that the loop preserves such conditions.

3.3 The Jahob Integrated Proof Language

Conceptually, most reasoning systems search a proof space — they start with a set of known facts and axioms, then (at a high level) search the resulting proof space in an attempt to discover a proof of the desired consequent fact. We have found that, in practice, when automated reasoning systems fail, they fail because there are key choice points in the proof search space that are difficult for them to resolve successfully. We have therefore developed a proof language, integrated into the underlying imperative programming language, that enables the developer to resolve such choice points [9]. Examples of such choice points include lemma decompositions, case splits, universal quantifier instantiations, and witness identification for existentially quantified facts. We have also augmented the language with constructs that allow developers to structure proofs by contradiction and induction.

Although we focus on the constructs that developers use to resolve choice points, Jahob provides a full range of proof constructs. Developers can therefore provide as much or as little guidance as desired. It is even possible for a developer to use the integrated proof language to explicitly perform every step of the proof.

Finally, Jahob provides a construct that enables developers to deal with a pragmatic problem that arises with modern reasoning systems. In practice, these reasoning systems are very sensitive to the *assumption base* — the assumptions from which to prove the desired consequent fact. If the assumption base is too large or contains too many irrelevant assumptions, the search space becomes too difficult for the reasoning system to search effectively and it fails to find a proof.

Jahob therefore allows developers to name and identify a set of assumptions for the reasoning systems to use when attempting to prove a specific fact. The resulting precise identification of a minimal assumption base can often enable the reasoning systems to prove desired facts without any additional assistance. This capability can be especially important when verifying complex data structures (because the reasoning systems place many properties into the assumption base during the course of the verification).

3.4 Verification Condition Generation and Integrated Reasoning

Jahob produces verification conditions by simplifying the Java code and transforming it into extended guarded commands, then desugaring extended guarded commands into simple guarded commands, and finally generating verification conditions from simple guarded commands in a standard way [10,8]. It then splits the verification conditions into subformulas and, with the aid of formula approximation, uses the integrated automated reasoning systems to prove the subformulas [10,8]. It runs each automated reasoning system with a timeout and, on multicore machines, supports the invocation of multiple automated reasoning systems in parallel to prove a given subformula. Our current system verifies most of our data structure implementations within several minutes [8]. Our binary search tree implementation, with a verification time of an hour and forty-five minutes (primarily because of time spent in the MONA decision procedure), is an outlier.

3.5 Hob

Certain data structure consistency properties involve multiple data structures. The goal of the Hob project was to develop techniques for statically verifying such properties. Hob worked with set abstractions — it modelled the state of each data structure as an abstract set of objects. Developers could then use a standard set algebra (involving the boolean operators along with operations such as set inclusion and set difference) to state the desired consistency properties [25,26,27,28].

Although Hob contained techniques for verifying the consistency of individual data structure implementations, in the long run a more effective approach would be to first use a system like Jahob to verify the full functional correctness of individual data structure implementations, then use a more scalable system like Hob to verify properties involving multiple data structures. The Hob verification would, of course, work with the interfaces for the individual data structures (these interfaces would be verified by Jahob or a system like Jahob), not the data structure implementations. This approach would appropriately leverage the relative strengths of the two systems.

Our experience with Hob indicates that sets of objects provide a particularly compelling abstraction. In many domains it is possible to model user-level concepts with sets of objects; in these domains the resulting Hob specifications often capture properties that are directly relevant to the user. This relevance stands

in stark contrast to standard specifications, which tend to focus heavily on internal implementation details as opposed to high-level concepts that are directly meaningful to users.

Extending Hob to support relations would significantly increase its expressiveness and utility. This extension would enable Hob to support more sophisticated consistency constraints — for example, that one data structure contains every object in the domain of a map implemented by another data structure. The challenge with this extension is developing the scalable analyses required to verify these more expressive constraints.

3.6 Self-defending Data Structures

Many verification approaches (including Jahob and Hob) use assume/guarantee reasoning. Each procedure has a precondition that it assumes is true upon entry. It is the responsibility of the client to ensure that the precondition holds. Each procedure also has a postcondition, which the procedure implementation guarantees to be true assuming that the precondition holds upon entry to the procedure. Encapsulated invariants capture the key consistency properties.

With this approach, the data structure consistency depends both on the implementation and on the client — if the client fails to satisfy a precondition, there is no guarantee that the implementation will preserve any encapsulated invariants. Moreover, the standard preconditions for many abstract data types are often quite complicated and difficult to verify with the kind of scalable analyses required to successfully verify that clients correctly satisfy the preconditions.

We therefore advocate the use of *self-defending* data structures with empty preconditions (such data structures can, of course, have encapsulated invariants). Self-defending data structures contain all of the checks required to ensure that they remain consistent regardless of client behavior. The advantage is that such data structures completely eliminate the need for client analyses in the verification of the consistency of individual data structures.

4 Infinite Loops

In the last several years researchers have developed a set of techniques for statically verifying that loops terminate [29]. We expect that, in general, researchers will be able to develop techniques that are effective in proving that the vast majority of loops terminate. But we also expect certain kinds of loops to continue to be beyond the reach of any practically deployable lightweight static analysis. Two examples of such loops are loops that operate on linked data structures (the termination of these loops may depend on complex data structure consistency properties) and loops whose termination depends on the convergence of complex numerical algorithms.

Given the tractable size of abstract data types, we advocate the use of heavyweight formal reasoning techniques (much like those that the Jahob system supports) to prove termination properties. Because abstract data types are standard, widely used components, they can justify large formal reasoning efforts.

We anticipate that a similar approach would work for many numerical algorithms. But formal reasoning techniques for numerical algorithms are less developed than those for more discrete or symbolic algorithms. And in many cases properties such as termination depend on subtle mathematical properties of discretized representations of continuous quantities. It is unclear the extent to which formally verified termination proofs will become practical for widespread use. One potential solution is simply to place a predetermined bound on the number of iterations of each loop (in a manner similar to the loop termination technique described above in Section 2.2).

5 General Control Flow Anomalies

Infinite loops are a special case of more general control flow anomalies that can prevent the program from executing components required to provide acceptable service. For example, many event-driven programs have an event processing loop. In many cases it may be worthwhile to develop static analyses that reason about the control flow to prove that every execution will, in a finite amount of time, return back to the top of the event processing loop. We anticipate that the major complication (to the extent that there is one) will be reasoning about the behavior of the program in the face of exceptions and explicit program exits. Developer-provided assertions and safety checks can be especially counterproductive in this context. Developers are often overly conservative about the conditions they check — in our experience, programs are often able to survive violations of checks that developers have added to terminate the program in the face of unexpected executing conditions or state changes. One straightforward approach is to simply excise all calls to primitives that terminate the program [3].

6 Memory Errors

Reasonably simple and usable augmented type systems exist that are capable of statically guaranteeing the absence of null pointer dereferences [30]. Statically verifying the absence of out of bounds array accesses is a much more challenging and complex task, in large part because of the need to reason about (potentially complex) array indexing expressions [31]. Once again, we expect the most productive approach to involve isolating complex array indexing computations inside abstract data types or similar modules, then using potentially heavyweight sophisticated reasoning techniques to prove the absence of array bounds violations. With this approach, simpler and more scalable techniques should be able to verify the absence of array bounds violations in the remaining parts of the code.

7 Memory Leaks

Statically verifying the absence of memory leaks given standard program semantics is, with current program analysis and verification technology, the most

challenging task we consider in this paper. Techniques exist for finding certain classes of leaks in array-based data structure implementations [32]. Modified escape analyses should also be able to find leaks that occur when the program takes certain exceptional execution paths [33]. A key difficulty is that objects may remain reachable in linked data structures with the data structure interface enabling invocation sequences that can cause the computation to access the objects. But in some cases, because of restricted usage patterns in the data structure clients, these invocation sequences can never actually occur in the program as a whole. We are aware of no static analysis or verification techniques that are designed to operate successfully in this scenario. The difficulty of developing such techniques would depend heavily on the characteristics of the usage patterns that occur in practice.

8 Conclusion

Program survival is a key consideration in a world in which full program verification is unrealistic. Statically verifying key survival properties will involve scalable static analyses that operate over the vast majority of the program working in combination with sophisticated program verification technologies that leverage both heavyweight automated reasoning techniques and developer intervention to prove complex survival properties in targeted regions of the program. These technologies promise to significantly enhance our ability to deliver software systems that can successfully execute through errors to provide acceptable service to users.

References

1. Demsky, B., Rinard, M.: Data structure repair using goal-directed reasoning. In: Proceedings of the 2005 International Conference on Software Engineering (2005)
2. Rinard, M.: Acceptability-oriented computing. In: 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications Companion (OOPSLA 2003 Companion) Onwards! Session (October 2003)
3. Rinard, M., Cadar, C., Nguyen, H.H.: Exploring the acceptability envelope. In: 2005 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications Companion (OOPSLA 2005 Companion) Onwards! Session (October 2005)
4. Rinard, M., Cadar, C., Dumitran, D., Roy, D.M., Leu, T., William, S., Beebee, J.: Enhancing server availability and security through failure-oblivious computing. In: Proceeding of 6th Symposium on Operating System Design and Implementation (OSDI 2004) (2004)
5. Rinard, M., Cadar, C., Dumitran, D., Roy, D.M., Leu, T.: A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In: Proceedings of the 2004 Annual Computer Security Applications Conference (2004)
6. Demsky, B., Rinard, M.: Automatic detection and repair of errors in data structures. In: Proc. 18th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (2003)

7. Demsky, B., Rinard, M.: Static specification analysis for termination of specification-based data structure repair. In: IEEE International Symposium on Software Reliability (2003)
8. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 2008) (June 2008)
9. Zee, K., Kuncak, V., Rinard, M.: An integrated proof language for imperative programs. In: Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI 2009) (June 2009)
10. Kuncak, V.: Modular Data Structure Verification. PhD thesis, EECS Department, Massachusetts Institute of Technology (February 2007)
11. Weidenbach, C.: Combining superposition, sorts and splitting. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. II, pp. 1965–2013. Elsevier Science, Amsterdam (2001)
12. Schulz, S.: E – A Brainiac Theorem Prover. Journal of AI Communications 15(2/3), 111–126 (2002)
13. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
14. Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. In: Pfenning, F. (ed.) CADE 2007. LNCS, vol. 4603, pp. 167–182. Springer, Heidelberg (2007)
15. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
16. de Moura, L., Bjørner, N.: Efficient E-matching for SMT solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS, vol. 4603, pp. 183–198. Springer, Heidelberg (2007)
17. Henriksen, J., Jensen, J., Jørgensen, M., Klarlund, N., Paige, B., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019. Springer, Heidelberg (1995)
18. Ranise, S., Tinelli, C.: The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa (2006), `www.SMT-LIB.org`
19. Kuncak, V., Nguyen, H.H., Rinard, M.: Deciding Boolean Algebra with Presburger Arithmetic. J. of Automated Reasoning (2006), `http://dx.doi.org/10.1007/s10817-006-9042-1`
20. Kuncak, V., Rinard, M.: Towards efficient satisfiability checking for Boolean Algebra with Presburger Arithmetic. In: Pfenning, F. (ed.) CADE 2007. LNCS, vol. 4603, pp. 215–230. Springer, Heidelberg (2007)
21. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
22. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development– Coq'Art: The Calculus of Inductive Constructions. Springer, Heidelberg (2004)
23. Witchel, E., Rhee, J., Asanović, K.: Mondrix: Memory isolation for linux using mondriaan memory protection. In: 20th ACM Symposium on Operating Systems Principles (SOSP-20) (2005)
24. Nguyen, H.H., Rinard, M.: Detecting and eliminating memory leaks using cyclic memory allocation. In: Proceedings of the 2007 International Symposium on Memory Management (2007)
25. Lam, P., Kuncak, V., Rinard, M.: Cross-cutting techniques in program specification and analysis. In: 4th International Conference on Aspect-Oriented Software Development (AOSD 2005) (2005)

26. Kuncak, V., Lam, P., Zee, K., Rinard, M.: Modular pluggable analyses for data structure consistency. IEEE Transactions on Software Engineering 32(12) (December 2006)
27. Lam, P., Kuncak, V., Rinard, M.: Generalized typestate checking for data structure consistency. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 430–447. Springer, Heidelberg (2005)
28. Lam, P.: The Hob System for Verifying Software Design Properties. PhD thesis, Massachusetts Institute of Technology (February 2007)
29. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI 2006) (June 2006)
30. Papi, M.M., Ali, M., Correa Jr., T.L., Perkins, J.H., Ernst, M.D.: Practical pluggable types for java. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis, Seattle, WA (July 2008)
31. Rugina, R., Rinard, M.C.: Symbolic bounds analysis of pointers, array indices, and accessed memory regions. ACM Trans. Program. Lang. Syst. 27(2) (2005)
32. Shaham, R., Kolodner, E., Sagiv, S.: Automatic removal of array memory leaks in java. In: Watt, D.A. (ed.) CC 2000. LNCS, vol. 1781, p. 50. Springer, Heidelberg (2000)
33. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for Java programs. In: OOPSLA, Denver (November 1999)

Superposition and Model Evolution Combined

Peter Baumgartner[1] and Uwe Waldmann[2]

[1] NICTA* and Australian National University, Canberra, Australia
Peter.Baumgartner@nicta.com.au
[2] MPI für Informatik, Saarbrücken, Germany
uwe@mpi-inf.mpg.de

Abstract. We present a new calculus for first-order theorem proving with equality, $\mathcal{ME}$+Sup, which generalizes both the Superposition calculus and the Model Evolution calculus (with equality) by integrating their inference rules and redundancy criteria in a non-trivial way. The main motivation is to combine the advantageous features of both—rather complementary—calculi in a single framework. For instance, Model Evolution, as a lifted version of the propositional DPLL procedure, contributes a non-ground splitting rule that effectively permits to split a clause into *non* variable disjoint subclauses. In the paper we present the calculus in detail. Our main result is its completeness under semantically justified redundancy criteria and simplification rules.

1 Introduction

We present a new calculus for first-order theorem proving with equality, $\mathcal{ME}$+Sup, which generalizes both the Superposition calculus and the Model Evolution calculus (with equality), $\mathcal{ME}_{\mathrm{E}}$. It integrates the inference rules of Superposition and of Model Evolution in a non-trivial way while preserving the individual semantically-based redundancy criteria. The inference rules are controlled by a rather flexible labelling function on atoms. This permits non-trivial combinations where inference rule applicability is disjoint, but pure forms of both calculi can be (trivially) configured, too.

On a research-methodological level, this paper attempts to bridge the gap between instance-based methods (per $\mathcal{ME}_{\mathrm{E}}$) and Resolution methods (per Superposition). Both methods are rather successful, for instance in terms of performance of implemented systems at the annual CASC theorem proving competition. However, they currently stand rather separated. They provide decision procedure for different sub-classes of first-order logic, and their inference rules are incompatible, too. For instance, subsumption deletion can be used with instance-based methods in only a limited way.

The main motivation for this work is to combine the advantages of both calculi in a single framework. Technically, $\mathcal{ME}$+Sup can be seen as an extension of the

* NICTA is funded by the Australian Government's *Backing Australia's Ability* initiative.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 17–34, 2009.

essential Model Evolution inference rules by Superposition inference rules. Alternatively, $\mathcal{ME}$+Sup can be seen to extend Superposition with a new splitting rule that permits, as a special case, to split a clause into *non* variable disjoint subclauses, which is interesting, e.g., to obtain a decision procedure for function-free clause logic. It seems not too difficult to extend current Superposition theorem provers with the new splitting rule, in particular those that already provide infrastructure for a weaker form of splitting (such as SPASS [6]). Finally, another motivation for this work is to simplify the presentation of $\mathcal{ME}_{\mathrm{E}}$ by aligning it with the better-known superposition framework. The following clause set is prototypical for the intended applications of $\mathcal{ME}$+Sup (function symbols are typeset in sans-serif and variables in *italics*).

(1)	$x \leq z \vee \neg(x \leq y) \vee \neg(y \leq z)$	(4)	$\mathsf{select}(\mathsf{store}(a,i,e),i) \approx e$
(2)	$x \leq y \vee y \leq x$	(5)	$\mathsf{select}(\mathsf{store}(a,i,e),j) \approx \mathsf{select}(a,j) \vee i \approx j$
(3)	$x \approx y \vee \neg(x \leq y) \vee \neg(y \leq x)$	(6)	$i \leq j \vee \neg(\mathsf{select}(\mathsf{a0},i) \leq \mathsf{select}(\mathsf{a0},j))$

The clauses (1)-(3) axiomatize a total order, clauses (4)-(5) axiomatize arrays, and clause (6) says that the array a0 is sorted and that there are no duplicates in a0 (the converse of (6), $\neg(i \leq j) \vee \mathsf{select}(\mathsf{a0},i) \leq \mathsf{select}(\mathsf{a0},j)$, is entailed by (1)-(3),(6)). This clause set is satisfiable, but Superposition equipped with standard redundancy criteria (with or without selection of negative literals) does not terminate on these clauses. This is, essentially, because the length of the clauses derived cannot be bounded.The clauses (1) and (2) are enough to cause non-termination, and $\mathcal{ME}_{\mathrm{E}}$ does not terminate on (1)-(6) either. However, $\mathcal{ME}$+Sup does terminate when all $\leq$-atoms are labelled as "split atoms" and all other atoms are "superposition atoms".[1] Intuitively, the $\mathcal{ME}$-part of $\mathcal{ME}$+Sup takes care of computing a model for the split atoms through a *context*, the main data structure of $\mathcal{ME}$ to represent interpretations, and the Superposition part of $\mathcal{ME}$+Sup takes care of (implicitly) computing a model for the superposition atoms.

To demonstrate how $\mathcal{ME}$+Sup can be used to effectively provide a new splitting rule for Superposition consider the clauses (1) and (2) from above. Let us now "split" clause (1) into two non-variable disjoint clauses by introducing a name s:

$$\text{(1a)} \quad x \leq z \vee \neg(x \leq y) \vee \neg\mathsf{s}(y,z) \qquad \text{(1b)} \quad \mathsf{s}(y,z) \vee \neg(y \leq z)$$

Now declare all $\leq$-atoms as superposition atoms and all s-atoms as split atoms. Further, all s-atoms must be strictly greater than all $\leq$-atoms (this can be achieved using standard orderings and using a two-sorted signature). In effect then, resolution and factoring inferences are all blocked on clauses that contain s-literals, as the usual maximality restrictions for resolution and factorisation apply in $\mathcal{ME}$+Sup, too. Therefore, only factorisation is applicable, to clause (2), yielding $x \leq x$. The only inference rule that is applicable now is Neg-U-Res,

[1] In general, split atoms can be equations, too, and the signatures of the split and the superposition atoms need not be disjoint. We intended to keep the examples simple.

which gives $\neg(y \leq z) \cdot \neg\mathsf{s}(y,z)$. (This is a *constrained clause*, a pair $C \cdot \Gamma$, where C is a clause and the constraints Γ are split atoms or their negation.) That is, $\mathsf{s}(y,z)$ has been shifted into the constraint part, put aside for later processing by $\mathcal{ME}$ rules. The literal $\neg(y \leq z)$ is now maximal in $\neg(y \leq z) \cdot \neg\mathsf{s}(y,z)$, and resolution between this clause and (2) gives $z \leq y \cdot \neg\mathsf{s}(y,z)$. Similarly, resolution between $\neg(y \leq z) \cdot \neg\mathsf{s}(y,z)$ and $x \leq x$ gives the constrained empty clause $\Box \cdot \neg\mathsf{s}(x,x)$. This does not make a refutation, because a model that assigns true to $\mathsf{s}(x,x)$, and hence falsifies the constraint, has not been excluded. Indeed, to constrained empty clauses the $\mathcal{ME}$-style split rule is applicable, resulting in two cases (contexts), with $\mathsf{s}(x,x)$ and $\neg\mathsf{s}(x,x)$, respectively. Notice this is a *non*-ground splitting. The derivation stops at this point, as no inference rule is applicable, and $\mathsf{s}(x,x)$ specifies a model. The other case with $\neg\mathsf{s}(x,x)$ can be used to derive the empty clause $\Box \cdot \emptyset$, which stands for "false".

Related Work. $\mathcal{ME}$+Sup subsumes the Superposition calculus [2] and its redundancy concept and also the essentials of propositional DPLL, that is, split and unit propagation. Model Evolution [3] and Model Evolution with Equality [4] are not completely covered, though, since universal literals and some optional inference rules are missing. The model construction that we use has some similarity with the one used for Constraint Superposition [5], where one also starts with constructing a model for reduced instances and later extends this to the full clause set provided that this is constraint-free.

2 Formal Preliminaries

We consider signatures Σ comprised of a binary predicate symbol $\approx$ (equality), and a finite set of function symbols of given arity (constants are 0-ary function symbols). We also need a denumerable set of variables X disjoint from Σ. Terms (over Σ and X) are defined as usual. If t is a term we denote by $\mathcal{V}ar(t)$ the set of t's variables. A term t is *ground* iff $\mathcal{V}ar(t) = \emptyset$. A *substitution* is a mapping of variables to terms that is the identity almost everywhere. We write $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ for the substitution that maps the variable x_i to the term t_i, for $i = 1, \ldots, n$. The application of a substitution to a term t is written as $t\sigma$. A *renaming* is a substitution that is a bijection of X onto itself. We write $s \gtrsim t$, iff there is a substitution σ such that $s\sigma = t$.[2] We say that s is *a variant of* t, and write $s \sim t$, iff $s \gtrsim t$ and $t \gtrsim s$. We write $s \gtrsim\!\!\!\!\!\!\cdot\ \ t$ if $s \gtrsim t$ but $s \not\sim t$. The notation $s[t]_p$ means that the term t occurs in the term s at position p, as usual. The index p is left away when not important or clear from the context. Because equality is the only predicate symbol, an *atom* is always an equation $s \approx t$, which is identified with the multiset $\{s,t\}$. Consequently, equations are treated symmetrically, as $s \approx t$ and $t \approx s$ denote the same multiset. A literal is an atom (a *positive literal*) or the negation of an atom (a *negative literal*). Negative literals are generally written $s \not\approx t$ instead of $\neg(s \approx t)$. In the examples below we often write a non-equational literal like $P(t_1, \ldots, t_n)$, which is meant

[2] Note that many authors would write $s \lesssim t$ in this case.

to stand for the equation $P(t_1, \ldots, t_n) \approx \mathrm{tt}$, where tt is a fresh constant that is smaller than all other terms, and similarly for negative literals. We write $\overline{L}$ to denote the complement of a literal L, i.e. $\overline{A} = \neg A$ and $\overline{\neg A} = A$, for any atom A. A *clause* is a multiset of literals $\{L_1, \ldots, L_n\}$, generally written as a disjunction $L_1 \vee \cdots \vee L_n$. We write $L \vee C$ to denote the clause $\{L\} \cup C$. The empty clause is written as $\Box$. All the notions on substitutions above are extended from terms to atoms, literals and clauses in the obvious way.

Orderings. We suppose as given a reduction ordering $\succ$ that is total on ground Σ-terms.[3] Following usual techniques [2,5, e.g.], it is extended to an ordering on literals by taking a positive literal $s \approx t$ as the multiset $\{s, t\}$, a negative literal $s \not\approx t$ as the multiset $\{s, s, t, t\}$ and using the extension of $\succ$ to multisets of terms to compare literals. Similarly, clauses are compared by the multiset extension of the ordering on literals. All these (strict, partial) orderings will be denoted by the same symbol, $\succ$. The non-strict orderings $\succeq$ are defined as $s \succeq t$ iff $s \succ t$ or $s = t$. We say that a literal L is *maximal* (*strictly maximal*) in a clause $L \vee C$ iff there is no $K \in C$ with $K \succ L$ ($K \succeq L$).

Rewrite Systems. A *(rewrite) rule* is an expression of the form $l \to r$ where l and r are terms. A *rewrite system* is a set of rewrite rules. We say that a rewrite system R is *ordered by* $\succ$ iff $l \succ r$, for every rule $l \to r \in R$. In this paper we consider only (ground) rewrite systems that are ordered by $\succ$. A term t is *reducible by* $l \to r$ iff $t = t[l]_p$ for some position p, and t is *reducible wrt.* R if it is reducible by some rule in R. The notion *irreducible* means "not reducible". A rewrite system R is *left-reduced* (*fully reduced*) iff for every rule $l \to r \in R$, l is (l and r are) irreducible wrt. $R \setminus \{l \to r\}$. In other words, in a fully reduced rewrite system no rule is reducible by another rule, neither its left hand side *nor its right hand side.*

Interpretations. A *(Herbrand) interpretation* I is a set of ground atoms—exactly those that are true in the interpretation. Validity of ground literals, ground clauses, and clause sets in a Herbrand interpretation is defined as usual. We write $I \models F$ to denote the fact that I satisfies F, where F is a ground literal or a clause (set), which stands for the set of all its ground instances (of all clauses in the set). An *E-interpretation* is an interpretation that is also a congruence relation on the ground terms. If I is an interpretation, we denote by $I^\star$ the smallest congruence relation on all ground terms that includes I, which is an E-interpretation. We say that I *E-satisfies* F iff $I^\star \models F$. We say that F *E-entails* F', written $F \models F'$, iff every E-interpretation that satisfies F also satisfies F'.

The above notions are applied to ground rewrite systems instead of interpretations by taking the rules as equations. We write $R^\star \models F$ and mean $\{l \approx r \mid l \to r \in R\}^\star \models F$. It is well-know that any left-reduced (and hence any fully

[3] A *reduction ordering* is a strict partial ordering that is well-founded and is closed unter context i.e., $s \succ s'$ implies $t[s] \succ t[s']$ for all terms t, and liftable, i.e., $s \succ t$ implies $s\delta \succ t\delta$ for every term s and t and substitution δ.

reduced) ordered rewrite system R is convergent,[4] see e.g. [1]) and that any ground equation $s \approx t$ is E-satisfied by R, i.e., $R^\star \models s \approx t$ if and only if s and t have the same (unique) normal form wrt. R.

Labelling Function. Broadly speaking, $\mathcal{ME}$+Sup combines inference rules from the Superposition calculus and inference rules resembling those of Model Evolution, but for each atom only a subset of the full set of inference rules is usable. This is controlled by assuming a *labelling function* that partitions the set of positive ground atoms into two sets, the *split atoms* and the *superposition atoms*.[5] We say a (possibly non-ground) atom is a *split atom* (*superposition atom*) iff at least one ground instance is a split atom (superposition atom).

Thus, while a ground atom is either one or the other, the distinction is blurred for non-ground atoms. From a practical point of view, to avoid overlap between the $\mathcal{ME}$ and the superposition inference rules, it is desirable to keep the (non-ground) split atoms and superposition atoms as separate as possible.

The separation into split atoms and superposition atoms is quite flexible. No assumptions regarding disjointness of their underlying signatures or ordering assumptions between their elements are required. For instance, one may declare all ground atoms up to a certain term depth as split atoms. Even the set of non-ground split atoms is finite then, modulo renaming. As will become clear, the *contexts* evolved by the Model Evolution part of $\mathcal{ME}$+Sup are finite then, which might be interesting, e.g., to finitely represent (parts of) a counter-example for non-theorems.

3 Contexts

Contexts have been introduced in [3] as the main data structure in the Model Evolution calculus to represent interpretations; they have been adapted to the equality case in [4], but here we work with the original definition, which is simpler and more practical. More formally, when l and r are terms, a *rewrite literal* is a rule $l \to r$ or its negation $\neg(l \to r)$, the latter generally written as $l \not\to r$. By treating $\to$ as a predicate symbol, all operations defined on equational literals apply to rewrite literals as well. In particular, $\overline{l \to r} = l \not\to r$ and $\overline{l \not\to r} = l \to r$. If clear from the context, we use the term "literal" to refer to equational literals as introduced earlier or to rewrite literals.

A *context* is a set of rewrite literals that also contains a pseudo-literal $\neg x$, for some variable x. In examples we omit writing $\neg x$ and instead implicitly assume it is present. A non-equational literal $P(t_1, \ldots, t_n)$ in a context stands for $P(t_1, \ldots, t_n) \to \mathrm{tt}$, and similarly for negative literals. Where L is a rewrite literal and Λ a context, we write $L \in_\sim \Lambda$ if L is a variant of a literal in Λ. A rewrite literal L is *contradictory with a context* Λ iff $\overline{L} \in_\sim \Lambda$. A context Λ is *contradictory* iff it contains a rewrite literal that is contradictory with Λ. For

[4] A rewrite system is convergent iff it is confluent and terminating.

[5] Notice that with the symmetric treatment of equations, $l \approx r$ is a split atom if and only if $r \approx l$ is, and similarly for superposition atoms.

instance, if $\Lambda = \{f(x) \to a, f(x) \not\to x\}$ then $f(y) \not\to a$ and $f(y) \to y$ are contradictory with Λ, while $f(a) \to a$, $a \not\to f(x)$ and $f(x) \to y$ are not. From now on we assume that all contexts are non-contradictory. This is justified by the fact that the $\mathcal{ME}$+Sup calculus defined below can derive non-contradictory contexts only.

A context stands for its *produced* literals, defined as follows:

Definition 3.1 (Productivity [3]). *Let L be a rewrite literal and Λ a context. A rewrite literal K* produces *L in Λ iff $K \gtrsim L$ and there is no $K' \in \Lambda$ such that $K \gtrsim \overline{K'} \gtrsim L$. The context Λ* produces *L iff it contains a literal K that produces L in Λ, and Λ* produces *a multiset Γ of rewrite literals iff Λ produces each $L \in \Gamma$.*

For instance, the context Λ above produces $f(b) \to a$, $f(a) \to a$ and $f(a) \not\to a$, but Λ produces neither $f(a) \to b$ nor $a \to f(x)$.

For the model construction in Section 7 we will need the set of positive ground rewrite rules produced by Λ, $\Pi_\Lambda := \{l \to r \mid \Lambda \text{ produces } l \to r \text{ and } l \to r \text{ is ground}\}$. For instance, if $\Lambda = \{f(x) \to x\}$ and Σ consists of a constant a and the unary function symbol f then $\Pi_\Lambda = \{f(a) \to a, f(f(a)) \to f(a), \ldots\}$. We note that productivity of rewrite literals corresponding to *split* atoms only is relevant for the calculus.

4 Constrained Clauses

Let $C = L_1 \vee \cdots \vee L_n$ be a clause, let $\Gamma = \{K_1, \ldots, K_m\}$ be a multiset of rewrite literals such that no K_i is of the form $x \to t$, where x is a variable and t is a term. The expression $C \cdot \Gamma$ is called a *constrained clause (with constraint Γ)*, and we generally write $C \cdot K_1, \ldots, K_m$ instead of $C \cdot \{K_1, \ldots, K_m\}$. The notation $C \cdot \Gamma, K$ means $C \cdot \Gamma \cup \{K\}$.[6]

Applying a substitution σ to $C \cdot \Gamma$, written as $(C \cdot \Gamma)\sigma$, means to apply σ to C and all literals in Γ. A constrained clause $C \cdot \Gamma$ is *ground* iff both C and Γ are ground. For a set of constrained clauses Φ, Φ^{gr} is the set of all ground instances of all elements in Φ.

Constraints are compared in a similar way as clauses by taking the multiset extension of a (any) total ordering on ground rewrite literals. Constrained clauses then are compared lexicographically, using first the clause ordering introduced earlier to compare the clause components, and then using the ordering on constraints. Again we use the symbol $\succ$ to denote this (strict) ordering on constrained clauses. It follows with well-known results that $\succ$ is total on ground constrained clauses. Observe that this definition has the desirable property that proper subsumption among constrained clauses is always order-decreasing (the subsuming constrained clause is smaller).

For the soundness proof of $\mathcal{ME}$+Sup we need the *clausal form* of a constrained clause $C \cdot \Gamma = L_1 \vee \cdots \vee L_m \cdot l_1 \to r_1, \ldots, l_k \to r_k, l_{k+1} \not\to r_{k+1}, \ldots, l_n \not\to r_n$,

[6] As will become clear later, literals $x \to t$ can never occur in constraints, because, in essence, paramodulation into variables is unnecessary.

which is the ordinary clause $L_1 \vee \cdots \vee L_m \vee l_1 \not\approx r_i \vee \cdots \vee l_k \not\approx r_k \vee l_{k+1} \approx r_{k+1} \vee \cdots \vee l_n \approx r_n$ and which we denote by $(C \cdot \Gamma)^{\mathrm{c}}$. From a completeness perspective, however, a different reading of constrained clauses is appropriate. The clause part C of a (ground) constrained clause $C \cdot \Gamma$ is evaluated in an E-interpretation I, whereas the literals in Γ are evaluated wrt. a context Λ in terms of productivity. The following definition makes this precise.

We say that a ground constraint Γ *consists of split rewrite literals* iff $l \approx r$ is a split atom and $l \succ r$, for every $l \to r \in \Gamma$ or $l \not\to r \in \Gamma$. A possibly non-ground constraint Γ *consists of split rewrite literals* if some ground instance of Γ does.

Definition 4.1 (Satisfiaction of Constrained Clauses). *Let Λ be a context, I an E-Interpretation and $C \cdot \Gamma$ a ground constrained clause. We say that Λ* satisfies *Γ and write $\Lambda \models \Gamma$ iff Γ consists of split rewrite literals and Λ produces Γ. We say that the pair (Λ, I)* satisfies *$C \cdot \Gamma$ and write $\Lambda, I \models C \cdot \Gamma$ iff $\Lambda \not\models \Gamma$ or $I \models C$.*

The pair (Λ, I) *satisfies* a possibly non-ground constrained clause (set) F, written as $\Lambda, I \models F$ iff (Λ, I) satisfies all ground instances of (all elements in) F. For a set of constrained clauses Φ we say that Φ *entails* $C \cdot \Gamma$ *wrt.* Λ, and write $\Phi \models_\Lambda C \cdot \Gamma$ iff for every E-interpretation I it holds $\Lambda, I \not\models \Phi$ or $\Lambda, I \models C \cdot \Gamma$.

The definitions above are also applied to pairs (Λ, R), where R is a rewrite system, by implicitly taking $(\Lambda, R^\star)$. Indeed, in the main applications of Definition 4.1 such a rewrite system R will be determined by the model construction in Section 7 below.

Example 4.2. Let $\Lambda = \{f(x) \to x, f(c) \not\to c\}$, $R = \{f(a) \to a, f(b) \to b\}$ and $C \cdot \Gamma = f(f(a)) \approx x \cdot f(x) \to x$. Let $\gamma_a = \{x \mapsto a\}$, $\gamma_b = \{x \mapsto b\}$ and $\gamma_c = \{x \mapsto c\}$. Suppose that all (ground) atoms are split atoms. Notice that $\Gamma\gamma_a$, $\Gamma\gamma_b$ and $\Gamma\gamma_c$ consist of split rewrite literals. Then, $R \models \Gamma\gamma_a$, as Λ produces $\{f(a) \to a\}$ and so we need to check $R^\star \models f(f(a)) \approx a$, which is the case, to conclude $\Lambda, R \models (C \cdot \Gamma)\gamma_a$. As $R \models \Gamma\gamma_b$ but $R^\star \not\models f(f(a)) \approx b$ we have $\Lambda, R \not\models (C \cdot \Gamma)\gamma_a$. Finally, Λ does not produce $\{f(c) \to c\}$, and with $\Lambda \not\models \Gamma\gamma_c$ it follows $\Lambda, R \models (C \cdot \Gamma)\gamma_c$.

5 Inference Rules on Constrained Clauses

We are going to define several inference rules on constrained clauses, which will be embedded into the $\mathcal{ME}$+Sup calculus below.

$$\mathsf{Ref}\ \frac{s \not\approx t \vee C \cdot \Gamma}{(C \cdot \Gamma)\sigma}$$

where (i) σ is a mgu of s and t, and (ii) $(s \not\approx t)\sigma$ is maximal in $(s \not\approx t \vee C)\sigma$.

The next three rules combine a rewrite literal, which will be taken from a current context, and a constrained clause, which will be taken from a current clause set.

$$\mathsf{U\text{-}Sup\text{-}Neg}\ \frac{l \to r \qquad s[u]_p \not\approx t \vee C \cdot \Gamma}{(s[r]_p \not\approx t \vee C \cdot \Gamma, l \to r)\sigma}$$

where (i) σ is a mgu of l and u, (ii) u is not a variable, (iii) $(l \approx r)\sigma$ is a split atom, (iv) $r\sigma \not\succeq l\sigma$, (v) $t\sigma \not\succeq s\sigma$, and (vi) $(s \not\approx t)\sigma$ is maximal in $(s \not\approx t \vee C)\sigma$.

$$\textsf{U-Sup-Pos}\ \frac{l \to r \qquad s[u]_p \approx t \vee C \cdot \Gamma}{(s[r]_p \approx t \vee C \cdot \Gamma, l \to r)\sigma}$$

where (i) σ is a mgu of l and u, (ii) u is not a variable, (iii) $(l \approx r)\sigma$ is a split atom, (iv) $r\sigma \not\succeq l\sigma$, and if $(s \approx t)\sigma$ is a split atom then (v-a) $(s \approx t)\sigma$ is maximal in $(s \approx t \vee C)\sigma$ else (v-b) $t\sigma \not\succeq s\sigma$ and $(s \approx t)\sigma$ is strictly maximal in $(s \approx t \vee C)\sigma$, and (vi) if $l\sigma = s\sigma$ then $r\sigma \not\succeq t\sigma$.

U-Sup-Pos and U-Sup-Neg are the only rules that create new rewrite literals $(l \to r)\sigma$ in the constraint part (Sup-Neg and Sup-Pos below only merge existing constraints). Notice that because u is not a variable, in both cases $l\sigma$ is not a variable, even if l is. It follows easily that all expressions $C \cdot \Gamma$ derivable by the calculus are constrained clauses.

$$\textsf{Neg-U-Res}\ \frac{\neg A \qquad s \approx t \vee C \cdot \Gamma}{(C \cdot \Gamma, s \not\to t)\sigma}$$

where $\neg A$ is a pseudo literal $\neg x$ or a negative rewrite literal $l \not\to r$, and (i) $(s \approx t)\sigma$ is a split atom, (ii) σ is a mgu of A and $s \to t$, (iii) $(s \approx t)\sigma$ is a split atom, (iv) $t\sigma \not\succeq s\sigma$, and (v) $(s \approx t)\sigma$ is maximal in $(s \approx t \vee C)\sigma$.

The following three rules are intended to be applied to clauses from a current clause set. To formulate them we need one more definition: let $l \approx r$ be an equation and $C = x_1 \approx t_1 \vee \cdots \vee x_n \approx t_n$ a (possibly empty) clause of positive literals, where x_i is a variable and t_i a term, for all $i = 1, \ldots, n$. We say that a substitution π *merges* C *with* $l \approx r$ iff π is an mgu of $l, x_1, \ldots, x_n$, $r\pi \not\succeq l\pi$, and $t_i\pi \not\succeq l\pi$.

$$\textsf{Sup-Neg}\ \frac{l \approx r \vee C' \cdot \Gamma' \qquad s[u]_p \not\approx t \vee C \cdot \Gamma}{(s[r]_p \not\approx t \vee C \vee C' \cdot \Gamma, \Gamma')\sigma\pi}$$

where (i) σ is a mgu of l and u, (ii) u is not a variable, (iii) π merges $x_1 \approx t_1 \vee \cdots \vee x_n \approx t_n \subseteq C'\sigma$ with $(l \approx r)\sigma$, (iv) $\{x_1, \ldots, x_n\} \subseteq \mathcal{V}ar(\Gamma'\sigma)$, (v) $(l \approx r)\sigma$ is a superposition atom, (vi) $r\sigma\pi \not\succeq l\sigma\pi$, (vii) $(l \approx r)\sigma\pi$ is strictly maximal in $(l \approx r \vee C')\sigma\pi$, (viii) $t\sigma \not\succeq s\sigma$, and (ix) $(s \not\approx t)\sigma$ is maximal in $(s \not\approx t \vee C)\sigma$.

The need for merge substitutions is demonstrated in Example 7.4 below.

$$\textsf{Sup-Pos}\ \frac{l \approx r \vee C' \cdot \Gamma' \qquad s[u]_p \approx t \vee C \cdot \Gamma}{(s[r]_p \approx t \vee C \vee C' \cdot \Gamma, \Gamma')\sigma\pi}$$

where (i) σ is a mgu of l and u, (ii) u is not a variable, (iii) π merges $x_1 \approx t_1 \vee \cdots \vee x_n \approx t_n \subseteq C'\sigma$ with $(l \approx r)\sigma$, (iv) $\{x_1, \ldots, x_n\} \subseteq \mathcal{V}ar(\Gamma'\sigma)$, (v) $(l \approx r)\sigma$ is a superposition atom, (vi) $r\sigma\pi \not\succeq l\sigma\pi$, (vii) $(l \approx r)\sigma\pi$ is strictly maximal in $(l \approx r \vee C')\sigma\pi$, and if $(s \approx t)\sigma$ is a split atom then (viii-a) $(s \approx t)\sigma$ is maximal in $(s \approx t \vee C)\sigma$ else (viii-b) $t\sigma \not\succeq s\sigma$ and $(s \approx t)\sigma$ is strictly maximal in $(s \approx t \vee C)\sigma$.

Notice that $(s \approx t)\sigma$ could be both a split atom and a superposition atom. In this case the weaker condition (viii-a) is used to take care of a ground instance of a Sup-Pos inference applied to a split atom, which requires the weaker condition.

In both Sup-Neg and Sup-Pos inference rules we assume the additional condition $\mathcal{C}\sigma\pi \not\succeq \mathcal{D}\sigma\pi$, where by $\mathcal{C}$ and $\mathcal{D}$ we mean their left and right premise, respectively.

$$\mathsf{Fact}\ \frac{l \approx r \vee s \approx t \vee C \cdot \Gamma}{(l \approx t \vee r \not\approx t \vee C \cdot \Gamma)\sigma}$$

where (i) σ is an mgu of l and s, (ii) $(l \approx r)\sigma$ is a superposition atom, (iii) $(l \approx r)\sigma$ is maximal in $(l \approx r \vee s \approx t \vee C)\sigma$, (iv) $r\sigma \not\succeq l\sigma$, and (v) $t\sigma \not\succeq s\sigma$.

In each of the inference rules above we assume the additional condition that $\Gamma\sigma$ ($\Gamma\sigma\pi$ and $\Gamma'\sigma\pi$ in case of Sup-Neg or Sup-Pos) consists of split rewrite literals.

An *inference system* ι is a set of inference rules. By an ι *inference* we mean an instance of an inference rule from ι such that all conditions are satisfied. An inference is *ground* if all its premises and the conclusion are ground.

The *base inference system* ι_{Base} consists of Ref, Fact, U-Sup-Neg, U-Sup-Pos, Neg-U-Res, Sup-Neg and Sup-Pos. If from a given ι_{Base} inference a ground ι_{Base} inference results by applying a substitution γ to all premises and the conclusion, we call the resulting ground inference a *ground instance via* γ *(of the* ι_{Base} *inference).* This is not always the case, as, e.g., ordering constraints can become unsatisfiable after application of γ. An important consequence of the ordering restrictions stated with the inference rules is that the conclusion of a ground ι_{Base} inference is always strictly smaller than the right or only premise.

6 Inference Rules on Sequents

Sequents are the main objects manipulated by the $\mathcal{ME}$+Sup calculus. A *sequent* is a pair $\Lambda \vdash \Phi$ where Λ is a context and Φ is a set of constrained clauses. The following inference rules extend the inference rules ι_{Base} above to sequents.

$$\mathsf{Deduce}\ \frac{\Lambda \vdash \Phi}{\Lambda \vdash \Phi, C \cdot \Gamma}$$

if one of the following cases applies:

- $C \cdot \Gamma$ is the conclusion of a Ref or Fact inference with a premise from Φ.
- $C \cdot \Gamma$ is the conclusion of a U-Sup-Neg, U-Sup-Pos or Neg-U-Res inference with a right premise from Φ and a left premise $K \in \Lambda$ that produces $K\sigma$ in Λ, where σ is the mgu used in that inference.
- $C \cdot \Gamma$ is the conclusion of a Sup-Neg or Sup-Pos inference with both premises from Φ.

In each case the second or only premise of the underlying ι_{Base} inference is called the *selected clause (of a* Deduce *inference)*. In inferences involving two premises, a fresh variant of the, say, right premise is taken, so that the two premises are variable disjoint.

$$\textsf{Split}\ \frac{\Lambda \vdash \Phi}{\Lambda,\ \overline{K} \vdash \Phi \qquad \Lambda,\ K \vdash \Phi}$$

if there is a constrained clause $\Box \cdot \Gamma \in \Phi$ such that (i) $K \in \Gamma$, (ii) $s \approx t$ is a split atom, where $K = s \to t$ or $K = s \not\to t$, and (iii) neither K nor $\overline{K}$ is contradictory with Λ. A Split inference is *productive* if Λ produces Γ; the clause $\Box \cdot \Gamma$ is called the *selected clause (of the* Split *inference)*.

The intuition behind Split is to make a constrained empty clause $\Box \cdot \Gamma$ true, which is false when Λ produces Γ (in the sense of Definition 4.1). This is achieved by adding $\overline{K}$ to the current context. For example, if $\Lambda = \{P(a,y), \neg P(x,b)\}$ and $\Box \cdot \Gamma = \Box \cdot P(a,b)$ then a (productive) Split inference will give $\{P(a,y), \neg P(x,b), \neg P(a,b)\}$, which no longer produces $P(a,b)$. Intuitively, the calculus tries to "repair" the current context towards a model for a constrained empty clause.

Notice that a Split inference can never add a rewrite to a context that already contains a variant of it or its complement, as this would contradict condition (iii).[7] Because of the latter property the calculus will never derive contradictory contexts.

$$\textsf{Close}\ \frac{\Lambda \vdash \Phi}{\Lambda \vdash \Phi, \Box \cdot \emptyset}$$

if there is a constrained clause $\Box \cdot \Gamma \in \Phi$ such that $L \in_\sim \Lambda$ for every $L \in \Gamma$. The clause $\Box \cdot \Gamma$ is called the *selected clause (of a* Close *inference)* and the variants of the L's in Λ are the *closing literals*. A sequent $\Lambda \vdash \Phi$ is *closed* if Φ contains $\Box \cdot \emptyset$. The purpose of Close is to abandon a sequent that cannot be "repaired".

The $\iota_{\mathcal{ME}+\textsf{Sup}}$ inference system consists of the rules Deduce, Split and Close.

In the introduction we mentioned that the $\mathcal{ME}$+Sup calculus can be configured to obtain a pure Superposition or a pure Model Evolution calculus (with equality). For the former, every ground atom is to be labelled as a superposition atom. Then, the only inference rules in effect are Ref, Sup-Neg, Sup-Pos and Fact, all of which are standard inference rules of the Superposition calculus. Furthermore, under the reasonable assumption that the input clauses are constraint-free, all derivable contexts will be $\{\neg x\}$, and also the constraints in all derivable clauses will be empty. In consequence, not even Close is applicable (unless the clause set in the premise already contains $\Box \cdot \emptyset$). In contrast, if all atoms are labelled as split atoms, then the only inference rules in effect are Ref,U-Sup-Neg, U-Sup-Pos, Neg-U-Res, Split and Close. The resulting calculus is similar to the $\mathcal{ME}_{\text{E}}$ calculus [4] but not quite the same. On the one hand, $\mathcal{ME}_{\text{E}}$ features *universal*

[7] The Deduce rule and the Close rule could be strengthened to exclude adding variants to the clause sets in the conclusion. We ignore this (trivial) aspect.

variables, a practically important improvement, which $\mathcal{ME}$+Sup does not (yet) have. On the other hand, $\mathcal{ME}_{\mathrm{E}}$ needs to compute additional unifiers, for instance in the counterpart to the Close rule, which are not necessary in $\mathcal{ME}$+Sup.

7 Model Construction

To obtain the completeness result for $\mathcal{ME}$+Sup we associate to a sequent $\Lambda \vdash \Phi$ a convergent left-reduced rewrite system $R_{\Lambda \vdash \Phi}$. The general technique is taken from the completeness proof of the Superposition calculus [2,5] and adapted to our needs. One difference is that $\mathcal{ME}$+Sup requires the construction of a fully reduced rewrite system for its split atoms, whereas for the superposition atoms a left-reduced rewrite system is sufficient. Another difference is that certain aspects of lifting must be reflected already for the model generation. For the latter, we need a preliminary definition.

Definition 7.1 (Relevant Instance wrt. (Λ, R)). *Let Λ be a context, R a rewrite system, and γ a ground substitution for a constrained clause $C \cdot \Gamma$. We say that $(C \cdot \Gamma)\gamma$*[8] *is a* relevant instance (of $C \cdot \Gamma$) wrt. (Λ, R) *iff*

(i) $\Gamma\gamma$ consists of rewrite split literals,
(ii) Λ produces Γ and Λ produces $\Gamma\gamma$ by the same literals (see below), and
(iii) $(\mathcal{V}ar(C) \cap \mathcal{V}ar(\Gamma))\gamma$ is irreducible wrt. R.

In the previous definition, item (ii) is to be understood to say that, for each $L \in \Gamma$, there is a literal $K \in \Lambda$ that produces both L and $L\gamma$ in Λ.

Notice that in order for $C \cdot \Gamma$ to have relevant instances it is not necessary that $C \cdot \Gamma$ is taken from a specific clause set. Notice also that for a clause with an empty constraint all its instances are relevant.

Example 7.2. If $\Lambda = \{P(x), a \to b, \neg P(b)\}$, $R = \{a \to b\}$ and $C \cdot \Gamma = x \approx b \vee x \approx d \cdot P(x)$ then the substitution $\gamma = \{x \mapsto a\}$ gives a ground instance that satisfies condition (ii) but not (iii). With the substitution $\gamma = \{x \mapsto c\}$ both (ii) and (iii) are satisfied, and with $\gamma = \{x \mapsto b\}$ the condition (ii) is not satisfied but (iii) is. If $\Lambda = \{P(a)\}$ then $\Box \cdot P(x)$ does not have relevant instances (although Λ produces the ground constraint $P(a)$) because Λ does not produce $P(x)$. The calculus needs to make sure that such "irrelevant" constrained clauses need not be considered, as (in particular) Close cannot be applied to, say, $\{P(a)\} \vdash \Box \cdot P(x)$ although $\{P(a)\}, \emptyset \not\models \Box \cdot P(x)$. □

For a given sequent $\Lambda \vdash \Phi$, where Φ does not contain $\Box \cdot \emptyset$, we define by induction on the clause ordering $\succ$ sets of rewrite rules $\epsilon_{\mathcal{C}}$ and $R_{\mathcal{C}}$, for every $\mathcal{C} \in \Phi^{\mathrm{gr}} \cup \Pi_{\Lambda}$. Here, for the purpose of comparing (positive) rewrite literals, $l \to r$ is taken as the constrained clause $l \approx r \cdot \bot$, where $\bot$ is a fresh symbol that is considered

[8] Strictly speaking, the definition works with pairs $(C \cdot \Gamma, \gamma)$ instead of ground instances $(C \cdot \Gamma)\gamma$, but this causes no problems as γ will always be clear from the context. Similarly in other definitions below.

smaller than the empty multiset. This way, $\succ$ is a total ordering on $\Phi^{\mathrm{gr}} \cup \Pi_\Lambda$. For instance $(l \approx r \cdot \emptyset) \succ l \to r$ as $(l \approx r \cdot \emptyset) \succ (l \approx r \cdot \bot)$, as $\emptyset \succ \bot$.

Assume that $\epsilon_\mathcal{D}$ has already been defined for all $\mathcal{D} \in \Phi^{\mathrm{gr}} \cup \Pi_\Lambda$ with $\mathcal{C} \succ \mathcal{D}$ and let $R_\mathcal{C} = \bigcup_{\mathcal{C} \succ \mathcal{D}} \epsilon_\mathcal{D}$. The set $\epsilon_\mathcal{C}$ is defined differently depending on the type of $\mathcal{C}$. If $\mathcal{C}$ is rewrite literal $l \to r \in \Pi_\Lambda$ then let $\epsilon_{l \to r} = \{l \to r\}$ if

1. $l \approx r$ is a split atom,
2. $l \succ r$, and
3. l and r are irreducible wrt. $R_{l \to r}$.

Otherwise $\epsilon_{l \to r} = \emptyset$. If $\mathcal{C}$ is a constrained clause $C \cdot \Gamma \in \Phi^{\mathrm{gr}}$ then let $\epsilon_{C \cdot \Gamma} = \{s \to t\}$ if

1. $C = s \approx t \vee D$,
2. $s \approx t$ is strictly maximal in C,
3. $s \approx t$ is a superposition atom,
4. $s \succ t$,
5. $C \cdot \Gamma$ is a relevant instance of a constrained clause $C' \cdot \Gamma' \in \Phi$ wrt. $(\Lambda, R_{C \cdot \Gamma})$,
6. $R^\star_{C \cdot \Gamma} \not\models C$,
7. $(R_{C \cdot \Gamma} \cup \{s \to t\})^\star \not\models D$, and
8. s is irreducible wrt. $R_{C \cdot \Gamma}$.

Otherwise $\epsilon_{C \cdot \Gamma} = \emptyset$.

Finally, $R = \bigcup_\mathcal{C} \epsilon_\mathcal{C}$. If $\epsilon_{l \to r} = \{l \to r\}$ then we say that $l \to r$ *generates* $l \to r$ in R. If $\epsilon_{C \cdot \Gamma} = \{l \to r\}$ then we say that $C \cdot \Gamma$ *generates* $l \to r$ in R via $C' \cdot \Gamma'$. Often we write $R_{\Lambda \vdash \Phi}$ instead of R to make clear that R is constructed from $\Phi^{\mathrm{gr}} \cup \Pi_\Lambda$.

It is not difficult to show that R is a left-reduced rewrite system and the rules contributed by Π_Λ are even fully reduced wrt. R. Since $\succ$ is a well-founded ordering, R is a convergent rewrite system.

Notice that the evaluation of condition 5 for $\epsilon_{C \cdot \Gamma}$ refers to the context Λ, which is fixed prior to the model construction, and the rewrite system $R_{C \cdot \Gamma}$ constructed so far. The definition can be seen to work in a hierarchical way, by first building the set of those constrained clauses from Φ^{gr} whose constraints are produced in Λ, and then generating R from that set, which involves checking irreducibility of substitutions wrt. $R_{C \cdot \Gamma}$.

Example 7.3. Let $\Lambda = \{a \to x, b \to c, a \not\to c\}$, $\Phi = \emptyset$ and assume that all equations are split atoms. With $a \succ b \succ c$ the induced rewrite system R is $\{b \to c\}$. To see why, observe that the candidate rule $a \to c$ is not included in R, as Λ does not produce $a \to c$, and that the other candidate $a \to b$, although produced in Λ, is reducible by the smaller rule $b \to c$. Had we chosen to omit in the definition of $\epsilon_{C \cdot \Gamma}$ the condition "r is irreducible wrt. $R_{l \to r}$" [9] the construction would have given $R = \{a \to b, b \to c\}$. This leads to the undesirable situation that a constrained clause, say, $a \not\approx c \cdot \emptyset$ is falsified by $R^\star$. But the calculus cannot modify Λ to revert this situation, and to detect the inconsistency (ordered) paramodulation into variables would be needed.

[9] This condition is absent in the model construction for superposition atoms. Its presence explains why paramodulation into smaller sides of positive split literals in clauses is necessary.

Example 7.4. Let $a \succ b \succ c$, $\Lambda = \{P(x), \neg P(b), \neg P(c)\}$ and $C \cdot \Gamma = y \approx b \vee x \approx c \cdot P(x)$ be the only clause in Φ. Then the instance $a \approx b \vee a \approx c \cdot P(a)$ generates $a \to b$ in R. This is, because $a \approx b \vee a \approx c \cdot P(a)$ is relevant instance of $y \approx b \vee x \approx c \cdot P(x)$ wrt. $(\Lambda, R_{C\cdot\Gamma}) = (\Lambda, \emptyset)$. Let $\gamma = \{x \mapsto a, y \mapsto a\}$ be the corresponding ground substitution. Now, a (ground) inference with $(C \cdot \Gamma)\gamma$ as the left premise and a relevant instance of a clause as the right premise will possibly not preserve relevancy. This is, because the conclusion, say, $\mathcal{C}\gamma$, can be bigger than the left premise $(C \cdot \Gamma)\gamma$ (even if the right premise is bigger than the left premise, which is safe to assume) and this way $x\gamma$ could be reducible wrt. $R_{\mathcal{C}\gamma}$. For instance, if the right premise is $f(a) \not\approx f(b) \cdot \emptyset$ then a Sup-Neg inference yields $\mathcal{C} = f(b) \not\approx f(b) \vee x \approx c \cdot P(x)$. But $\mathcal{C}\gamma = f(b) \not\approx f(b) \vee a \approx c \cdot P(a)$ is not a relevant instance wrt. Λ, as $x\gamma = a$ is reducible wrt. $R_{\mathcal{C}\gamma} = \{a \to b\}$. This is a problem from the completeness perspective, because the calculus needs to reduce relevant instances of clauses that are false (in a certain interpretation) to smaller *relevant* instances. The suggested Sup-Neg step would thus not work in this case. The problem is avoided by a different Sup-Neg inference with a merge substitution:

$$\text{Sup-Neg}\ \frac{y \approx b \vee x \approx c \cdot P(x) \qquad f(a) \not\approx f(b) \cdot \emptyset}{f(b) \not\approx f(b) \vee a \approx c \cdot P(a)}$$

where $\sigma = \{y \mapsto a\}$ and $\pi = \{x \mapsto a\}$. Then, $f(b) \not\approx f(b) \vee a \approx c \cdot P(a)$ is a relevant instance (of itself) wrt. Λ. It can be shown that situations like the one above are the *only* critical ones and that relevancy can always be preserved by a merge substitution. □

8 Redundancy, Saturation and Static Completeness

To define concepts of redundancy we need a specific notion of relevant instances that takes the model construction into account. We extend Definition 7.1 and say that $(C\cdot\Gamma)\gamma$ is a *relevant instance of* $C\cdot\Gamma$ *wrt.* Λ iff $(C\cdot\Gamma)\gamma$ is a relevant instance of $C \cdot \Gamma$ wrt. $(\Lambda, R_{(C\cdot\Gamma)\gamma})$. Relevancy of an instance $(C \cdot \Gamma)\gamma$ wrt. Λ thus does not depend on rules from $R \setminus R_{(C\cdot\Gamma)\gamma}$. When Φ is a set of constrained clauses, let $\Phi^\Lambda = \{(C \cdot \Gamma)\gamma \mid C \cdot \Gamma \in \Phi$ and $(C \cdot \Gamma)\gamma$ is a relevant instance of $C \cdot \Gamma$ wrt. $\Lambda\}$. Let $\Lambda \vdash \Phi$ be a sequent and $\mathcal{D}$ a ground constrained clause. Define $\Phi^\Lambda_\mathcal{D} = \{C\cdot\Gamma \in \Phi^\Lambda \mid \mathcal{D} \succ C \cdot \Gamma\}$ as the set of relevant instances wrt. Λ of all constrained clauses from Φ that are all smaller wrt. $\succ$ than $\mathcal{D}$.

We say that a ground constrained clause $C \cdot \Gamma$ is *redundant wrt.* $\Lambda \vdash \Phi$ *and* $\mathcal{D}$ iff $\Phi^\Lambda_\mathcal{D} \models_\Lambda C \cdot \Gamma$, that is, iff $C \cdot \Gamma$ is entailed wrt. Λ by relevant instances wrt. Λ of clauses in Φ that are smaller than $\mathcal{D}$. We say that $C \cdot \Gamma$ is *redundant wrt.* $\Lambda \vdash \Phi$ iff $C \cdot \Gamma$ is redundant wrt. $\Lambda \vdash \Phi$ and $C \cdot \Gamma$.

The previous definitions are essential to prove completeness but difficult to directly exploit in practice. The following, related definition is more practical, as it refers to a context Λ only by checking if *ground* rewrite literals are contained, a property that is preserved as Λ grows.

For a context Λ let $\mathrm{grd}(\Lambda)$ denote the set of all ground literals in Λ.

Definition 8.1 (Universal Redundancy). *Let $\Lambda \vdash \Phi$ be a sequent, $\mathcal{D}$ a ground constrained clause, and γ a ground substitution for a constrained clause $C \cdot \Gamma$. We say that $(C \cdot \Gamma)\gamma$ is* universally redundant wrt. *$\Lambda \vdash \Phi$ and $\mathcal{D}$, iff there exists an $L \in \Gamma$ such that $\overline{L}\gamma \in \mathrm{grd}(\Lambda)$, or there exist ground instances $(C_i \cdot \Gamma_i)\gamma_i$ of constrained clauses $C_i \cdot \Gamma_i \in \Phi$ such that (i) if $L \in \Gamma_i$, then $L \in \mathrm{grd}(\Lambda)$ or there exists a $K \in \Gamma$ such that $L \sim K$ and $L\gamma_i = K\gamma$, (ii) $\mathcal{D} \succ (C_i \cdot \Gamma_i)\gamma_i$ for every i, (iii) $C_1\gamma_1 \ldots C_n\gamma_n \models C\gamma$, and (iv) if $x \in \mathcal{V}ar(C_i) \cap \mathcal{V}ar(\Gamma_i)$, then there exists a $y \in \mathcal{V}ar(C) \cap \mathcal{V}ar(\Gamma)$ such that $x\gamma_i = y\gamma$.*

We say that $(C \cdot \Gamma)\gamma$ is *universally redundant wrt. $\Lambda \vdash \Phi$*, iff $(C \cdot \Gamma)\gamma$ is universally redundant wrt. $\Lambda \vdash \Phi$ and $(C \cdot \Gamma)\gamma$, and we say that $C \cdot \Gamma$ is *universally redundant wrt. $\Lambda \vdash \Phi$* iff $(C \cdot \Gamma)\gamma$ is universally redundant wrt. $\Lambda \vdash \Phi$, for every ground substitution γ for $C \cdot \Gamma$.

For instance, when A is a ground literal, any (possibly non-ground) clause of the form $C \cdot A, \Gamma$ is universally redundant wrt. every $\Lambda \vdash \Phi$ such that $\overline{A} \in \Lambda$. Dually, $C \cdot A, \Gamma$ is universally redundant wrt. every $\Lambda \vdash \Phi$ such that $A \in \Lambda$ and $C \cdot \Gamma \in \Phi$. Correspondingly, the simplification rule defined below can be used to delete $C \cdot A, \Gamma$ if $\overline{A} \in \Lambda$, and if $A \in \Lambda$ then $C \cdot A, \Gamma$ can be simplified to $C \cdot \Gamma$. This generalizes corresponding simplification rules by unit clauses in the propositional DPLL-procedure.

Also, a constrained clause $C \cdot \Gamma'$ is universally redundant wrt. any sequent containing a constrained clause $C \cdot \Gamma$ such that $\Gamma \subset \Gamma'$. This can be exploited to finitely bound the number of derivable constrained clauses under certain conditions. For instance, if the clause parts cannot grow in length, e.g., by disabling superposition by labelling all atoms as split atoms, and if the term depth is limited, too, e.g., for Bernays-Schönfinkel formulas, then $\mathcal{ME}$+Sup derivations can be finitely bounded, too.

Proposition 8.2. *If $C \cdot \Gamma$ is universally redundant wrt. $\Lambda \vdash \Phi$, then every relevant instance of $C \cdot \Gamma$ wrt. Λ is redundant wrt. $\Lambda \vdash \Phi$.*

Proposition 8.2 explains the relationship between the two concepts of redundancy above. Because the completeness proof needs to consider relevant, non-redundant (ground) instances only, Proposition 8.2 then justifies that the calculus need not work with universally redundant clauses. More specifically, referring to the notion of derivation trees formally defined in Section 9 below, it can be shown that a clause that is universally redundant at some node of the derivation tree will remain universally redundant in all successor nodes, that all its relevant ground instances are redundant (and therefore cannot be minimal counterexamples in the model construction), and that its ground instances cannot generate rewrite rules. Consequently, a universally redundant clause can be deleted from a clause set without endangering refutational completeness. We emphasize that for clauses with empty constraints, universal redundancy coincides with the classical notion of redundancy for the Superposition calculus.

Definition 8.3 (Universally Redundant $\iota_{\mathcal{ME}+\mathsf{Sup}}$ Inference). *Let $\Lambda \vdash \Phi$ and $\Lambda' \vdash \Phi'$ be sequents. A $\iota_{\mathcal{ME}+\mathsf{Sup}}$ inference with premise $\Lambda \vdash \Phi$ and selected*

clause $C \cdot \Gamma \in \Phi$ *is* universally redundant wrt. $\Lambda' \vdash \Phi'$ *iff for every ground substitution* γ, $(C \cdot \Gamma)\gamma$ *is universally redundant wrt.* $\Lambda' \vdash \Phi'$, *or the following holds, depending on the inference rule applied:*

Deduce: *One of the following holds:*

(i) Applying γ *to all premises and the conclusion* $C' \cdot \Gamma'$ *of the underlying* $\iota_{\mathcal{ME}+\mathsf{Sup}}$ *inference does not result in a ground instance via* γ *of this* $\iota_{\mathcal{ME}+\mathsf{Sup}}$ *inference.*

(ii) $(C' \cdot \Gamma')\gamma$ *is universally redundant wrt.* $\Lambda' \vdash \Phi'$ *and* $(C \cdot \Gamma)\gamma$.

(iii) In case of Sup-Neg *or* Sup-Pos, *where* $C'' \cdot \Gamma''$ *is the left premise,* $(C'' \cdot \Gamma'')\gamma$ *is universally redundant wrt.* $\Lambda' \vdash \Phi'$.

Split: $C \cdot \Gamma = \Box \cdot \Gamma$ *and* Λ' *does not produce* Γ.

Close: $C \cdot \Gamma = \Box \cdot \emptyset \in \Phi'$.

It is not difficult to show that actually carrying out an inference renders it universally redundant in the resulting sequent. With a view to implementation, this indicates that effective proof procedures for $\mathcal{ME}$+Sup indeed exist.

Finally, a sequent $\Lambda \vdash \Phi$ is *saturated* iff every $\iota_{\mathcal{ME}+\mathsf{Sup}}$ inference with premise $\Lambda \vdash \Phi$ is universally redundant wrt. $\Lambda \vdash \Phi$.

Theorem 8.4 (Static Completeness). *If* $\Lambda \vdash \Phi$ *is a saturated sequent with a non-contradictory context* Λ *and* $\Box \cdot \emptyset \notin \Phi$ *then the induced rewrite system* $R_{\Lambda \vdash \Phi}$ *satisfies all relevant instances of all clauses in* Φ *wrt.* Λ *, i.e.,* $\Lambda, R_{\Lambda \vdash \Phi} \models \Phi^{\Lambda}$. *Moreover, if* Ψ *is a clause set and* Φ *includes* Ψ, *i.e.,* $\{D \cdot \emptyset \mid D \in \Psi\} \subseteq \Phi$, *then* $R^{\star}_{\Lambda \vdash \Phi} \models \Psi$.

The stronger statement $\Lambda, R_{\Lambda \vdash \Phi} \models \Phi$ does in general not follow, as $(\Lambda, R_{\Lambda \vdash \Phi})$ possibly falsifies a *non-relevant* ground instance of a constrained clause in Φ. An example is the sequent $\Lambda \vdash \Phi = P(f(x)),\ f(a) \to a \vdash \Box \cdot P(f(x)), f(x) \to x$. Observe that Close is not applicable. Further, Λ does not produce the constraint $\{P(f(x)), f(x) \to x\}$ and hence the Split application with selected clause $\Box \cdot P(f(x)), f(x) \to x$ is universally redundant wrt. $\Lambda \vdash \Phi$. Altogether, $\Lambda \vdash \Phi$ is saturated. However, $\Lambda, R_{\Lambda \vdash \Phi} \not\models \Box \cdot P(f(a)), f(a) \to a$ as $\Lambda \models \{P(f(a)), f(a) \to a\}$ and no rewrite system satisfies $\Box$. Hence $\Lambda, R_{\Lambda \vdash \Phi} \not\models \Box \cdot P(f(x)), f(x) \to x$. But this does not violate Theorem 8.4, as $\Box \cdot P(f(a)), f(a) \to a$ is not a *relevant* instance of $\Box \cdot P(f(x)), f(x) \to x$. Although $x\{x \mapsto a\}$ is irreducible wrt. $R_{\Box \cdot P(f(a)), f(a) \to a} = \emptyset$, Λ does not produce $f(x) \to x$, and hence does not produce $\{P(f(x)), f(x) \to x\}$ and $\{P(f(a)), f(a) \to a\}$ by the same literals.

9 Derivations with Simplification

To make derivation in $\mathcal{ME}$+Sup practical the universal redundancy criteria defined above should be made available not only to avoid inferences, but also to, e.g., delete universally redundant clauses that come up in derivations. The following generic simplification rule covers many practical cases.

$$\mathsf{Simp}\ \frac{\Lambda \vdash \Phi, C \cdot \Gamma}{\Lambda \vdash \Phi, C' \cdot \Gamma'}$$

if (i) $C \cdot \Gamma$ is universally redundant wrt. $\Lambda \vdash \Phi, C' \cdot \Gamma'$, and (ii) $(\Lambda^{\mathrm{c}})^a \cup (\Phi \cup \{C \cdot \Gamma\})^{\mathrm{c}} \models (C' \cdot \Gamma')^{\mathrm{c}}$.

The Simp rule generalizes the widely-used simplification rules of the Superposition calculus, such as deletion of trivial equations $t \not\approx t$ from clauses, demodulation with unit clauses and (non-proper) subsumption; these rules immediately carry over to $\mathcal{ME}$+Sup as long as all involved clauses have empty constraints. Also, as said above, the usual unit propagation rules of the (propositional) DPLL procedure are covered in a more general form. As $\mathcal{ME}$+Sup is intended as a generalization of propositional DPLL (among others), it is mandatory to provide this feature.

Condition (ii) is needed for soundness. The $\cdot^a$-operator uniformly replaces each variable in each (unit) clause by a constant a. This way, all splits are effectively over complementary propositional literals.

Derivations. The purpose of the $\mathcal{ME}$+Sup calculus is to build for a given clause set a derivation tree over sequents all of whose branches end in a closed sequent iff the clause set is unsatisfiable. Formally, we consider ordered trees $\mathbf{T} = (\mathbf{N}, \mathbf{E})$ where $\mathbf{N}$ and $\mathbf{E}$ are the sets of nodes and edges of $\mathbf{T}$, respectively, and the nodes N are labelled with sequents. Often we will identify a node's label with the node itself.

Derivation trees $\mathbf{T}$ *(of a set* $\{C_1, \ldots, C_n\}$ *of clauses)* are defined inductively as follows: an *initial tree* is a derivation tree, i.e., a tree $\mathbf{T}$ with a root node only that is labeled with the sequent $\neg x \vdash C_1 \cdot \emptyset, \ldots, C_n \cdot \emptyset$; if $\mathbf{T}$ is a derivation tree, N is a leaf node of $\mathbf{T}$ and $\mathbf{T}'$ is a tree obtained from $\mathbf{T}$ by adding one or two child nodes to N so that N is the premise of an $\iota_{\mathcal{ME}+\mathsf{Sup}}$ inference, a Simp inference or a Cancel inference, and the child node(s) is (are) its conclusion(s), then $\mathbf{T}'$ is derivation tree. In this case we say that $\mathbf{T}'$ is *derived* from $\mathbf{T}$. A *derivation (of* $\{C_1, \ldots, C_n\}$*)* is a possibly infinite sequence of derivation trees that starts with an initial tree and all subsequent derivation trees are derived from their immediate predecessor. Each derivation $\mathcal{D} = ((\mathbf{N}_i, \mathbf{E}_i))_{i<\kappa}$, where $\kappa \in \mathbb{N} \cup \{\omega\}$, determines a *limit tree* $(\bigcup_{i<\kappa} \mathbf{N}_i, \bigcup_{i<\kappa} \mathbf{E}_i)$. It is easy to show that a limit tree of a derivation $\mathcal{D}$ is indeed a tree. But note that it will not be a derivation tree unless $\mathcal{D}$ is finite.

Now let $\mathbf{T}$ be the limit tree of some derivation, let $\mathbf{B} = (N_i)_{i<\kappa}$ be a branch in $\mathbf{T}$ with κ nodes, and let $\Lambda_i \vdash \Phi_i$ be the sequent labeling node N_i, for all $i < \kappa$. Define $\Lambda_{\mathbf{B}} = \bigcup_{i<\kappa} \bigcap_{i\le j<\kappa} \Lambda_j$[10] and $\Phi_{\mathbf{B}} = \bigcup_{i<\kappa} \bigcap_{i\le j<\kappa} \Phi_j$, the sets of *persistent context literals* and *persistent clauses*, respectively. These two sets can be combined to obtain the *limit sequent* $\Lambda_{\mathbf{B}} \vdash \Phi_{\mathbf{B}}$ (of $\mathbf{T}$).

As usual, the completeness of $\mathcal{ME}$+Sup relies on a suitable notion of fairness, which is defined in terms of exhausted branches. When we say that "X is not persistent" we mean that X is not among the persistent context literals or X is not among the persistent clauses, depending on whether X is a rewrite literal or a constrained clause.

[10] The definition of $\Lambda_{\mathbf{B}}$ is slightly more general as needed. Currently, there are no inference rules to delete context elements, and so $\Lambda_{\mathbf{B}}$ is always $\bigcup_{i<\kappa} \Lambda_i$.

Definition 9.1 (Exhausted Branch). *Let* $\mathbf{T}$ *be a limit tree and* $\mathbf{B} = (N_i)_{i<\kappa}$ *a branch in* $\mathbf{T}$ *with* κ *nodes. For all* $i < \kappa$*, let* $\Lambda_i \vdash \Phi_i$ *be the sequent labeling node* N_i*. The branch* $\mathbf{B}$ *is* exhausted *iff*

(i) for all $i < \kappa$*, every* $\iota_{\mathcal{ME}+\mathsf{Sup}}$ *inference with premise* $\Lambda_i \vdash \Phi_i$ *and a persistent selected clause and a persistent left premise (in case of* Deduce*) is universally redundant wrt.* $\Lambda_j \vdash \Phi_j$*, for some* $j < \kappa$ *with* $j \geq i$*, and*

(ii) $\square \cdot \emptyset \notin \Phi_{\mathbf{B}}$

A limit tree of a derivation is *fair* iff it is a refutation tree that is, a finite tree all of whose leafs contain $\square \cdot \emptyset$ in the constrained clause part of their sequent, or it has an exhausted branch. A derivation is *fair* iff its limit tree is fair.

Notice that if in condition (i) in the above definition the selected clause or the left premise (in case of Deduce) is universally redundant wrt. $\Lambda_i \vdash \Phi_i$, then the $\iota_{\mathcal{ME}+\mathsf{Sup}}$ inference is already redundant wrt. $\Lambda_i \vdash \Phi_i$. In other words, inferences with a universally redundant premise need not be carried out.

Proposition 9.2 (Exhausted Branches are Saturated). *If* $\mathbf{B}$ *is an exhausted branch of a limit tree of a fair derivation then* $\Lambda_{\mathbf{B}} \vdash \Phi_{\mathbf{B}}$ *is saturated.*

Proposition 9.2 is instrumental in the proof of our main result, which is the following.

Theorem 9.3 (Completeness). *Let* Ψ *be a clause set and* $\mathbf{T}$ *be the limit tree of a fair derivation of* Ψ*. If* $\mathbf{T}$ *is not a refutation tree then* Ψ *is satisfiable; more specifically, for every exhausted branch* $\mathbf{B}$ *of* $\mathbf{T}$ *with limit sequent* $\Lambda_{\mathbf{B}} \vdash \Phi_{\mathbf{B}}$ *and induced rewrite system* $R_{\mathbf{B}} = R_{\Lambda_{\mathbf{B}} \vdash \Phi_{\mathbf{B}}}$ *it holds* $\Lambda_{\mathbf{B}}, R_{\mathbf{B}} \models (\Phi_{\mathbf{B}})^{\Lambda_{\mathbf{B}}}$ *and* $R^{\star}_{\mathbf{B}} \models \Psi$*.*

The $\mathcal{ME}$+Sup calculus is also sound. The idea behind the soundness proof is to conceptually replace every variable in every literal in all contexts by a constant, say, a. This results in a refutation tree where all splits are over complementary propositional literals. Regarding Close inferences, any closing clause will still be closing after instantiating all its variables in the same way. Furthermore, observe that the Ref, Sup-Neg, Sup-Pos and Fact inference rules are sound in the standard sense by taking the clausal forms of the premises and the conclusions. For the remaining ι_{Base} inference rules U-Sup-Pos, U-Sup-Neg and Neg-U-Res this is even simpler as the constraint in the conclusion contains the left premise (they are strongly sound). The soundness of Simp follows from its condition (ii). This way, a set of ground instances can be identified that demonstrates the unsatisfiability of the input clause set whenever a refutation exists. A formal completeness proof can be carried out as for the $\mathcal{ME}_{\text{E}}$ calculus [4].

10 Conclusions

Our main result is the completeness of the new $\mathcal{ME}$+Sup calculus. On the theoretical side, we plan to investigate how it can be exploited to obtain decision procedures for fragments of first-order logic that are beyond the scope of current superposition or instance-based methods. Ultimately, we will need an implementation to see how the labelling function is best exploited in practice for general refutational theorem proving.

Acknowledgements. We thank the reviewers and Cesare Tinelli for their helpful comments.

References

1. Baader, F., Nipkow, T.: Term Rewriting and all that. Cambridge University Press, Cambridge (1998)
2. Bachmair, L., Ganzinger, H.: Chapter 11: Equational Reasoning in Saturation-Based Theorem Proving. In: Bibel, W., Schmitt, P.H. (eds.) Automated Deduction. A Basis for Applications. Foundations. Calculi and Refinements, vol. I, pp. 353–398. Kluwer, Dordrecht (1998)
3. Baumgartner, P., Tinelli, C.: The Model Evolution Calculus. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 350–364. Springer, Heidelberg (2003)
4. Baumgartner, P., Tinelli, C.: The model evolution calculus with equality. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 392–408. Springer, Heidelberg (2005)
5. Nieuwenhuis, R., Rubio, A.: Theorem Proving with Ordering and Equality Constrained Clauses. Journal of Symbolic Computation 19, 321–351 (1995)
6. Weidenbach, C., Schmidt, R., Hillenbrand, T., Rusev, R., Topic, D.: System description: Spass version 3.0. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 514–520. Springer, Heidelberg (2007)

On Deciding Satisfiability by DPLL($\Gamma + \mathcal{T}$) and Unsound Theorem Proving

Maria Paola Bonacina[1,⋆], Christopher Lynch[2], and Leonardo de Moura[3]

[1] Dipartimento di Informatica, Università degli Studi di Verona
Strada Le Grazie 15, I-37134 Verona, Italy
mariapaola.bonacina@univr.it
[2] Department of Mathematics and Computer Science
Clarkson University, Potsdam, NY 13699-5815, U.S.A.
clynch@clarkson.edu
[3] Microsoft Research, One Microsoft Way, Redmond, WA 98052, U.S.A.
leonardo@microsoft.com

Abstract. Applications in software verification often require determining the satisfiability of first-order formulæ with respect to some background theories. During development, conjectures are usually false. Therefore, it is desirable to have a theorem prover that terminates on satisfiable instances. Satisfiability Modulo Theories (SMT) solvers have proven highly scalable, efficient and suitable for integrated theory reasoning. Superposition-based inference systems are strong at reasoning with equalities, universally quantified variables, and Horn clauses. We describe a calculus that tightly integrates Superposition and SMT solvers. The combination is refutationally complete if background theory symbols only occur in ground formulæ, and non-ground clauses are variable inactive. Termination is enforced by introducing additional axioms as hypotheses. The calculus detects any unsoundness introduced by these axioms and recovers from it.

1 Introduction

Applications in software verification have benefited greatly from recent advances in automated reasoning. Applications in this field often require determining the satisfiability of first-order formulæ with respect to some background theories. In numerous contexts in software verification, quantifiers are needed. For example, they are used for capturing frame conditions over loops, axiomatizing type systems, summarizing auxiliary invariants over heaps, and for supplying axioms of theories that are not already equipped with decision procedures for ground formulæ. Thus, many verification problems consist in determining the satisfiability of a set of formulæ $\mathcal{R} \uplus P$ modulo a background theory $\mathcal{T}$, where $\mathcal{R}$ is a set of non-ground clauses without occurrences of $\mathcal{T}$-symbols, and P is a large ground formula (or set of ground clauses) that may contain $\mathcal{T}$-symbols. The set of formulæ $\mathcal{R}$ can be viewed as the axiomatization of an application specific theory. The background

⋆ Research supported in part by MIUR grant no. 2007-9E5KM8.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 35–50, 2009.

theory $\mathcal{T}$ is a combination of general-purpose theories commonly used in hardware and software verification, such as linear arithmetic and bit-vectors.

Satisfiability Modulo Theories (SMT) solvers have proven highly scalable, efficient and suitable for integrated theory reasoning. Most SMT solvers are restricted to ground formulæ, and integrate the Davis-Putnam-Logemann-Loveland procedure (DPLL) with satellite $\mathcal{T}_i$-solvers for ground satisfiability problems in special theories $\mathcal{T}_i$, $1 \leq i \leq n$, so that $\mathcal{T} = \bigcup_{i=1}^{n} \mathcal{T}_i$. In comparison, superposition-based inference systems (SP) are strong at reasoning with equalities, universally quantified variables, and Horn clauses. Moreover, SP was proved to terminate and hence to be a satisfiability procedure for several theories of data structures [1,2,5].

The DPLL($\Gamma + \mathcal{T}$) calculus [11] integrates an SMT solver with an inference system Γ that is sound and refutationally complete for first-order logic with equality. The key to the integration is that the literals in the candidate model built by the DPLL engine can occur as premises of Γ-inferences. In general, the DPLL($\Gamma + \mathcal{T}$) calculus is not refutationally complete when $\mathcal{T}$ is not empty, even when $\mathcal{T}$-symbols do not occur in $\mathcal{R}$. For example, assume $\mathcal{R} = \{x = a \vee x = b\}$ and $P = \emptyset$, and the background theory $\mathcal{T}$ is arithmetic. The clause $\{x = a \vee x = b\}$ implies that any model has at most two elements, which is clearly incompatible with any model for arithmetic. A first contribution of this paper are the conditions under which DPLL($\Gamma + \mathcal{T}$) is refutationally complete when $\mathcal{T}$ is not empty.

DPLL($\Gamma + \mathcal{T}$) has to combine all the theories in $\mathcal{T} = \bigcup_{i=1}^{n} \mathcal{T}_i$ and $\mathcal{R}$. Combination of theories in SMT solvers is usually done by the Nelson-Oppen scheme [20], which requires that each $\mathcal{T}_i$ be *stably infinite*[1] and its solvers capable of generating all entailed disjunctions of equalities between constants. The second requirement could be relaxed as in [12], if each $\mathcal{T}_i$-solver were able to generate a candidate model, which may not be the case in general for all $\mathcal{T}_i$-solvers and for Γ acting as $\mathcal{R}$-solver. A second contribution of this work is to explain how to apply known results on *variable inactivity* [1,8] to combine the built-in theories $\mathcal{T}_1, \ldots, \mathcal{T}_n$ and the axiomatized theory $\mathcal{R}$ in DPLL($\Gamma + \mathcal{T}$).

In software verification, during development time, several conjectures are false because of mistakes in the implementation or specification. Therefore, it is desirable to have a theorem prover that terminates on satisfiable instances. In general, this is not a realistic goal since pure first-order logic is not decidable, and, even worse, there is no sound and complete procedure for first-order logic formulæ of linear arithmetic with uninterpreted functions [15]. Axioms such as *transitivity* ($\neg(x \sqsubseteq y) \vee \neg(y \sqsubseteq z) \vee x \sqsubseteq z$) and *monotonicity* ($\neg(x \sqsubseteq y) \vee f(x) \sqsubseteq f(y)$) are problematic for any resolution-based Γ, since they tend to generate an unbounded number of clauses, even with a selection function that selects negative literals to prevent self-resolutions. Such axioms may arise in formalizations of type systems for programming languages. The signature features a predicate $\sqsubseteq$ that represents a subtype relationship, and a monadic function f that represents a type constructor, such as `Array-of`. As an example, assume that the axiomatization contains a

[1] Every $\mathcal{T}_i$-satisfiable ground formula has a model with domain of infinite cardinality.

monotonicity axiom $\neg(x \sqsubseteq y) \vee f(x) \sqsubseteq f(y)$. Resolution with negative selection would generate an infinite sequence $\{f^i(a) \sqsubseteq f^i(b)\}_{i\geq 0}$ for each literal $a \sqsubseteq b$ in its input. In practice, it is seldom the case that we need to go beyond $f(a) \sqsubseteq f(b)$ or $f^2(a) \sqsubseteq f^2(b)$ to show satisfiability. A third and main contribution of this paper is a new calculus that combines DPLL($\Gamma + \mathcal{T}$) with *unsound theorem proving* [17] to avoid such infinitary behaviors and obtain decision procedures for axiomatizations relevant to software verification. The idea is to control the infinitary behavior by using additional hypotheses/axioms, detect any unsoundness they may introduce and recover from it.

2 Background

We employ basic notions from logic usually assumed in theorem proving. Let Σ be a *signature* consisting of a set of *function* and *predicate* symbols, each with its *arity*, denoted by *arity*(f), for symbol f. We call 0-arity function symbols *constant* symbols, and use a, b, c and d for constants, f, g, h for non-constant function symbols, and x, y, z for variables. We use $\simeq$ to denote the interpreted predicate symbol for equality and $Var(l)$ to denote the set of variables occurring in a term or literal l. A first order Σ-theory is presented, or axiomatized, by a set of Σ-sentences. We use the symbols $\mathcal{T}$ and $\mathcal{R}$ to denote such presentations. Interpreted symbols are those symbols whose interpretation is restricted to the models of a certain theory, whereas free or uninterpreted symbols are those symbols whose interpretation is unrestricted.

A Σ-structure $\mathcal{\Phi}$ consists of a non-empty universe $|\mathcal{\Phi}|$ and an interpretation for variables and symbols in Σ. For each symbol f in Σ, the interpretation of f is denoted by $\mathcal{\Phi}(f)$. For a function symbol f with *arity*$(f) = n$, the interpretation $\mathcal{\Phi}(f)$ is an n-ary function on $|\mathcal{\Phi}|$ with $\mathsf{range}(\mathcal{\Phi}(f)) = \{u \mid \exists v \in |\mathcal{\Phi}|,\ \mathcal{\Phi}(f)(v) = u\}$. For a predicate symbol p with *arity*$(p) = n$, $\mathcal{\Phi}(p)$ is a subset of $|\mathcal{\Phi}|^n$. The interpretation of a term t is denoted by $\mathcal{\Phi}(t)$. If t is a variable or constant, $\mathcal{\Phi}(t)$ is an element in $|\mathcal{\Phi}|$. Otherwise, $\mathcal{\Phi}(f(t_1, \ldots, t_n)) = \mathcal{\Phi}(f)(\mathcal{\Phi}(t_1), \ldots, \mathcal{\Phi}(t_n))$. If S is a set of terms, $\mathcal{\Phi}(S)$ means the set $\{\mathcal{\Phi}(t) \mid t \in S\}$. Satisfaction $\mathcal{\Phi} \models C$ is defined as usual, and if $\mathcal{\Phi} \models C$, the structure $\mathcal{\Phi}$ is said to be a model of C.

An *inference system* Γ is a set of inference rules. We consider an *ordering-based* inference system, that assumes an ordering $\succ$ on terms and literals, and uses it to restrict expansion inferences and define contraction inferences. This ordering is a *complete simplification ordering* (stable, monotone, with the subterm property, hence well-founded, and total on ground terms and literals). An *inference rule* γ with n premises is an $n+1$-ary relation on clauses. Each inference rule has a *main premise* that yields the conclusion in the context of the other (*side*) premises. For contraction rules, the main premise is reduced to the conclusion. Let I be a mapping, called a *model functor*, that assigns to each set of ground clauses N not containing $\Box$ an interpretation I_N, called the *candidate model*. An inference system Γ has the *reduction property for counterexamples*, if for all sets N of clauses and minimal counterexamples C for I_N in N, there is an inference in Γ from N with main premise C, side premises that are true in I_N, and conclusion D that is a smaller counterexample for I_N than C.

3 Variable Inactivity in DPLL($\Gamma + \mathcal{T}$)

In this section we will see how previous results from the rewrite-based approach to satisfiability procedures [1,8] can be imported into the DPLL($\Gamma + \mathcal{T}$) framework to combine a built-in theory $\mathcal{T}$ and an axiomatized theory $\mathcal{R}$. In a purely rewrite-based approach there is no built-in theory and all axioms are part of the input in $\mathcal{R}$. The core of the methodology is to show that a first-order engine, such as SP, is an $\mathcal{R}$-satisfiability procedure, by showing that it is guaranteed to terminate on $\mathcal{R}$-satisfiability problems $\mathcal{R} \uplus S$, where S is a set of ground unit $\mathcal{R}$-clauses. Termination is *modular*: if SP terminates on $\mathcal{R}_i$-satisfiability problems, for $1 \leq i \leq n$, it terminates also on $\mathcal{R}$-satisfiability problems for $\mathcal{R} = \bigcup_{i=1}^{n} \mathcal{R}_i$, provided the signatures of the $\mathcal{R}_i$'s do not share function symbols, and all the $\mathcal{R}_i$'s are *variable inactive* [1]:

Definition 1. *A clause C is* variable-inactive *if no maximal literal in C is an equation $t \simeq x$ where $x \notin Var(t)$. A set of clauses is* variable-inactive *if all its clauses are.*

Maximality is relative to the ordering $\succ$ of Γ, which is required to be *good*, meaning that $t \succ c$ for all ground compound term t and constant c [1,6].

Definition 2. *A theory presentation $\mathcal{R}$ is* variable-inactive *for an inference system Γ if the limit S_∞ of a fair Γ-derivation from $S_0 = \mathcal{R} \uplus S$ is variable-inactive, where S is a set of ground unit $\mathcal{R}$-clauses.*

It was proved in [1] (cf. Thm. 4.5) that if $\mathcal{R}$ is variable-inactive, then it is stably-infinite. This observation is a corollary of a result of [8] (cf. Lemma 5.2) that says that if S_0 is satisfiable, then S_0 admits no infinite models if and only if the limit S_∞ of a fair SP-derivation from S_0 contains a *cardinality constraint*, that is, a clause containing only non-trivial (i.e., other than $x \simeq x$) positive equations between variables (e.g., $y \simeq x \vee y \simeq z$). Such a clause is clearly not variable-inactive. SP will reveal the lack of stable infiniteness by generating a cardinality constraint.[2] Thus, variable-inactivity is a sufficient condition for modularity of termination, hence to combine theories in the rewrite-based approach, and for stable-infiniteness, hence to mix combination of axiomatized theories as in the rewrite-based approach with combination of built-in theories à la Nelson-Oppen, as investigated also in [7] in a different setting.

In DPLL($\Gamma+\mathcal{T}$) applied to a problem $\mathcal{R} \uplus P$ modulo $\mathcal{T}$, Γ deals only with non-ground clauses and ground unit clauses, so that Γ works on an $\mathcal{R}$-satisfiability problem $\mathcal{R} \uplus S$, where S is a set of ground unit clauses. Thus, it makes sense to apply the results from the rewrite-based approach to Γ seen as an $\mathcal{R}$-solver. DPLL($\Gamma+\mathcal{T}$) needs to combine $\mathcal{T}_1, \ldots, \mathcal{T}_n, \mathcal{R}$ in the Nelson-Oppen scheme, which requires that the theories do not share function symbols, are stably infinite and

[2] Lemma 5.2 in [8] requires that the superposition-based inference system is invariant with respect to renaming finitely many constants. Most inference systems satisfy a stronger requirement, namely they allow signature extensions, e.g., to introduce Skolem constants.

each solver generates all entailed (disjunctions of) equalities between constants. We assume that $\mathcal{T}_1, \ldots, \mathcal{T}_n$ satisfy these requirements and that $\mathcal{R}$ does not share function symbols with them. For stable infiniteness of $\mathcal{R}$, we apply the above result about variable inactivity implying stable infiniteness: in the new version of Z3(SP), the SP engine is equipped with a test that detects the generation of variable-inactive clauses, hence cardinality constraints, and discovers whether $\mathcal{R}$ is not stably infinite. Such a test also excludes upfront a situation such as $\mathcal{R} = \{x = a \vee x = b\}$ of the example in Section 1. For the generation of (disjunctions of) equalities between constants in $\mathcal{R}$, we assume that the Γ engine is *fair*, which ensures that every theorem is implied by some generated formulae.[3] If contraction is also done systematically, only irredundant clauses generated by Γ are kept and passed to the DPLL($\mathcal{T}$) core.

The aforementioned results on variable inactivity were proved under the hypotheses that the ground unit clauses in S are $\mathcal{R}$-clauses and equality is the only predicate symbol. In the framework of DPLL($\Gamma + \mathcal{T}$), ground clauses may contain also $\mathcal{T}$-symbols, and $\mathcal{R}$ may introduce predicate symbols other than equality. We handle the first issue by *purification*, a standard step in the Nelson-Oppen method, which separate occurrences of function symbols from different signatures, by introducing new constant symbols (e.g., $f(g(a)) \simeq b$, where f and g belongs to different signatures, becomes $f(c) \simeq b \wedge g(a) \simeq c$, where c is new). The initial set of ground clauses P is transformed in two disjoint sets P_1 and P_2, where P_1 contains only $\mathcal{R}$-symbols and P_2 only $\mathcal{T}$-symbols. Since only constants are introduced, the problem remains ground. We deal with the second issue by representing an $\mathcal{R}$-atom $p(t_1, \ldots, t_n)$ as $f_p(t_1, \ldots, t_n) = \top$, where f_p is a new function symbols and $\top$ is a special constant.

Definition 3. *A set of formulæ $\mathcal{R} \uplus P$ is* smooth *with respect to a background theory $\mathcal{T} = \bigcup_{i=1}^{n} \mathcal{T}_i$, if the signatures of $\mathcal{T}_1, \ldots, \mathcal{T}_n, \mathcal{R}$ do not share function symbols, $\mathcal{R}$ is variable inactive, and P is a set of ground formulæ $P_1 \uplus P_2$, where P_1 contains only $\mathcal{R}$-symbols, and P_2 only $\mathcal{T}$-symbols.*

This definition summarizes the problem requirements for the sequel.

4 Unsound Theorem Proving in DPLL($\Gamma + \mathcal{T}$)

In theorem proving applied to mathematics, most conjectures are true. Thus, it is customary to sacrifice completeness for efficiency, and retain soundness, which is necessary to attribute unsatisfiability to a set of clauses F if a proof is found. A traditional example is *deletion by weight* [19], where clauses that are too "heavy" are deleted. In theorem proving applied to verification, most conjectures are false. Thus, it was suggested in [17] to sacrifice soundness for termination, and retain completeness, which is necessary to establish satisfiability if a proof is *not* found. Dually to deletion by weight, an unsound inference could suppress literals in clauses that are too heavy.

[3] Fairness guarantees that inferences are done systematically, in such a way that every theorem has a minimal proof in the limit: see [4] for details.

We consider a single unsound inference rule: adding an arbitrary clause C. This rule is unsound because C may not be implied by F. This rule is simple, but can simulate different kinds of unsound inferences. Suppose we want to suppress the literals D in $C \vee D$, then we can simply add C, which subsumes $C \vee D$. Suppose a clause $C[t]$ contains a deep term t, and we want to replace it with a constant a. We can accomplish this by adding $t \simeq a$. The idea is to extend DPLL($\Gamma + \mathcal{T}$) with a *reversible* unsound inference rule. We say it is *reversible*, because we track the consequences of the clauses added by this rule.

DPLL($\Gamma + \mathcal{T}$) works on *hypothetical clauses* of the form $H \triangleright C$, where C is a clause (i.e., a disjunction of literals), and H is the set of ground literals, from the candidate model built by DPLL($\Gamma+\mathcal{T}$), that C depends on, in the sense that they were used as premises to infer C by Γ-inferences. The set of hypotheses should be interpreted as a conjunction, and a hypothetical clause $(l_1 \wedge \ldots \wedge l_n) \triangleright (l'_1 \vee \ldots l'_m)$ should be interpreted as $\neg l_1 \vee \ldots \vee \neg l_n \vee l'_1 \vee \ldots \vee l'_m$. In this context, rather than merely adding a clause C, the unsound inference rule introduces a hypothetical clause $\lceil C \rceil \triangleright C$, where $\lceil C \rceil$ is a new propositional variable that is used to track the consequences of adding C. Note that the hypothetical clause $\lceil C \rceil \triangleright C$ is semantically equivalent to $\neg\lceil C \rceil \vee C$. This clause does not change the satisfiability of the input formula because $\lceil C \rceil$ is a new propositional variable.

The DPLL($\Gamma + \mathcal{T}$) calculus is described as a transition system [11]. States of the transition system are of the form $M \parallel F$, where M is a sequence of *assigned literals*, and F a set of *hypothetical clauses*. Intuitively, M represents a partial assignment to ground literals, with their justifications, and therefore it represents a partial model, or a set of candidate models. An assigned literal can be either a *decided literal* or an *implied literal*. A decided literal represents a guess, and an implied literal l_C a literal l that was implied by a clause C. No assigned literal occurs twice in M nor does it occur negated in M. If neither l nor $\neg l$ appears in M, then l is said to be *undefined*. The initial state is $\parallel F_0$, where F_0 is the set $\{\emptyset \triangleright C \mid C \in \mathcal{R} \uplus P\}$. During conflict resolution, we also use states of the form $M \parallel F \parallel C$, where C is a ground clause. In the following, *clauses*(F) denotes the set $\{C \mid H \triangleright C \in F\}$, $M \models_P C$ indicates that M propositionally satisfies C, and if C is the clause $l_1 \vee \ldots \vee l_n$, then $\neg C$ is the formula $\neg l_1 \wedge \ldots \wedge \neg l_n$. We use *lits*$(M)$ to denote the set of assigned literals, *ngclauses*(F) for the subset of non-ground clauses of *clauses*(F), and *clauses*$^\star(M \parallel F)$ for *ngclauses*$(F) \cup$ *lits*(M). We also write C instead of $\emptyset \triangleright C$.

We extend the calculus with the rule UnsoundIntro. This rule introduces an arbitrary clause C into F, and it adds the ground literal $\lceil C \rceil$ to M, where $\lceil C \rceil$ is a new propositional variable used as a label for clause C. The idea is to record the fact that we are *guessing* C.

UnsoundIntro

$$M \parallel F \qquad \Longrightarrow M\ \lceil C \rceil \parallel F,\ \lceil C \rceil \triangleright C \qquad \textbf{if} \begin{cases} C \notin \mathit{clauses}(F), \\ \lceil C \rceil \ \mathit{is\ new}, \\ \lceil C \rceil, \neg\lceil C \rceil \notin M, \end{cases}$$

where the side condition prevents the system from adding C, if it is already known to be inconsistent with the partial model M.

In order to combine the theories in $\mathcal{T} = \bigcup_{i=1}^{n} \mathcal{T}_i$ and $\mathcal{R}$ in the Nelson-Oppen scheme, we need to communicate to $\mathcal{R}$ the (disjunctions of) equalities between constants entailed by $\mathcal{T}$ and P. The next inference rule takes care of this requirement, which we relax as in [12], because the $\mathcal{T}_i$-solvers for linear arithmetic and bit-vectors can build a specific candidate $\mathcal{T}$-model for M, that we denote by $\mathsf{model}(M)$. The idea is to inspect $\mathsf{model}(M)$ and propagate all the equalities it implies, hedging that they are consistent with $\mathcal{R}$. Since these equalities are *guesses*, if one of them is inconsistent with $\mathcal{R}$, backtracking will be used to fix $\mathsf{model}(M)$. The rationale for this approach is practical: it is generally far less expensive to enumerate the equalities satisfied in a particular $\mathcal{T}$-model than those satisfied by all $\mathcal{T}$-models consistent with M; the number of equalities that really matter is small in practice.

PropagateEq

$$M \parallel F \qquad \Longrightarrow M\ t \simeq s \parallel F \qquad \textbf{if} \quad \begin{cases} t \text{ and } s \text{ are ground,} \\ t, s \text{ occur in } F, \\ (t \simeq s) \text{ is undefined in } M, \\ \mathsf{model}(M)(t) = \mathsf{model}(M)(s). \end{cases}$$

The basic and theory propagation rules of DPLL($\Gamma + \mathcal{T}$) are repeated from [11] in Figure 1.

The interface with the inference system Γ is realized by the Deduce rule: assume γ is an inference rule of Γ with n premises, $\{H_1 \rhd C_1, \ldots, H_m \rhd C_m\}$ is a set of hypothetical clauses in F, $\{l_{m+1}, \ldots, l_n\}$ is a set of assigned literals in M, and $\mathsf{H}(\gamma)$ denotes the set $H_1 \cup \ldots \cup H_m \cup \{l_{m+1}, \ldots, l_n\}$; then γ is applied to the set of premises $\mathsf{P}(\gamma) = \{C_1, \ldots, C_m, l_{m+1}, \ldots, l_n\}$, and the conclusion $\mathsf{C}(\gamma)$ is added to F as $\mathsf{H}(\gamma) \rhd \mathsf{C}(\gamma)$. The hypotheses of the clauses $H_i \rhd C_i$ are hidden from the inference rules in Γ. Our Deduce rule is slightly different from its predecessor, named Deduce$^\sharp$ in [11]: Deduce$^\sharp$ allowed Γ to use as premises non-ground clauses and ground unit clauses in $\mathit{clauses}(F)$, whereas our Deduce allows it to use only non-ground clauses in $\mathit{clauses}(F)$. This is a consequence of the addition of PropagateEq, which adds the relevant ground unit clauses directly to M, so that Γ finds them in $\mathit{lits}(M)$. This is also the reason why we let PropagateEq add equalities between ground terms and not only between constants.

We say a hypothetical clause $H \rhd C$ is in *conflict* if every literal in C is complementary to an assigned literal. The Conflict rule converts a hypothetical conflict clause $H \rhd C$ into a regular clause by negating its hypotheses, and puts the DPLL($\Gamma + \mathcal{T}$) system in conflict resolution mode. The Explain rule unfolds literals from conflict clauses that were produced by unit propagation. Any clause derived by Explain can be added to F by the Learn rule, because it is a logical consequence of the original set of clauses. The rule Backjump drives the DPLL($\Gamma + \mathcal{T}$) system back from conflict resolution to search mode, and it unassigns at least one decided literal (l' in the rule definition). All hypothetical clauses $H \rhd C$ which contain hypotheses that will be unassigned by the Backjump rule are deleted. Note that a learnt clause D may contain $\neg\lceil C \rceil$. In this case, the clause D is recording the context where guessing the clause C is unsound.

Decide

$$M \parallel F \quad \Longrightarrow M\ l \parallel F \quad \textbf{if} \quad \begin{cases} l \text{ is ground,} \\ l \text{ or } \neg l \text{ occurs in } F, \\ l \text{ is undefined in } M. \end{cases}$$

UnitPropagate

$$M \parallel F, H \triangleright (C \vee l) \quad \Longrightarrow M\ l_{H \triangleright (C \vee l)} \parallel F, H \triangleright (C \vee l) \quad \textbf{if} \quad \begin{cases} l \text{ is ground,} \\ M \models_P \neg C, \\ l \text{ is undefined in } M. \end{cases}$$

Deduce

$$M \parallel F \quad \Longrightarrow M \parallel F, \mathsf{H}(\gamma) \triangleright \mathsf{C}(\gamma) \quad \textbf{if} \quad \begin{cases} \gamma \in \Gamma, \\ \mathsf{P}(\gamma) \subseteq \mathit{clauses}^\star(M \parallel F), \\ \mathsf{C}(\gamma) \notin \mathit{clauses}(F). \end{cases}$$

Conflict

$$M \parallel F, H \triangleright C \quad \Longrightarrow M \parallel F, H \triangleright C \parallel \neg H \vee C \quad \textbf{if} \quad M \models_P \neg C$$

Explain

$$M \parallel F \parallel C \vee \bar{l} \quad \Longrightarrow M \parallel F \parallel \neg H \vee D \vee C \quad \textbf{if} \quad l_{H \triangleright (D \vee l)} \in M$$

Learn

$$M \parallel F \parallel C \quad \Longrightarrow M \parallel F, C \parallel C \quad \textbf{if} \quad C \notin \mathit{clauses}(F)$$

Backjump

$$M\ l'\ M' \parallel F \parallel C \vee l \quad \Longrightarrow M\ l_{C \vee l} \parallel F' \quad \textbf{if} \quad \begin{cases} M \models_P \neg C, \\ l \text{ is undefined in } M, \\ F' = \left\{ \begin{array}{l} H \triangleright C \in F \mid \\ \quad H \cap \mathit{lits}(l'\ M') = \emptyset \end{array} \right\} \end{cases}$$

Unsat

$$M \parallel F \parallel \Box \quad \Longrightarrow \mathit{unsat}$$

T-Propagate

$$M \parallel F \quad \Longrightarrow M\ l_{(\neg l_1 \vee \ldots \vee \neg l_n \vee l)} \parallel F \quad \textbf{if} \quad \begin{cases} l \text{ is ground and occurs in } F, \\ l \text{ is undefined in } M, \\ l_1, \ldots, l_n \in \mathit{lits}(M), \\ l_1, \ldots, l_n \models_T l. \end{cases}$$

T-Conflict

$$M \parallel F \quad \Longrightarrow M \parallel F \parallel \neg l_1 \vee \ldots \vee \neg l_n \quad \textbf{if} \quad \begin{cases} l_1, \ldots, l_n \in \mathit{lits}(M), \\ l_1, \ldots, l_n \models_T \mathit{false}. \end{cases}$$

Fig. 1. Basic and theory propagation rules

It was proved in [11] that DPLL($\Gamma + \mathcal{T}$) is refutationally complete when $\mathcal{T}$ is empty. We prove a stronger result for the case where $\mathcal{T}$ is not empty. We say a state $M \parallel F$ is *saturated* if the only applicable rule is UnsoundIntro.

Theorem 1. *If the initial set of clauses $S = \mathcal{R} \uplus P$ is smooth, and Γ has the reduction property for counterexamples, whenever $M \parallel F$ is saturated, S is satisfiable modulo the background theory $\mathcal{T}$.*

All inference systems considered in the rest of this paper satisfy the reduction property for counterexamples.

We assign an *inference depth* to every clause in $\mathit{clauses}(F)$ and literal in $\mathit{lits}(M)$. Intuitively, the inference depth of a clause C indicates the depth of the derivation needed to produce C. More precisely, all clauses in the original set of clauses have inference depth 0. If a clause C is produced using the Deduce rule, and n is the maximum inference depth of the premises, then the inference

depth of C is $n+1$. The inference depth of a literal l_C in M is equal to the inference depth of C. If l is a decided literal, and n is the minimum inference depth of the clauses in F that contain l, then the inference depth of l is n. We say DPLL($\Gamma + \mathcal{T}$) is $\langle k_d, k_u \rangle$-bounded if Deduce is restricted to premises with inference depth $< k_d$, and UnsoundIntro can only be applied k_u times.

Theorem 2. *$\langle k_d, k_u \rangle$-bounded DPLL($\Gamma + \mathcal{T}$) always terminates.*

A state $M \parallel F$ is *stuck* at k_d if the only applicable rules are UnsoundIntro and Deduce, and Deduce is only applicable to premises with inference depth $\geq k_d$. Theorem 2 suggests a simple saturation strategy where the bounds k_d and k_u are increased whenever the procedure reaches a blocked state.

We use U to denote a sequence of "unsound axioms" introduced by UnsoundIntro. In the next section, we investigate some examples where DPLL($\Gamma + \mathcal{T}$) is a decision procedure for a smooth set of formulæ S. This is accomplished by showing that for some sequence of "unsound axioms" U, there are k_d and k_u, such that $\langle k_d, k_u \rangle$-bounded DPLL($\Gamma + \mathcal{T}$) is guaranteed to terminate in the *unsat* state, whenever S is unsatisfiable, and in a state $M \parallel F$ which is not *stuck* at k_d, whenever S is satisfiable.

Due to space limitations, we refer to [11] for the contraction inference rules of DPLL($\Gamma + \mathcal{T}$). DPLL($\Gamma + \mathcal{T}$) assigns a *scope level* to each literal in M. The scope level of a literal l, *level*(l), in $M\ l\ M'$, is equal to the number of decided literals in $M\ l$. The level of a set of literals H is $\mathit{level}(H) = \mathit{max}\{\mathit{level}(l) \mid l \in H\}$. A contraction rule γ from Γ is generalized to hypothetical clauses as follows: given a main premise $H \rhd C$, and side premises $H_2 \rhd C_2, \ldots, H_m \rhd C_m$, and $l_{m+1}, \ldots, l_n$, taken from F and *lits*(M), respectively, let $H' = H_2 \cup \ldots \cup H_m \cup \{l_{m+1}, \ldots, l_n\}$. Assume that γ applies to the premises $C, C_2, \ldots, C_m, l_{m+1}, \ldots, l_n$. If $\mathit{level}(H) \geq \mathit{level}(H')$, we claim it is safe to delete $H \rhd C$. In contrast, if $\mathit{level}(H) < \mathit{level}(H')$, then it is only safe to *disable* the clause $H \rhd C$ until *level*(H') is backjumped. A *disabled* clause is not deleted, but it is not used as premise until it is re-enabled.

Example 1. Let $\mathcal{R}$ be $\{\neg(x \sqsubseteq y) \vee \neg(y \sqsubseteq z) \vee x \sqsubseteq z,\ \neg(x \sqsubseteq y) \vee f(x) \sqsubseteq f(y)\}$, and P be $\{a \sqsubseteq b,\ a \sqsubseteq f(c),\ \neg(a \sqsubseteq c)\}$. Assume Γ features Resolution, Superposition and Simplification. If UnsoundIntro adds $\lceil f(x) \simeq x \rceil \rhd f(x) \simeq x$, the monotonicity axiom and $a \sqsubseteq f(c)$ get rewritten. Note that $\lceil f(x) \simeq x \rceil$ is a decision literal, and $\mathit{level}(\lceil f(x) \simeq x \rceil) = 1$. Thus, the rewriting step only disables $a \sqsubseteq f(c)$, and adds $\lceil f(x) \simeq x \rceil \rhd a \sqsubseteq c$ to F. Resolution generates the conflict clause $\lceil f(x) \simeq x \rceil \rhd \Box$. Using the conflict resolution rules, the literal $\neg\lceil f(x) \simeq x \rceil$ is added to M, preventing DPLL($\Gamma + \mathcal{T}$) from guessing $f(x) \simeq x$ again. Next, if UnsoundIntro adds $\lceil f(f(x)) \simeq x \rceil \rhd f(f(x)) \simeq x$, monotonicity and $a \sqsubseteq b$ produce only $f(a) \sqsubseteq f(b)$, while monotonicity and $a \sqsubseteq f(c)$ produce only $f(a) \sqsubseteq f(f(c))$, which is disabled and replaced by $\lceil f(f(x)) = x \rceil \rhd f(a) \sqsubseteq c$. Then, DPLL($\Gamma + \mathcal{T}$) reaches a saturated state, and satisfiability is detected.

Example 2. Let $\mathcal{R}$ be $\{\neg(x \sqsubseteq y) \vee \neg(y \sqsubseteq z) \vee x \sqsubseteq z\}$, P be $\{a \sqsubseteq b_1,\ b_2 \sqsubseteq c,\ \neg(a \sqsubseteq c), b_1 \leq b_2, b_1 > b_2 - 1\}$, and $\mathcal{T}$ be the theory of linear integer arithmetic. Assume Γ is Hyperresolution, Superposition and Simplification. UnitPropagate adds the

literals of P to M. In the model $\mathsf{model}(M)$ maintained by the linear arithmetic solver, $\mathsf{model}(M)(b_1) = \mathsf{model}(M)(b_2)$. Thus, PropagateEq *guesses* the equation $b_1 \simeq b_2$. Say $b_2 \succ b_1$: Simplification rewrites $b_2 \sqsubseteq c$ to $b_1 \sqsubseteq c$. Hyperresolution derives $a \sqsubseteq c$ from $a \sqsubseteq b_1$, $b_1 \sqsubseteq c$ and transitivity, so that an inconsistency is detected. DPLL($\Gamma + \mathcal{T}$) backtracks and adds $\neg(b_1 \simeq b_2)$ to M. T-Conflict detects the inconsistency between this literal and $\{b_1 \leq b_2, b_1 > b_2 - 1\}$. The conflict resolution rules are applied again and the empty clause is produced.

5 Essentially Finite Theories

We say a structure Φ is *essentially finite* with respect to the function symbol f if $\Phi(f)$ has finite range. Essential finiteness is slightly weaker than finiteness, because it admits an infinite domain provided the range of $\Phi(f)$ is finite.

Theorem 3. *If Φ is an essentially finite structure with respect to a monadic function symbol f, then there exist k_1, k_2, $k_1 \neq k_2$, such that $\Phi \models f^{k_1}(x) \simeq f^{k_2}(x)$.*

Proof. For all $v \in |\Phi|$, we call *f-chain* starting at v, the sequence:

$$v = \Phi(f)^0(v),\ \Phi(f)^1(v),\ \Phi(f)^2(v), \ldots,\ \Phi(f)^i(v), \ldots$$

Since $\Phi(f)$ has finite range, there exist q_1, q_2, with $q_1 \neq q_2$, such that $\Phi(f)^{q_1}(v) = \Phi(f)^{q_2}(v)$. Say that $q_1 > q_2$. Then we call *size*, denoted $\mathsf{sz}(\Phi, f, v)$, and *prefix*, denoted $\mathsf{pr}(\Phi, f, v)$, of the f-chain starting at v, the smallest q_1 and q_2, respectively, such that $\Phi(f)^{q_1}(v) = \Phi(f)^{q_2}(v)$ and $q_1 > q_2$. We term *lasso*, denoted $\mathsf{ls}(\Phi, f, v)$, of the f-chain starting at v, the difference between size and prefix, that is, $\mathsf{ls}(\Phi, f, v) = \mathsf{sz}(\Phi, f, v) - \mathsf{pr}(\Phi, f, v)$. We say that $\Phi(f)^n(v)$ is *in the lasso* of the f-chain starting at v, if $n \geq \mathsf{pr}(\Phi, f, v)$. Clearly, for all elements u in the lasso of the f-chain starting at v, $\Phi(f)^m(u) = u$, when $m = \mathsf{ls}(\Phi, f, v)$. Also, for all multiples of the lasso, that is, for all $l = h \cdot \mathsf{ls}(\Phi, f, v)$ for some $h > 0$, $\Phi(f)^l(u) = u$. Let $p = \mathsf{max}\{\mathsf{pr}(\Phi, f, v) \mid v \in \mathsf{range}(\Phi(f))\} + 1$ and $l = \mathsf{lcm}\{\mathsf{ls}(\Phi, f, v) \mid v \in \mathsf{range}(\Phi(f))\}$, where lcm abbreviates least common multiple. We claim that $\Phi \models f^{p+l}(x) \simeq f^p(x)$, that is, $k_1 = p + l$ and $k_2 = p$. By way of contradiction, assume that for some $v \in |\Phi|$, $\Phi(f)^{p+l}(v) \neq \Phi(f)^p(v)$. Take the f-chain starting at v: $\Phi(f)^p(v)$ is in the lasso of this chain, because $p \geq \mathsf{pr}(\Phi, f, v)$. Since l is a multiple of $\mathsf{ls}(\Phi, f, v)$, we have $\Phi(f)^{p+l}(v) = \Phi(f)^l(\Phi(f)^p(v)) = \Phi(f)^p(v)$, a contradiction. □

Example 3. Let Φ be a structure such that $|\Phi| = \{v_0, v_1, v_2, \ldots, v_9, \ldots\}$, and let $\Phi(f)$ be the function defined by the following mapping: $\{v_0 \mapsto v_1, v_1 \mapsto v_2, v_2 \mapsto v_3, v_3 \mapsto v_4, v_4 \mapsto v_2, v_5 \mapsto v_6, v_6 \mapsto v_7, v_7 \mapsto v_8, v_8 \mapsto v_5, * \mapsto v_9\}$, where $*$ stands for any other element. The f-chain starting at v_0 has $\mathsf{pr}(\Phi, f, v_0) = 2$, $\mathsf{sz}(\Phi, f, v_0) = 5$ and $\mathsf{ls}(\Phi, f, v_0) = 3$. The f-chain starting at v_5 has $\mathsf{pr}(\Phi, f, v_5) = 0$, $\mathsf{sz}(\Phi, f, v_5) = 4$ and $\mathsf{ls}(\Phi, f, v_5) = 4$. Then, $p = 2+1 = 3$, $l = 12$, $k_1 = p+l = 15$ and $k_2 = p = 3$, and $\Phi \models f^{15}(x) \simeq f^3(x)$.

To identify classes of problems for which DPLL($\Gamma + \mathcal{T}$) is a decision procedure, we focus on theories $\mathcal{R}$ that satisfy either one of the following properties:

Definition 4. *$\mathcal{R}$ has the* finite model property*, if for all sets P of ground $\mathcal{R}$-clauses, such that $\mathcal{R} \uplus P$ is satisfiable, $\mathcal{R} \uplus P$ has a model Φ with finite $|\Phi|$.*

Definition 5. *Let $\mathcal{R}$ be a presentation whose signature contains a single monadic function symbol f. $\mathcal{R}$ is* essentially finite*, if for all sets P of ground $\mathcal{R}$-clauses, such that $\mathcal{R} \uplus P$ is satisfiable, $\mathcal{R} \uplus P$ has a model Φ, such that* $\mathsf{range}(\Phi(f))$ *is finite.*

We show that essentially finite theories can give rise to decision procedures if clause length is bounded.

Theorem 4. *Let $\mathcal{R}$ be an essentially finite theory and P a set of ground clauses. Let Γ be a rewrite-based inference system. Consider a DPLL($\Gamma + \mathcal{T}$) procedure where* UnsoundIntro *progressively adds all equations of the form $f^j(x) \simeq f^k(x)$ with $j > k$. Then DPLL($\Gamma + \mathcal{T}$) is a decision procedure for the satisfiability modulo $\mathcal{T}$ of smooth problems in the form $\mathcal{R} \uplus P$ if there exists an n such that no clause created contains more than n literals.*

Proof. If $\mathcal{R} \uplus P$ is unsatisfiable, then by completeness DPLL($\Gamma + \mathcal{T}$) will generate the empty clause when k_d becomes large enough. If $\mathcal{R} \uplus P$ is satisfiable, choose k_u large enough to contain the axiom $f^{k_1}(x) \simeq f^{k_2}(x)$ as given in Theorem 3. We need to prove that if k_d is large enough, DPLL($\Gamma + \mathcal{T}$) will not get stuck at k_d. To do that, we prove that only a finite number of clauses are generated for unbounded k_d for the given k_u. The axiom $f^{k_1}(x) \simeq f^{k_2}(x)$ is oriented into the rewrite rule $f^{k_1}(x) \to f^{k_2}(x)$. This guarantees that no term $f^k(t)$ with $k > k_1$ is kept. Since no clause can contain more than n literals, only a finite number of clauses can be derived for an unbounded k_d. □

Assume that Γ is Superposition with negative selection plus Hyperresolution[4]. If $\mathcal{R}$ is Horn, Superposition is Unit Superposition, which does not increase clause length, and Hyperresolution only generates positive unit clauses, so that no clause containing more than n literals can be produced. If $\mathcal{R}$ is a set of nonequality clauses with no more than two literals each, and Γ is Resolution plus Simplification (to apply $f^{k_1}(x) \to f^{k_2}(x)$), then all generated clauses contain at most two literals. To give further examples, we need the following:

Definition 6. *A clause $C = \neg l_1 \vee \ldots \vee \neg l_n \vee l_{n+1} \vee \ldots \vee l_{n+m}$ is* ground-preserving *if $\bigcup_{j=n+1}^{n+m} Var(l_j) \subseteq \bigcup_{j=1}^{n} Var(l_j)$. A set is* ground-preserving *if all its clauses are.*

In a ground-preserving[5] set the only positive clauses are ground. If $\mathcal{R}$ is ground-preserving, Hyperresolution only generates ground clauses; Superposition with

[4] Hyperresolution is realized by Resolution with negative selection rule.

[5] This notion is a weakening of that of "positive variable dominated" clause of Definition 3.18 in [9].

negative selection yields either ground clauses or ground-preserving clauses with decreasing number of variable positions, so that no new non-ground terms can be created, and only finitely many non-ground ground-preserving clauses can be derived. If $\mathcal{R}$ is also essentially finite, the depth of terms is limited by Simplification by $f^{k_1}(x) \to f^{k_2}(x)$, so that only finitely many ground clauses can be generated. Below, we show that some specific theories relevant to the axiomatization of type systems in programming languages are essentially finite and satisfy the properties of Theorem 4. Given the axioms

$$\textsf{Reflexivity} \quad x \sqsubseteq x \tag{1}$$

$$\textsf{Transitivity} \quad \neg(x \sqsubseteq y) \vee \neg(y \sqsubseteq z) \vee x \sqsubseteq z \tag{2}$$

$$\textsf{Anti-Symmetry} \quad \neg(x \sqsubseteq y) \vee \neg(y \sqsubseteq x) \vee x \simeq y \tag{3}$$

$$\textsf{Monotonicity} \quad \neg(x \sqsubseteq y) \vee f(x) \sqsubseteq f(y) \tag{4}$$

$$\textsf{Tree-Property} \quad \neg(z \sqsubseteq x) \vee \neg(z \sqsubseteq y) \vee x \sqsubseteq y \vee y \sqsubseteq x \tag{5}$$

$\mathsf{MI} = \{(1), (2), (3), (4)\}$ presents a type system with *multiple inheritance*, and $\mathsf{SI} = \mathsf{MI} \uplus \{(5)\}$ presents a type system with *single inheritance*, where $\sqsubseteq$ is the subtype relationship and f is a type constructor.

Theorem 5. SI *has the finite model property hence it is essentially finite.*

Proof. Assume $\mathsf{SI} \uplus P$ is satisfiable, and let Φ be a model for it. It is sufficient to show there is a finite model Φ'. Let T_P be the set of subterms of terms in P, and V_P be the set $\Phi(T_P)$. Since P is finite and ground, V_P is finite. Let $|\Phi'|$ be $V_P \cup \{r\}$, where r is an element not in V_P. Then, we define $\Phi'(\sqsubseteq)(v_1, v_2)$ as:

$$r = v_2 \text{ or } (v_1, v_2) \in \Phi(\sqsubseteq)$$

Intuitively, r is a new maximal element. $\langle |\Phi'|, \Phi'(\sqsubseteq)\rangle$ is a poset and $\Phi'(\sqsubseteq)$ satisfies the $\textsf{Tree-Property}$. Now, we define an auxiliary function $g\colon |\Phi'| \to |\Phi'|$ as:

$$g(v) = \begin{cases} \Phi(f)(v) & \text{if } f(t) \in T_P, \text{ and } \Phi(t) = v; \\ r & \text{otherwise.} \end{cases}$$

Let dom_f, the relevant domain of f, be the set $\{\Phi(t) \mid f(t) \in T_P\} \cup \{r\}$. With a small abuse of notation, we use $v \sqsubseteq w$ to denote $(v, w) \in \Phi'(\sqsubseteq)$. Then, we define $\Phi'(f)(v)$ as $g(w)$, where w is an element in $|\Phi'|$ such that $v \sqsubseteq w$, $w \in \mathsf{dom}_f$, and for all w', $v \sqsubseteq w'$ and $w' \in \mathsf{dom}_f$ imply $w \sqsubseteq w'$. This function is well defined because $\Phi'(\sqsubseteq)$ satisfies the $\textsf{Tree-Property}$, r is the maximal element of $|\Phi'|$, and $r \in \mathrm{dom}_f$. Moreover, $\Phi'(f)$ is monotonic with respect to $\Phi'(\sqsubseteq)$. □

Definition 7. *Let $\langle A, \sqsubseteq\rangle$ be a poset. The* Dedekind-MacNeille completion [18] *of $\langle A, \sqsubseteq\rangle$ is the unique complete lattice $\langle B, \preceq\rangle$ satisfying the following properties.*

- *There is an injection α from A to B such that: $v_1 \sqsubseteq v_2$ iff $\alpha(v_1) \preceq \alpha(v_2)$,*
- *Every subset of B has a greatest (least) lower bound, and*
- *B is finite if A is finite. Actually, B is a subset of 2^A.*

Theorem 6. MI *has the finite model property hence it is essentially finite.*

Proof. The construction used for SI does not work for MI, because without the Tree-Property the w in the definition of $\Phi'(f)(v)$ may not be unique for a given v. First, we define an auxiliary structure Φ_0 such that $|\Phi_0| = V_P$, $\Phi_0(\sqsubseteq) = \Phi(\sqsubseteq)|_{V_P}$, and $\Phi_0(f)$ is defined as:

$$\Phi_0(f)(v) = \begin{cases} \Phi(f)(v) & \text{if } f(t) \in T_P, \text{ and } \Phi(t) = v, \\ w & \text{otherwise,} \end{cases}$$

where w is some element of V_P. Note that $\langle V_P, \Phi_0(\sqsubseteq)\rangle$ is a poset. Let dom_f be the set $\{\Phi(t) \mid f(t) \in T_P\}$. Then, following [10] we use the Dedekind-MacNeille completion to complete $\langle V_P, \Phi_0(\sqsubseteq)\rangle$ into a complete lattice $\langle B, \preceq\rangle$. We use $glb(S)$ to denote the greatest lower bound of a subset S of B. Now, we define a finite model Φ' for MI $\uplus P$ with domain $|\Phi'| = B$, in the following way:

$$\begin{aligned} \Phi'(c) &= \alpha(\Phi_0(c)) \quad \text{for every constant } c \text{ in } T_P, \\ \Phi'(\sqsubseteq) &= \preceq, \\ \Phi'(f)(v) &= glb(\{\alpha(\Phi_0(f)(w)) \mid w \in V_P,\ w \in \mathsf{dom}_f,\ v \preceq \alpha(w)\}). \end{aligned}$$

The function $\Phi'(f)$ is monotonic with respect to $\Phi'(\sqsubseteq)$. The structure Φ' satisfies P because for every term t in T_P, we have $\Phi'(t) = \alpha(\Phi(t))$. Moreover, the $\sqsubseteq$-literals in P are satisfied because the lattice $\langle B, \preceq\rangle$ is a Dedekind-MacNeille completion of Φ_0 which is a restriction of Φ. □

Now we show that DPLL($\Gamma + \mathcal{T}$) with the UnsoundIntro rule is a decision procedure for MI and SI.

Theorem 7. *Let P be a set of ground clauses. Let Γ be Hyperresolution plus Superposition and Simplification. Consider a DPLL($\Gamma + \mathcal{T}$) procedure where UnsoundIntro progressively adds all equations of the form $f^j(x) \simeq f^k(x)$ with $j > k$. Then DPLL($\Gamma + \mathcal{T}$) is a decision procedure for the satisfiability modulo $\mathcal{T}$ of smooth problems in the form* MI $\uplus P$.

Proof. Since MI is essentially finite, we only need to show that we never generate a clause with more than n literals. This follows from the fact that MI is a Horn theory. □

Theorem 8. *Let P be a set of ground clauses. Let Γ be Hyperresolution plus Superposition and Simplification. Consider a DPLL($\Gamma + \mathcal{T}$) procedure where UnsoundIntro progressively adds all equations of the form $f^j(x) \simeq f^k(x)$ with $j > k$. Then DPLL($\Gamma + \mathcal{T}$) is a decision procedure for the satisfiability modulo $\mathcal{T}$ of smooth problems in the form* SI $\uplus P$.

Proof. Since SI is essentially finite, we need to show that only finitely many clauses can be generated. SI is not Horn, because of Tree-Property, and it is not ground-preserving, because of Reflexivity. Since all the axioms besides Reflexivity are ground-preserving, any inference will yield either a ground clause or a non-ground ground-preserving clause with fewer variable positions. We just need to consider a Hyperresolution which includes Reflexivity. All those inferences either yield a tautology, a subsumed clause, or a ground clause. □

In Spec# [3], the axiomatization of the type system also contains the axioms $\mathsf{TR} = \{\neg(g(x) \simeq \mathit{null}),\ h(g(x)) \simeq x\}$, where the function g represents the *type representative* of some type. The first axiom states that the representative is never the constant *null*, and the second states that g has an inverse, hence it is injective. Note that any model of TR must be infinite.

Theorem 9. *Let P be a set of ground clauses. Let Γ be Hyperresolution plus Superposition and Simplification. Consider a DPLL($\Gamma + \mathcal{T}$) procedure where UnsoundIntro progressively adds all equations of the form $f^j(x) \simeq f^k(x)$ with $j > k$. Then DPLL($\Gamma + \mathcal{T}$) is a decision procedure for the satisfiability modulo $\mathcal{T}$ of smooth problems in the form $\mathsf{MI} \uplus \mathsf{TR} \uplus P$ and $\mathsf{SI} \uplus \mathsf{TR} \uplus P$.*

Proof. Γ applied to TR and ground equations only generates ground equations of non-increasing depth, hence it terminates. Since MI (SI) and TR do not share function symbols and are variable inactive, Γ terminates also on their union. □

6 Discussion

The DPLL($\Gamma + \mathcal{T}$) system integrates DPLL($\mathcal{T}$) with a generic first-order engine Γ to combine the strengths of DPLL, efficient solvers for special theories such as linear arithmetic, and first-order reasoning based on superposition and resolution. The study in [6] was concerned with using the first-order engine alone as decision procedure, without integrating an SMT-solver. In the method of [7], the first-order engine is used as a pre-processor to compile the theory $\mathcal{R}$ and reduce it to a theory that DPLL($\mathcal{T}$) alone can handle. Thus, it is a two-stage approach. In DPLL($\Gamma + \mathcal{T}$) the first-order engine is tightly integrated within DPLL($\mathcal{T}$). The downside of such a tight integration was that refutational completeness had not been established, except in the case where the background theory $\mathcal{T}$ is empty. In this paper we advanced the DPLL($\Gamma + \mathcal{T}$) approach by giving conditions under which it is refutationally complete when $\mathcal{T}$ is not empty.

Then, we introduced a new calculus that combines DPLL($\Gamma + \mathcal{T}$) with unsound theorem proving. The purpose is to try to enforce termination by introducing additional axioms as hypothesis. A framework for unsound theorem proving was originally given in [17] along with some examples. In the current paper we have provided a mechanism for the prover to detect any unsoundness introduced by the added axioms and recover from it, and we have instantiated the framework with concrete examples for which unsound theorem proving becomes a decision procedure. Some of these examples include monotonicity axioms. Another approach to handle such axioms is *locality*: for instance, extending a theory with a monotonicity axiom is a *local extension* [22,16]. However, in the applications that motivate our research, there is no guarantee that all relevant instances of $\mathcal{T} \uplus \mathcal{R} \uplus P$ can be seen as hierarchies of local extensions.

Directions for future work include extensions to more presentations, including, for instance, cases where the signature of $\mathcal{R}$ features also non-monadic function symbols (e.g., to cover axioms such as $y \sqsubseteq x \wedge u \sqsubseteq v \Rightarrow \mathit{map}(x, u) \sqsubseteq \mathit{map}(y, v)$). Another open issue is some duplication of reasoning on ground unit clauses in

DPLL($\Gamma+\mathcal{T}$), due to the fact that ground unit clauses are seen by both Γ and the congruence closure (CC) algorithm within DPLL($\mathcal{T}$). Using the CC algorithm to compute the completion of the set of ground equations [14,21], and pass the resulting canonical system to Γ, would not solve the problem entirely, because this solution is not *incremental*, as the addition of a single ground equation requires recomputing the canonical system.

The class of formulæ that can be decided using DPLL($\Gamma + \mathcal{T}$) with unsound inferences includes axiomatizations of type systems, used in tools such as ESC/Java [13] and Spec# [3], which is significant evidence of the relevance of this work to applications.

Acknowledgments. Part of this work initiated during a visit of the first author with the Software Reliability Group of Microsoft Research in Redmond.

References

1. Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: New results on rewrite-based satisfiability procedures. ACM TOCL 10(1), 129–179 (2009)
2. Armando, A., Ranise, S., Rusinowitch, M.: A rewriting approach to satisfiability procedures. Inf. Comput. 183(2), 140–164 (2003)
3. Barnett, M., Leino, K.R.M., Schulte, W.: The spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
4. Bonacina, M.P., Dershowitz, N.: Abstract canonical inference. ACM TOCL 8(1), 180–208 (2007)
5. Bonacina, M.P., Echenim, M.: Rewrite-based satisfiability procedures for recursive data structures. In: Cook, B., Sebastiani, R. (eds.) Proc. 4th PDPAR Workshop, FLoC 2006. ENTCS, vol. 174(8), pp. 55–70. Elsevier, Amsterdam (2007)
6. Bonacina, M.P., Echenim, M.: On variable-inactivity and polynomial T-satisfiability procedures. J. Logic and Comput. 18(1), 77–96 (2008)
7. Bonacina, M.P., Echenim, M.: Theory decision by decomposition. J. Symb. Comput., 1–42 (to appear)
8. Bonacina, M.P., Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Decidability and undecidability results for Nelson-Oppen and rewrite-based decision procedures. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS, vol. 4130, pp. 513–527. Springer, Heidelberg (2006)
9. Caferra, R., Leitsch, A., Peltier, N.: Automated Model Building. Kluwer Academic Publishers, Amsterdam (2004)
10. Cantone, D., Zarba, C.G.: A decision procedure for monotone functions over bounded and complete lattices. In: de Swart, H. (ed.) TARSKI 2006. LNCS (LNAI), vol. 4342, pp. 318–333. Springer, Heidelberg (2006)
11. de Moura, L., Bjørner, N.: Engineering DPLL(T) + saturation. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 475–490. Springer, Heidelberg (2008)
12. de Moura, L., Bjørner, N.: Model-based theory combination. In: Krstić, S., Oliveras, A. (eds.) Proc. 5th SMT Workshop, CAV 2007. ENTCS, vol. 198(2), pp. 35–50. Elsevier, Amsterdam (2008)

13. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proc. PLDI, pp. 234–245 (2002)
14. Gallier, J., Narendran, P., Plaisted, D.A., Raatz, S., Snyder, W.: Finding canonical rewriting systems equivalent to a finite set of ground equations in polynomial time. J. ACM 40(1), 1–16 (1993)
15. Halpern, J.Y.: Presburger Arithmetic with unary predicates is Π_1^1 Complete. J. Symb. Logic 56, 637–642 (1991)
16. Jacobs, S.: Incremental instance generation in local reasoning. In: Notes 1st CEDAR Workshop, IJCAR 2008, pp. 47–62 (2008)
17. Lynch, C.A.: Unsound theorem proving. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 473–487. Springer, Heidelberg (2004)
18. MacNeille, H.M.: Partially ordered sets. Transactions of the American Mathematical Society 42, 416–460 (1937)
19. McCune, W.W.: Otter 3.3 reference manual. Technical Report ANL/MCS-TM-263, MCS Division, Argonne National Laboratory, Argonne, IL, USA (2003)
20. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM TOPLAS 1(2), 245–257 (1979)
21. Snyder, W.: A fast algorithm for generating reduced ground rewriting systems from a set of ground equations. J. Symb. Comput. (1992)
22. Sofronie-Stokkermans, V.: Hierarchic reasoning in local theory extensions. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 219–234. Springer, Heidelberg (2005)

Combinable Extensions of Abelian Groups

Enrica Nicolini, Christophe Ringeissen, and Michaël Rusinowitch

LORIA & INRIA Nancy Grand Est, France
FirstName.LastName@loria.fr

Abstract. The design of decision procedures for combinations of theories sharing some arithmetic fragment is a challenging problem in verification. One possible solution is to apply a combination method à la Nelson-Oppen, like the one developed by Ghilardi for unions of non-disjoint theories. We show how to apply this non-disjoint combination method with the theory of abelian groups as shared theory. We consider the completeness and the effectiveness of this non-disjoint combination method. For the completeness, we show that the theory of abelian groups can be embedded into a theory admitting quantifier elimination. For achieving effectiveness, we rely on a superposition calculus modulo abelian groups that is shown complete for theories of practical interest in verification.

1 Introduction

Decision procedures are the basic engines of the verification tools used to check the satisfiability of formulae modulo background theories, which may include axiomatizations of standard data-types such lists, arrays, bit-vectors, etc. Nowadays, there is a growing interest in applying theorem provers to construct decision procedures for theories of interest in verification [2,1,8,4]. The problem of incorporating some reasoning modulo arithmetic properties inside theorem provers is particularly challenging. Many works are concerned with the problem of building-in certain equational axioms, starting from the seminal contributions by Plotkin [21] and by Peterson and Stickel [20]. The case of Associativity-Commutativity has been extensively investigated since it appears in many equational theories, and among them, the theory of abelian groups is a very good candidate as fragment of arithmetic. Recently, the standard superposition calculus [19] has been extended to a superposition calculus modulo the built-in theory of abelian groups [12]. This work paves the way for the application of a superposition calculus modulo a fragment of arithmetic to build decision procedures of practical interest in verification. However, practical problems are often expressed in a combination of theories where the fragment of arithmetic is shared by all the other theories involved. In this case the classical Nelson-Oppen combination method cannot be applied since the theories share some arithmetic operators. An extension of the Nelson-Oppen combination method to the non-disjoint case has been proposed in [11]. This non-disjoint combination framework has been recently applied to the theory of Integer Offsets [18]. In this paper, our aim is to consider a more expressive fragment by studying the case of abelian groups.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 51–66, 2009.

The contributions of the paper are twofold. First, we show that abelian groups satisfy all the properties required to prove the completeness, the termination and the effectiveness of the non-disjoint extension of the Nelson-Oppen combination method. To prove the completeness, we show the existence of an extension of the theory of abelian groups having quantifier elimination and that behaves the same w.r.t. the satisfiability of constraints. Second, we identify a class of theories that extend the theory of abelian groups and for which a simplified constraint-free (but many-sorted) version of the superposition calculus introduced in [12] is proved to be complete. This superposition calculus allows us to obtain effective decision procedures that can be plugged into the non-disjoint extension of the Nelson-Oppen combination method.

This paper is organized as follows. Section 2 briefly introduces the main concepts and the non-disjoint combination framework. In Section 3, we show some very useful properties in order to use the theory of abelian groups, namely AG, in the non-disjoint combination framework, especially we prove the quantifier elimination of a theory that is an extension of AG. In Section 4, we present a calculus modulo AG. In Section 5, we show its refutational completeness and we study how this calculus may lead to combinable decision procedures. Examples are given in Section 6. We conclude with some final remarks in Section 7. Most of the proofs are omitted and can be found in [17].

2 Preliminaries

We consider a many-sorted language. A *signature* Σ is a set of sorts, functions and predicate symbols (each endowed with the corresponding arity and sort). We assume that, for each sort s, the equality "$=_s$" is a logical constant that does not occur in Σ and that is always interpreted as the identity relation over (the interpretation of) s; moreover, as a notational convention, we will often omit the subscript and we will shorten $=$ and $\neq$ with $\bowtie$. Again, as a matter of convention, we denote with $\Sigma^{\underline{a}}$ the signature obtained from Σ by adding a set $\underline{a}$ of new constants (each of them again equipped with its sort), and with $t\theta$ the application of a substitution θ to a term t. Σ-*atoms*, Σ-*literals*, Σ-*clauses*, and Σ-*formulae* are defined in the usual way, i.e. they must respect the arities of function and predicate symbols and the variables occurring in them must also be equipped with sorts (well-sortedness). The empty clause is denoted by $\Box$. A set of Σ-literals is called a Σ-*constraint*. Terms, literals, clauses and formulae are called *ground* whenever no variable appears in them; *sentences* are formulae in which free variables do not occur.

From the semantic side, we have the standard notion of a Σ-*structure* $\mathcal{M}$: it consists of non-empty pairwise disjoint domains M_s for every sort s and a sort- and arity-matching interpretation $\mathcal{I}$ of the function and predicate symbols from Σ. The truth of a Σ-formula in $\mathcal{M}$ is defined in any one of the standard ways. If $\Sigma_0 \subseteq \Sigma$ is a subsignature of Σ and if $\mathcal{M}$ is a Σ-structure, the Σ_0-*reduct* of $\mathcal{M}$ is the Σ_0-structure $\mathcal{M}_{|\Sigma_0}$ obtained from $\mathcal{M}$ by forgetting the interpretation of the symbols from $\Sigma \setminus \Sigma_0$.

A collection of Σ-sentences is a Σ-theory, and a Σ-theory T admits (or has) *quantifier elimination* iff for every formula $\varphi(\underline{x})$ there is a quantifier-free formula (over the same free variables $\underline{x}$) $\varphi'(\underline{x})$ such that $T \models \varphi(\underline{x}) \leftrightarrow \varphi'(\underline{x})$.

In this paper, we are concerned with the *(constraint) satisfiability problem* for a theory T, which is the problem of deciding whether a Σ-constraint is satisfiable in a model of T. Notice that a constraint may contain variables: since these variables may be equivalently replaced by free constants, we can reformulate the constraint satisfiability problem as the problem of deciding whether a finite conjunction of ground literals in a simply expanded signature $\Sigma^{\underline{a}}$ is true in a $\Sigma^{\underline{a}}$-structure whose Σ-reduct is a model of T.

2.1 A Brief Overview on Non-disjoint Combination

Let us consider now the constraint satisfiability problem w.r.t. a theory T that is the union of the two theories $T_1 \cup T_2$, and let us suppose that we have at our disposal two decision procedures for the constraint satisfiability problem w.r.t. T_1 and T_2 respectively. It is known (cf. [5]) that such a problem without any other assumption on T_1 and T_2 is undecidable; nevertheless, the following theorem holds:

Theorem 1 ([11]). *Consider two theories T_1, T_2 in signatures Σ_1, Σ_2 such that:*

1. *both T_1, T_2 have a decidable constraint satisfiability problem;*
2. *there is some universal theory T_0 in the signature $\Sigma_0 := \Sigma_1 \cap \Sigma_2$ such that:*
   (a) T_1, T_2 are both T_0-compatible;
   (b) T_1, T_2 are both effectively Noetherian extensions of T_0.

Then the $(\Sigma_1 \cup \Sigma_2)$-theory $T_1 \cup T_2$ also has a decidable constraint satisfiability problem.

The procedure underlying Theorem 1 basically extends the Nelson-Oppen combination method [16] to theories over non disjoint signatures, thus lifting the decidability of the constraint satisfiability problem from the component theories to their union.

The requirement (2a) of T_0-*compatibility* over the theories T_1 and T_2 means that there is a Σ_0-theory T_0^* such that (i) $T_0 \subseteq T_0^\star$; (ii) $T_0^\star$ has quantifier elimination; (iii) every Σ_0-constraint which is satisfiable in a model of T_0 is satisfiable also in a model of $T_0^\star$; and (iv) every Σ_i-constraint which is satisfiable in a model of T_i is satisfiable also in a model of $T_0^\star \cup T_i$, for $i = 1, 2$. This requirement guarantees the completeness of the combination procedure underlying Theorem 1 and generalizes the stable infiniteness requirement used for the completeness of the original Nelson-Oppen combination procedure.

The requirement (2b) on T_1, T_2 of being effectively Noetherian extensions of T_0 means the following: first of all (i) T_0 is *Noetherian*, i.e., for every *finite* set of free constants $\underline{a}$, every infinite ascending chain $\Theta_1 \subseteq \Theta_2 \subseteq \cdots \subseteq \Theta_n \subseteq \cdots$ of sets of ground $\Sigma_0^{\underline{a}}$-atoms is eventually constant modulo T_0, i.e. there is a Θ_n in the chain such that $T_0 \cup \Theta_n \models \Theta_m$, for every natural number m. Moreover, we require to be capable to (ii) compute T_0-*bases* for both T_1 and T_2, meaning

that, given a finite set Γ_i of ground clauses (built out of symbols from Σ_i and possibly further free constants) and a finite set of free constants $\underline{a}$, we can always compute a finite set Δ_i of *positive* ground $\Sigma_0^{\underline{a}}$-clauses such that *(a)* $T_i \cup \Gamma_i \models C$, for all $C \in \Delta_i$ and *(b)* if $T_i \cup \Gamma_i \models D$ then $T_0 \cup \Delta_i \models D$, for every positive ground $\Sigma_0^{\underline{a}}$-clause D $(i = 1, 2)$. Note that if Γ_i is T_i-unsatisfiable then w.l.o.g. $\Delta_i = \{\Box\}$. Intuitively, the Noetherianity of T_0 means that, fixed a finite set of constants, there exists only a finite number of atoms that are not redundant when reasoning modulo T_0; on the other hand, the capability of computing T_0-bases means that, for every set Γ_i of ground $\Sigma_i^{\underline{a}}$-literals, it is possible to compute a finite "complete set" of logical consequences of Γ_i over Σ_0; these consequences over the shared signature are exchanged between the satisfiability procedures of T_1 and T_2 in the loop of the combination procedure à la Nelson-Oppen, whose termination is ensured by the Noetherianity of T_0.

The combination procedure is depicted below, where Γ_i denotes a set of ground literals built out of symbols of Σ_i (for $i = 1, 2$), a set of shared free constants $\underline{a}$ and possibly further free constants.

Algorithm 1. Extending Nelson-Oppen

1. If $T_0\text{-}\mathsf{basis}_{T_i}(\Gamma_i) = \Delta_i$ and $\Box \notin \Delta_i$ for each $i \in \{1, 2\}$, then
  1.1. For each $D \in \Delta_i$ such that $T_j \cup \Gamma_j \not\models D$, $(i \neq j)$, add D to Γ_j
  1.2. If Γ_1 or Γ_2 has been changed in **1.1**, then rerun **1.**
  Else **return** *"unsatisfiable"*
2. If this step is reached, **return** *"satisfiable"*.

In what follows we see how to apply this combination algorithm in order to show the decidability of the constraint satisfiability problem for the union of theories that share the theory of abelian groups, denoted from now on by AG. To this aim, we first show that AG is Noetherian (Section 3.2). Second, we exhibit a theory $AG^* \supseteq AG$ that admits quantifier-elimination and whose behaviour w.r.t. the satisfiability of constraints is the same of AG (Section 3.3). Third, we see how to construct effectively Noetherian extensions of AG by using a superposition calculus (Section 5.1).

3 The Theory of Abelian Groups

In this section we focus on some properties that are particularly useful when trying to apply Theorem 1 to a combination of theories sharing AG.

$\boxed{AG}$ rules the behaviour of the binary function symbol $+$, of the unary function symbol $-$ and of the constant 0. More precisely, $\Sigma_{AG} := \{0 : \text{AG}, - : \text{AG} \to \text{AG}, + : \text{AG} \times \text{AG} \to \text{AG}\}$, and AG is axiomatized as follows:

$$\forall x, y, z \;\; (x + y) + z = x + (y + z) \qquad \forall x, y \;\; x + y = y + x$$
$$\forall x \;\; x + 0 = x \qquad \forall x \;\; x + (-x) = 0$$

From now on, given an expansion of Σ_{AG}, a generic term of sort AG will be written as $n_1 t_1 + \cdots + n_k t_k$, where t_i is a term whose root symbol is different both from $+$ and $-$, $t_1 - t_2$ is a shortening for $t_1 + (-t_2)$, and $n_i t_i$ is a shortening for $t_i + \cdots + t_i$ (n_i)-times if n_i is a positive integer, or $-t_i - \cdots - t_i$ $(-n_i)$-times if n_i is negative.

3.1 Unification in Abelian Groups

We will consider a superposition calculus using unification in AG with free symbols, which is known to be finitary [6]. In the following, we restrict ourselves to particular AG-unification problems with free symbols in which no variables of sort AG occur. By using a straightforward many-sorted extension of the Baader-Schulz combination procedure [3], one can show that an AG-equality checker is sufficient to construct a complete set of unifiers for these particular AG-unification problems with free symbols. Moreover, the following holds:

Lemma 1. *Let Γ be a AG-unification problem with free symbols in which no variable of sort* AG *occurs, and let $CSU_{AG}(\Gamma)$ be a complete set of AG-unifiers of Γ. For any $\mu \in CSU_{AG}(\Gamma)$, we have that* 1.) *$VRan(\mu) \subseteq Var(\Gamma)$, and that,* 2.) *for any AG-unifier σ of Γ such that $Dom(\sigma) = Var(\Gamma)$, there exists $\mu \in CSU_{AG}(\Gamma)$ such that $\sigma =_{AG} \mu(\sigma_{|VRan(\mu)})$.*

3.2 Noetherianity of Abelian Groups

Let us start by proving the Noetherianity of AG; the problem of discovering effective Noetherian extensions of AG will be addressed in Section 5.1, after the introduction of an appropriate superposition calculus (Section 4).

Proposition 1. *AG is Noetherian.*

Proof. Note that any equation is AG-equivalent to $(\sharp)$ $\sum_{i=1}^{k} n_i a_i = \sum_{j=1}^{h} m_j b_j$, where a_i, b_j are free constants in $\underline{a} \cup \underline{b}$ and n_i, m_j are positive integers, so we can restrict ourselves to chains of sets of equations of the kind $(\sharp)$. Theorem 3.11 in [7] shows that AC is Noetherian, where AC is the theory of an associative and commutative $+$ (thus $\Sigma_{AC} = \{+\}$). From the definition of Noetherianity it follows that, if T is a Noetherian Σ-theory, any other Σ-theory T' such that $T \subseteq T'$ is Noetherian, too. Clearly, the set of sentences over Σ_{AC} implied by AG extends AC; hence any ascending chain of sets of equations of the kind $(\sharp)$ is eventually constant modulo AG, too.

In order to apply Theorem 1 to a combination of theories that share AG, we need to find an extension of AG that admits quantifier elimination and such that any constraint is satisfiable w.r.t. such an extension iff it is already satisfiable w.r.t. AG. A first, natural candidate would be AG itself. Unfortunately it is not the case: more precisely, it is known that AG cannot admit quantifier elimination (Theorem A.1.4 in [13]). On the other hand, it is possible to find an extension AG^* with the required properties: AG^* is the theory of divisible abelian groups with infinitely many elements of each finite order.

3.3 An Extension of Abelian Groups Having Quantifier Elimination

Let $D_n := \forall x \exists y\, ny = x$, let $O_n(x) := nx = 0$ and let $L_{m,n} := \exists y_1, y_2, \ldots, y_m \bigwedge_{i \neq j} y_i \neq y_j \wedge \bigwedge_{i=1}^{m} O_n(y_i)$, for $n, m \in \mathbb{N}$. D_n expresses the fact that each element is divisible by n, $O_n(x)$ expresses that the element x is of order n, and $L_{m,n}$ expresses the fact that there exist at least m elements of order n. The theory AG^* of divisible abelian groups with infinitely many elements of each finite order can be thus axiomatized by $AG \cup \{D_n\}_{n>1} \cup \{L_{m,n}\}_{m>0,n>1}$.

Now, instead of showing directly that AG^* admits quantifier elimination and satisfies exactly the same constraints that are satisfiable w.r.t. AG, we rely on a different approach. Let us start by introducing some more notions about structures and their properties. Given a Σ-structure $\mathcal{M} = (M, \mathcal{I})$, let Σ^M be the signature where we add to Σ constant symbols m for each element of M. The *diagram* $\Delta(\mathcal{M})$ of $\mathcal{M}$ is the set of all the ground Σ^M-literals that are true in $\mathcal{M}$. Given two Σ-structures $\mathcal{M} = (M, \mathcal{I})$ and $\mathcal{N} = (N, \mathcal{J})$, a *$\Sigma$-embedding* (or, simply, an embedding) between $\mathcal{M}$ and $\mathcal{N}$ is a mapping $\mu : M \to N$ among the corresponding support sets satisfying, for all the Σ^M-atoms ψ, the condition $\mathcal{M} \models \psi$ iff $\mathcal{N} \models \psi$ (here $\mathcal{M}$ is regarded as a Σ^M-structure, by interpreting each additional constant $a \in M$ into itself, and $\mathcal{N}$ is regarded as a Σ^M-structure by interpreting each additional constant $a \in M$ into $\mu(a)$). If $M \subseteq N$ and if the embedding $\mu : M \to N$ is just the identity inclusion $M \subseteq N$, we say that $\mathcal{M}$ is a *substructure* of $\mathcal{N}$. If it happens that, given three models of T: $\mathcal{A}$, $\mathcal{M}$, $\mathcal{N}$ and two embeddings $f : \mathcal{A} \to \mathcal{M}$ and $g : \mathcal{A} \to \mathcal{N}$, there always exists another model of T, $\mathcal{H}$, and two embeddings $h : \mathcal{M} \to \mathcal{H}$ and $k : \mathcal{N} \to \mathcal{H}$ such that the composition $f \circ h = g \circ k$, we say that T has the *amalgamation property.* Finally if, given a Σ-theory T and a model $\mathcal{M}$ for T, it happens that, for each Σ-sentence ψ, $\mathcal{M} \models \psi$ if and only if $T \models \psi$, then we say that T is a *complete* theory.

Now, in [13], Exercise 8 page 380, it is stated that AG^* is the so-called *model companion* of the theory AG, meaning that *(i)* for each model $\mathcal{M}$ of AG^* the theory $AG^* \cup \Delta(\mathcal{M})$ is a complete theory, *(ii)* every constraint that is satisfiable in a model of AG is satisfiable in a model of AG^* and *(iii)* every constraint that is satisfiable in a model of AG^* is satisfiable in a model of AG (of course, since $AG \subset AG^*$, condition *(iii)* gets trivial, but we report here for sake of completeness). At this point, since the behaviour of AG and AG^* is the same w.r.t. the satisfiability of constraints, the only condition that remains to be verified is that AG^* admits quantifier elimination. But:

Theorem 2. *AG has the amalgamation property.*

Corollary 1. *AG^* admits quantifier elimination.*

Proof. In [10] it is shown that, if T is a universal theory and T^* is a model-companion of T, then the following are equivalent:

(i) T^* has quantifier elimination;
(ii) T has the amalgamation property.

Since AG has the amalgamation property, and AG^* is the model-companion of AG, we have that AG^* has quantifier elimination.

4 A Calculus for Abelian Groups

In [12] the authors give a superposition calculus in which the reasoning about elements of an abelian group is completely built-in. Our aim is to elaborate that calculus so that it provides a decision procedure for the satisfiability problem modulo theories modelling interesting data structures and extending AG. More precisely, we want to produce a calculus able to check the satisfiability, in the models of AG, of clauses in the shape $Ax(T) \cup G$, where $Ax(T)$ is a set of unit clauses, not necessarily ground, formalizing the behaviour of some data structure, and G is a set of ground literals. To that purpose, we eliminate the constraints from the calculus and we use a many-sorted language that extends the signature of the theory of abelian groups Σ_{AG} by additional function symbols. Moreover, we will adopt from now on the following assumption: we will consider only

$$\text{unit clauses with no occurrence of variables of sort AG.} \qquad (*)$$

Let us start to see more in detail the notations and the concepts used in the rules of the calculus.

First of all, we will reason over terms modulo an AG-rewriting system: quoting [12], the system R_{AG} consists of the rules $(i)\ x + 0 \to 0$, $(ii)\ -x + x \to 0$, $(iii)\ -(-x) \to 0$, $(iv)\ -0 \to 0$, $(v)\ -(x+y) \to (-x)+(-y)$. Moreover, rewriting w.r.t. R_{AG} is considered *modulo AC*, namely the associativity and the commutativity of the $+$, thus, when rewriting $\to_{R_{AG}}$, we mean the relation $=_{AC} \to_{R_{AG}} =_{AC}$. The normal form of a term t w.r.t. R_{AG} will be often written as AG-nf(t), and two terms t_1 and t_2 are equal modulo AG iff AG-nf$(t_1) =_{AC} AG$-nf(t_2). Accordingly, we say that a substitution σ is in AG-normal form whenever all the terms occurring in the codomain of σ are in AG-normal form.

Moreover, we will consider an order $\succ$ over terms that is total, well-founded, strict on ground terms and such that 1. $\succ$ is AC-compatible, meaning that $s' =_{AC} s \succ t =_{AC} t'$ implies $s' \succ t'$, 2. $\succ$ orients all the rules of R_{AG}, meaning that $l\sigma \succ r\sigma$ for every rule $l \to r$ of R_{AG} and all the grounding substitutions σ; 3. $\succ$ is monotonic on ground terms, meaning that for all ground terms s, t, u, $u[s]_p \succ u[t]_p$ whenever $s \succ t$. An ordering satisfying all the requirements above can be easily obtained considering an RPO ordering with a total precedence $\succ_\Sigma$ on the symbols of the signature Σ such that $f \succ_\Sigma - \succ_\Sigma + \succ_\Sigma 0$ for all symbols f in Σ and such that all the symbols have a lexicographic status, except $+$, whose status is multiset (see [9], where, in order to compare two terms, the arity of $+$ is considered variable, but always greater than 1).

As a last convention, with a little abuse of notation, we will call *summand* any term whose root symbol is different from both $+$ and $-$, notwithstanding its sort. In this way a *generic* term can be written in the shape $n_1t_1 + \cdots + n_kt_k$ (if it is of sort different from AG, it simply boils down to t_1).

Now, we are ready to describe the calculus. We will rely basically on three rules, *Direct AG-superposition, Inverse AG-superposition* and *Reflection*, and, as in [12], we will apply the rules only in case the premises satisfy certain conditions as explained in the following. Moreover, from now on we assume that all the

literals will be eagerly maintained in AG-normal form, meaning that they will be maintained as (dis)equations between terms in AG-normal form.

Orientation for the left premises of direct AG-superposition. Let $l = r$ be an equation; if it is on the sort AG, then it can be equivalently rewritten into $e = 0$. Thus the term e is a term of the form $n_1t_1 + n_2t_2 + \cdots + n_pt_p$, where the t_i are non variable distinct summands, and the n_i's are non zero integers. By splitting the summands into two disjoint sets, the equation $e = 0$ can be rewritten as $n_1t_1 + \cdots + n_kt_k = -n_{k+1}t_{k+1} - \cdots - n_pt_p$. In the following, we will call any equation over AG in that form an *orientation* for $e = 0$. If $l = r$ is an equation over a sort different from AG, then an *orientation* of $l = r$ will be either $l = r$ or $r = l$.

Orientation for the left premises of inverse AG-superposition. Let $e = 0$ be an equation over the sort AG. If e or $-e$ is a term of the form $s + e'$, where s is a summand that occurs positively and e' is a generic term, then $-s = e'$ is an *inverse orientation* for $e = 0$.

Splitting of the right premises for direct AG-superposition. Let t be a non-variable subterm of either r or s in the literal $r \bowtie s$; moreover, if s is of sort AG, we can freely assume that s is 0. If t is of sort AG, we ask that t is not immediately under $+$ nor under $-$, and that the root of t is different from $-$. Thus, we can imagine that t is of the kind $n_1s_1 + \cdots + n_ps_p + t'$, where all s_i are distinct summands, all n_i are positive integers and t' contains only negative summands. In this case, $t_1 + t_2$ is a *splitting* for t if t_1 is a term of the form $k_1s_1 + \cdots + k_ps_p$, where $0 \leq k_i \leq n_i$, and t_2 is $(n_1 - k_1)s_1 + \cdots + (n_p - k_p)s_p + t'$. If t is not over the sort AG, then the only splitting admissible for t is t itself.

Splitting of the right premises for inverse AG-superposition. Let t be a non variable subterm of either r or s in the literal $r \bowtie s$; moreover, if s is of sort AG, we can freely assume that s is 0. Let t be of sort AG, and let t be not immediately below $+$ nor $-$. If t is of the form $-s + t'$, where s is a summand, then $t_1 + t_2$ is an *inverse splitting* for t if t_1 is $-s$ and t_2 is t'.

AG-superposition rules. In the left premise $l = r$ of the direct AG-superposition rule, it is assumed that $l = r$ is an orientation of the literal. Similarly, in the right premise, $D[t_1+t_2]_p$ denotes that $D_{|_p}$ is a non-variable term that is not immediately below $+$ or $-$ with a splitting $t_1 + t_2$. Similarly, in the inverse AG-superposition rule, $l = r$ and $D_{|_p}$ denote inverse orientation and splitting, respectively. The inference system, denoted by $\mathcal{SP}_{AG}$, is made of the following rules:

$$\textit{Direct AG-superposition}\ \frac{l = r \qquad D[t_1 + t_2]_p}{(D[r + t_2]_p)\mu_i}\ (i)$$

$$\textit{Inverse AG-superposition}\ \frac{l = r \qquad D[t_1 + t_2]_p}{(D[r + t_2]_p)\mu_i}\ (ii)$$

$$\textit{Reflection}\ \frac{u' \neq u}{\Box}\ (iii)$$

The condition (i) is that μ_i is a most general solution of the AG-unification problem $l =_{AG} t_1$; moreover the inference has to be performed whenever there is a ground instantiation of μ_i, θ, s.t., if $nu = s$ is the AG-normal form of $(l = r)\mu_i\theta$ and $D'[nu]_q$ is the AG-normal form of $(D[t_1 + t_2]_p)\mu_i\theta$ in which, in position q, nu appears as subterm, then (a) $u \succ s$, (b) nu appears as subterm of the maximal term in D'.

The condition (ii) is that μ_i is a most general solution of the AG-unification problem $l =_{AG} t_1$; moreover the inference has to be performed whenever there is a ground instantiation of μ_i, θ, s.t., if $-u = s$ is the AG-normal form of $(l = r)\mu_i\theta$ and $D'[-u]_q$ is the AG-normal form of $(D[t_1 + t_2]_p)\mu_i\theta$ in which, in position q, $-u$ appears as subterm, then (a) either u is the maximal summand in s or $u \succ s$, (b) $-u$ appears as subterm of the maximal term in D'.

The condition (iii) is that the AG-unification problem $u =_{AG} u'$ has a solution (and $\Box$ is the syntactic convention for the empty clause).

Moreover, we assume that, after each inference step, the newly-derived literal is normalized modulo AG.

We point out that, thanks to Lemma 1(1.) and to our assumption $(*)$, at any step of a saturation no variable of sort AG is introduced, thus the resulting saturated set will consist of literals in which no variable of sort AG occurs. Moreover, we can note that the conditions on the inferences are, in general, far from being obvious to check. However, for our purposes, we will often perform inferences involving at least one ground literal. In that case, verifying all the conditions becomes easier.

5 Refutational Completeness of $\mathcal{SP}_{AG}$

In order to prove the refutational completeness of the calculus presented above, we will adapt the model generation technique presented in [12]. The idea behind this technique consists in associating to any saturated set of literals that does not contain the empty clause a model of terms identified modulo a rewriting system, the latter being built according to some of the equations in the saturated set. Even if in our calculus no constrained literal will appear, in order to build the model of terms we will rely only on ground instances of the literals in the saturation that are *irreducible*. Moving from [12] and extending to the many-sorted case, we say that:

Definition 1. *An equation $s = t$ is in* one-sided form *whenever, (a) if s and t are of sort* AG*, the equation is in the form $e = 0$, and e is in AG-normal form; (b) if s and t are not of sort* AG*, both s and t are in AG-normal form.*

Whereas an equation over a sort different from AG has a unique one-sided form, an equation over the sort AG has two AG-equivalent one-sided forms, but in what follows it does not matter which of the two will be considered. Thus, from now on, when we will refer to equations, we will always assume that the equations are in one-sided form.

Definition 2. *Let s be a term, σ be a grounding substitution such that both σ and s are in AG-normal form. Moreover, let R be a ground term rewriting system. We will say that the $maxred_R(s\sigma)$ is*

- *0, if AG-nf($s\sigma$) is R-irreducible;*
- *$\max PS$, where PS is the following set of terms (ordered w.r.t. $\succ$):*
  $PS := \{u$ is a summand $|$ for some term v and some n in $\mathbb{Z}$, AG-nf($s\sigma$) is of the form $nu + v$ and nu is R-reducible$\}$.

Definition 3. [1] *Let s be a term in which no variable of sort* AG *occurs, let σ be a grounding substitution such that both s and σ are in AG-normal form, and let R be a ground TRS. The pair (s, σ) is* irreducible *w.r.t. R whenever:*

- *AG-nf($s\sigma$) is R-irreducible, or*
- *if AG-nf($s\sigma$) is R-reducible, let u be the $maxred_R(s\sigma)$. Then, (s, σ) is irreducible if s is not a variable and, for each term of the form $t = f(t_1, \ldots, t_n)$ such that s is of the form $t + v$ or $-t + v$ or t and such that $u \succeq$ AG-nf($t\sigma$), each (t_i, σ) is irreducible.*

If L is a literal, the pair (L, σ) is irreducible w.r.t. R:

- *if L is an (dis)equation whose one-sided form is of the form $e \bowtie 0$, then (e, σ) is irreducible w.r.t. R;*
- *if L is an (dis)equation whose one-sided form is of the form $s \bowtie t$, both (s, σ) and (t, σ) are irreducible w.r.t. R.*

Before going on with the description of all the ingredients that are needed in order to show the completeness of the calculus, we want to point out a property that will be useful in the following.

Proposition 2. *Let s be a term in which no variable of sort* AG *occurs, let σ be a grounding substitution such that both s and σ are in AG-normal form, and let R be a ground TRS such that (s, σ) is irreducible w.r.t. R. Moreover, let $\sigma =_{AG} \mu\pi$, where π is another grounding substitution in AG-normal form and μ is a substitution that does not have variables of sort* AG *in its range. Then $(s\mu, \pi)$ is still irreducible w.r.t. R.*

To extract, from a given set of ground literals, a term rewriting system, we first of all transform all the equations in reductive normal form (see [12]):

Definition 4. *A ground literal $s \bowtie t$ in AG-normal form is in* reductive form *whenever s is of the form nu, t is the form $n_1v_1 + \cdots + n_kv_k$ and $n > 0$, n_i are non-zero integers, u and v_i are summands with $u \succ v_i$.*

Of course, if s and t are of sort different from AG, the definition above simply says that $s \succ t$; moreover, it is always possible, given an equation, to obtain an equivalent one in reductive normal form. Now, a term rewriting system is obtained as follows:

[1] Here we are adapting, in case of absence of variables of sort AG, the definition of recursive irreducibility of [12], but in our context the two notions of recursive irreducibility and irreducibility are collapsing.

Definition 5. *Let S be a set of literals, let L be an equation with a ground instance $L\sigma$, let G be the reductive form of $L\sigma$: $G \equiv nu = r$. Then G* generates *the rule $nu \to r$ if the following conditions are satisfied:*

(i) $(R_G \cup AG) \not\models G$;
(ii) $u \succ r$;
(iii) nu *is* R_G*-irreducible;*
(iv) (L, σ) *is irreducible w.r.t.* R_G.

where R_G is the set of rules generated by the reductive forms of the ground instances of S that are smaller than G w.r.t. $\succ$. Moreover, if $n > 1$, then also the rule $-u \to (n-1)u - r$ is generated.

Now, exactly as in [12], we associate to a generic set of literals saturated under the rules of our calculus and that does not contain the empty clause, S, a structure I that is an AG-model for S. I is the equality Herbrand interpretation defined as the congruence on ground terms generated by $R_S \cup AG$, where R_S is the set of rules generated by S according to Definition 5. Since we are in a many-sorted context, the domain of I consists of different sets, one for each sort; since the rewriting rules in $R_S \cup AG$ are sort-preserving, the congruence on the ground terms is well-defined. Applying the same kind of arguments used to prove Lemma 10 in [12], we have that $R_S \cup AG$ is terminating and confluent, and it still holds that $I \models s = t$ iff $s \to^*_{R_S \cup R_{AG}} \tau \leftarrow^*_{R_S \cup R_{AG}} t$ for some term τ. To show that I is really an AG-model for S, we can prove the following lemma:

Lemma 2. *Let S be the closure under the calculus of a set of literals S_0, and let us assume that the empty clause does not belong to S. Let I be the model of terms derived from S as described above, and let $Ir_{R_S}(S)$ be the set of ground instances $L\sigma$ of L in S such that (L, σ) is irreducible w.r.t. R_S. Then (1) $I \models Ir_{R_S}(S)$ implies that $I \models S$, and (2) $I \models Ir_{R_S}(S)$.*

From the lemma above, it follows immediately:

Theorem 3. *The calculus $\mathcal{SP}_{AG}$ is refutational complete for any set of literals that do not contain variables of sort* AG.

5.1 Computing *AG*-Bases

Let us go back, for the moment, to Theorem 1, and especially to condition (2b) that states that, in order to apply a combination procedure à la Nelson-Oppen to a pair of theories T_1 and T_2 sharing AG, we have to ensure that T_1 and T_2 are effectively Noetherian extensions of AG, i.e. we have to ensure the capability of computing AG-bases for T_1 and T_2. Let us suppose that T_i is a Σ_i-theory (for $i = 1, 2$) whose set of axioms is described by a finite number of unit clauses.

Now, for $i = 1, 2$, let Γ_i be a set of ground literals over an expansion of $\Sigma_i \supseteq \Sigma_{AG}$ with the finite sets of fresh constants $\underline{a}, \underline{b}_i$, and suppose to perform a saturation w.r.t. $\mathcal{SP}_{AG}$ adopting an RPO ordering in which the precedence is $f \succ a \succ - \succ + \succ 0$ for every function symbol f in $\Sigma_i^{\underline{b}_i}$ different from $+, -, 0$,

every constant a in $\underline{a}$ and that all the symbols have a lexicographic status, except $+$, whose status is multiset. Relying on the refutational completeness of $\mathcal{SP}_{AG}$, Proposition 3 shows how $\mathcal{SP}_{AG}$ can be used in order to ensure that T_1 and T_2 are effectively Noetherian extensions of AG:

Proposition 3. *Let S be a finite saturation of $T_i \cup \Gamma_i$ w.r.t $\mathcal{SP}_{AG}$ not containing the empty clause and suppose that, in every equation $e = 0$ containing at least one of the constants a in $\underline{a}$ as summand, the maximal summand is not unifiable with any other summand in e. Then the set Δ_i of all the ground equations over $\Sigma_{AG}^{\underline{a}}$ in S is an AG-basis for T_i w.r.t. $\underline{a}$ $(i = 1, 2)$.*

6 Some Examples

Theorem 3 guarantees that $\mathcal{SP}_{AG}$ is refutational complete, thus, if we want to turn it into a decision procedure for the constraint satisfiability problem w.r.t. a theory of the kind $T \cup AG$, it is sufficient to prove that any saturation under the rules of $\mathcal{SP}_{AG}$ of a set of ground literals and the axioms of T is finite. Let us show some examples in which this is actually the case.

Lists with Length. The theory of lists with length can be seen as the union of the theories $T_L \cup T_\ell \cup AG$, with T_L being the theory of lists and T_ℓ being the theory that axiomatizes the behaviour of the function for the length; more formally:

$\boxed{T_L}$ has the many-sorted signature of the theory of lists: Σ_L is the set of function symbols $\{\mathsf{nil} : \text{LISTS}, \mathsf{car} : \text{LISTS} \to \text{ELEM}, \mathsf{cdr} : \text{LISTS} \to \text{LISTS}, \mathsf{cons} : \text{ELEM} \times \text{LISTS} \to \text{LISTS}\}$ plus the predicate symbol $\mathsf{atom} : \text{LISTS}$, and it is axiomatized as follows:

$$\begin{array}{ll} & \forall x\ \neg\mathsf{atom}(x) \Rightarrow \mathsf{cons}(\mathsf{car}(x), \mathsf{cdr}(x)) = x \\ \forall x, y\ \mathsf{car}(\mathsf{cons}(x, y)) = x & \forall x, y\ \neg\mathsf{atom}(\mathsf{cons}(x, y)) \\ \forall x, y\ \mathsf{cdr}(\mathsf{cons}(x, y)) = y & \mathsf{atom}(\mathsf{nil}) \end{array}$$

$\boxed{T_\ell}$ is the theory that gives the axioms for the function length $\ell : \text{LISTS} \to \text{AG}$ and the constant $(1 : \text{AG})$: $\ell(\mathsf{nil}) = 0$; $\forall x, y\ \ell(\mathsf{cons}(x, y)) = \ell(y) + 1$; $1 \neq 0$

Applying some standard reasoning (see, e.g. [18]), we can substitute T_L with the set of the purely equational axioms of T_L, say $T_{L'}$, and enrich a bit the set of literals G to a set of literals G' in such a way $T_L \cup T_\ell \cup AG \cup G$ is equisatisfiable to $T_{L'} \cup T_\ell \cup AG \cup G'$. Let us choose as ordering an RPO with a total precedence $\succ$ such that all the symbols have a lexicographic status, except $+$, whose status is multiset, and such that it respects the following requirements: (a) $\mathsf{cons} \succ \mathsf{cdr} \succ \mathsf{car} \succ c \succ e \succ \ell$ for every constant c of sort LISTS and every constant e of sort ELEM; (b) $\ell \succ g \succ - \succ + \succ 0$ for every constant g of sort AG.

Proposition 4. *For any set G of ground literals, any saturation of $T_{L'} \cup T_\ell \cup G'$ w.r.t. $\mathcal{SP}_{AG}$ is finite.*

Trees with Size. Let us reason about trees and their size. We can propose a formalization in which we need to reason about a theory of the kind $T_T \cup T_{size} \cup AG$, where T_T rules the behaviour of the trees and T_{size} constraints the behaviour of a function that returns the number of nodes of a tree. Thus we have:

$\boxed{T_T}$ has the mono-sorted signature $\Sigma_T := \{\mathcal{E} : \text{TREES}, \mathsf{binL} : \text{TREES} \to \text{TREES}, \mathsf{binR} : \text{TREES} \to \text{TREES}, \mathsf{bin} : \text{TREES} \times \text{TREES} \to \text{TREES}\}$, and it is axiomatized as follows:

$$\forall x, y\ \mathsf{binL}(\mathsf{bin}(x, y)) = x \qquad \forall x, y\ \mathsf{binR}(\mathsf{bin}(x, y)) = y$$
$$\forall x\ \mathsf{bin}(\mathsf{binL}(x), \mathsf{binR}(x)) = x$$

$\boxed{T_{size}}$ is the theory that gives the axioms for the function $\mathsf{size} : \text{TREES} \to \text{AG}$: $\mathsf{size}(\mathcal{E}) = 0;\ \ \forall x, y\ \mathsf{size}(\mathsf{bin}(x, y)) = \mathsf{size}(x) + \mathsf{size}(y)$

Let us now put as ordering an RPO with a total precedence $\succ$ on the symbols of the signature such that all the symbols have a lexicographic status, except $+$, whose status is multiset, and such that it respects the following requirements: (a) $\mathsf{bin} \succ \mathsf{binR} \succ \mathsf{binL} \succ c \succ \mathsf{size}$ for every constant c of sort TREES; (b) $\mathsf{size} \succ g \succ - \succ + \succ 0$ for every constant g of sort AG.

Proposition 5. *For any set G of ground literals, any saturation of $T_T \cup T_{size} \cup G$ w.r.t. $\mathcal{SP}_{AG}$ is finite.*

Application *(Algorithm 2.8 in [25]: Left-Rotation of trees)* Using the procedure induced by the calculus $\mathcal{SP}_{AG}$, it is possible to verify, e.g. that the input tree x and the output tree y have the same size:

1. $t := x$; 2. $y := \mathsf{binR}(t)$; 3. $\mathsf{binR}(t) := \mathsf{binL}(y)$; 4. $\mathsf{binL}(y) := t$; 5. Return y

In order to check that the size of x is exactly the one of y, we check for unsatisfiability modulo $T_T \cup T_{size} \cup AG$ the following constraint (see, again [25]):

$$\mathsf{binR}(t') = \mathsf{binL}(\mathsf{binR}(x')) \wedge \mathsf{binL}(t') = \mathsf{binL}(x') \wedge \mathsf{binL}(y') = t'$$
$$\wedge\, \mathsf{binR}(y') = \mathsf{binR}(\mathsf{binR}(x')) \wedge \mathsf{size}(x') \neq \mathsf{size}(y')$$

where x', y' and t' are fresh constants that identify the trees on which the algorithm applies.

6.1 Applying the Combination Framework

In the section above we have shown some examples of theories that extend the theory of abelian groups and whose constraint satisfiability problem is decidable. We have proved that AG can be enlarged to AG^* and AG and AG^* behave the same w.r.t. the satisfiability of constraints; moreover we have checked that AG is a Noetherian theory. To guarantee now that the theories that have been studied can be combined together it is sufficient to show that they fully satisfy the requirement of being AG-compatible and effectively Noetherian extension of AG (requirements (2a) and (2b) of Theorem 1). The AG-compatibility both of lists with length and trees with size is easily ensured observing that a constraint is

satisfied w.r.t. $T_L \cup T_\ell \cup AG$ iff it is satisfied w.r.t. $T_L \cup T_\ell \cup AG^*$ and, analogously, any constraint is satisfiable w.r.t. $T_T \cup T_{size} \cup AG$ iff it is w.r.t. $T_T \cup T_{size} \cup AG^*$.

Moreover, checking the shape of the saturations produced, it is immediate to see that all the hypotheses required by Proposition 3 are satisfied when considering both the cases of lists with length and trees with size, turning $\mathcal{SP}_{AG}$ not only into a decision procedure for the constraint satisfiability problem, but also into an effective method for deriving complete sets of logical consequences over the signature of abelian groups (namely, the AG-bases). This implies that also the requirement (2b) of being effectively Noetherian extensions of abelian groups is fulfilled for both lists with length and trees with size. To sum up, we have proved that the theories presented so far can be combined preserving the decidability of the constraint satisfiability problem.

7 Conclusion

The problem of integrating a reasoning modulo arithmetic properties into the superposition calculus has been variously studied, and different solutions have been proposed, both giving the possibility of reasoning modulo the linear rational arithmetic ([14]) and relying on an over-approximation of arithmetic via abelian groups ([12,22]) or divisible abelian groups ([23,24]).

We have focused on the second kind of approach, giving an original solution to the satisfiability problem in combinations of theories sharing the theory of abelian groups. We have shown that in this case all the requirements to apply the non-disjoint combination method are satisfied, and we have considered an appropriate superposition calculus modulo abelian groups in order to derive satisfiability procedures. This calculus relies on a non trivial adaptation the one proposed in [12]: We consider a many-sorted and constraint-free version of the calculus, in which we use a restricted form of unification in abelian groups with free symbols, and in which only literals are involved. Under these assumptions we have proved that the calculus is refutationally complete, but, as a side remark, we notice that the same kind of proof works also in case the rules are extended to deal with Horn clauses and also, exactly as it happens in [12], after the introduction of an appropriate rule for the Factoring, to deal with general clauses. Our focus on the unit clause case is justified by our interest in the application to particular theories whose formalization is actually through axioms of that form.

It is worth noticing that two combination methods are involved in our approach: the method for unification problems [3] and the non-disjoint extension of Nelson-Oppen for satisfiability problems [11].

The framework for the non-disjoint combination used here cannot be applied, as it is, to the case where we consider a combination of theories sharing the Presburger arithmetic, because the latter is not Noetherian. Another framework, able to guarantee the termination of the resulting procedure on all the inputs, should be designed for that case.

We envision several directions for future work. As a first direction, we would like to relax current restrictions on theories and saturation types to apply effectively the calculus in the non-disjoint combination method. At the moment,

since the presence of variables of sort AG in the clauses is not allowed, the results in [18] are not subsumed by the present paper. That restriction is justified by technical reasons: an important issue would be to discard it, enlarging in this way the applicability of our results. As a second direction we foresee, it would be interesting to find general methods to ensure the termination of the calculus by developing, for instance, an automatic meta-saturation method [15], or by considering a variable-inactivity condition [1]. Finally, it would be interesting to study how our calculus can be integrated into Satisfiability Modulo Theories solvers, by exploiting for instance the general framework developed in [4].

References

1. Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: New results on rewrite-based satisfiability procedures. ACM Transactions on Computational Logic 10(1) (2009)
2. Armando, A., Ranise, S., Rusinowitch, M.: A rewriting approach to satisfiability procedures. Information and Computation 183(2), 140–164 (2003)
3. Baader, F., Schulz, K.U.: Unification in the union of disjoint equational theories: Combining decision procedures. Journal of Symbolic Computation 21(2), 211–243 (1996)
4. Bonacina, M.P., Echenim, M.: T-decision by decomposition. In: Pfenning, F. (ed.) CADE 2007. LNCS, vol. 4603, pp. 199–214. Springer, Heidelberg (2007)
5. Bonacina, M.P., Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Decidability and undecidability results for Nelson-Oppen and rewrite-based decision procedures. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS, vol. 4130, pp. 513–527. Springer, Heidelberg (2006)
6. Boudet, A., Jouannaud, J.-P., Schmidt-Schauß, M.: Unification in boolean rings and abelian groups. In: Kirchner, C. (ed.) Unification, pp. 267–296. Academic Press, London (1990)
7. Chenadec, P.L.: Canonical Forms in Finitely Presented Algebras. Research Notes in Theoretical Computer Science. Pitman-Wiley, Chichester (1986)
8. de Moura, L.M., Bjørner, N.: Engineering DPLL(T) + saturation. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS, vol. 5195, pp. 475–490. Springer, Heidelberg (2008)
9. Dershowitz, N.: Orderings for term-rewriting systems. Theoretical Computer Science 17(3), 279–301 (1982)
10. Eklof, P.C., Sabbagh, G.: Model-completions and modules. Annals of Mathematical Logic 2, 251–295 (1971)
11. Ghilardi, S., Nicolini, E., Zucchelli, D.: A comprehensive combination framework. ACM Transactions on Computational Logic 9(2), 1–54 (2008)
12. Godoy, G., Nieuwenhuis, R.: Superposition with completely built-in abelian groups. Journal of Symbolic Computation 37(1), 1–33 (2004)
13. Hodges, W.: Model Theory. Encyclopedia of Mathematics and its Applications, vol. 42. Cambridge University Press, Cambridge (1993)
14. Korovin, K., Voronkov, A.: Integrating linear arithmetic into superposition calculus. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 223–237. Springer, Heidelberg (2007)
15. Lynch, C., Tran, D.-K.: Automatic Decidability and Combinability Revisited. In: Pfenning, F. (ed.) CADE 2007. LNCS, vol. 4603, pp. 328–344. Springer, Heidelberg (2007)

16. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Transaction on Programming Languages and Systems 1(2), 245–257 (1979)
17. Nicolini, E., Ringeissen, C., Rusinowitch, M.: Combinable Extensions of Abelian Groups. Research Report, INRIA, RR-6920 (2009)
18. Nicolini, E., Ringeissen, C., Rusinowitch, M.: Satisfiability procedures for combination of theories sharing integer offsets. In: TACAS 2009. LNCS, vol. 5505, pp. 428–442. Springer, Heidelberg (2009)
19. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch.7, vol. I, pp. 371–443. Elsevier Science, Amsterdam (2001)
20. Peterson, G.E., Stickel, M.E.: Complete sets of reductions for some equational theories. J. ACM 28(2), 233–264 (1981)
21. Plotkin, G.: Building-in equational theories. Machine Intelligence 7, 73–90 (1972)
22. Stuber, J.: Superposition theorem proving for abelian groups represented as integer modules. Theoretical Computer Science 208(1-2), 149–177 (1998)
23. Waldmann, U.: Superposition and chaining for totally ordered divisible abelian groups. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS, vol. 2083, pp. 226–241. Springer, Heidelberg (2001)
24. Waldmann, U.: Cancellative abelian monoids and related structures in refutational theorem proving (Part I,II). Journal of Symbolic Computation 33(6), 777–829 (2002)
25. Zhang, T.: Arithmetic integration of decision procedures. PhD thesis, Department of Computer Science, Stanford University, Stanford, US (2006)

Locality Results for Certain Extensions of Theories with Bridging Functions

Viorica Sofronie-Stokkermans

Max-Planck-Institut für Informatik, Campus E 1.4, Saarbrücken, Germany

Abstract. We study possibilities of reasoning about extensions of base theories with functions which satisfy certain recursion (or homomorphism) properties. Our focus is on emphasizing possibilities of hierarchical and modular reasoning in such extensions and combinations thereof. We present practical applications in verification and cryptography.

1 Introduction

In this paper we study possibilities of reasoning in extensions of theories with functions which satisfy certain recursion (or homomorphism) axioms. This type of axioms is very important in verification – for instance in situations in which we need to reason about functions defined by certain forms of primitive recursion – and in cryptography, where one may need to model homomorphism axioms of the form $\forall x, y, z(\mathsf{encode}_z(x * y) = \mathsf{encode}_z(x) * \mathsf{encode}_z(y))$. Decision procedures for recursive data structures exist. In [13], Oppen gave a PTIME decision procedure for absolutely free data structures based on bidirectional closure; methods which use rewriting and/or basic equational reasoning were given e.g. by Barrett et al. [2] and Bonacina and Echenim [3]. Some extensions of theories with recursively defined functions and homomorphisms have also been studied. In [1], Armando, Rusinowitch, and Ranise give a decision procedure for a theory of homomorphisms. In [18], Zhang, Manna and Sipma give a decision procedure for the extension of a theory of term structures with a recursively defined length function. In [8] tail recursive definitions are studied. It is proved that tail recursive definitions can be expressed by shallow axioms and therefore define so-called "stably local extensions". Locality properties have also been studied in a series of papers on the analysis of cryptographic protocols (cf. e.g. [4,5,6]).

In this paper we show that many extensions with recursive definitions (or with generalized homomorphism properties) satisfy locality conditions. This allows us to significantly extend existing results on reasoning about functions defined using certain forms of recursion, or satisfying homomorphism properties [1,8,18], and at the same time shows how powerful and widely applicable the concept of local theory (extension) is in automated reasoning. As a by-product, the methods we use provide a possibility of presenting in a different light (and in a different form) locality phenomena studied in cryptography in [4,5,6]; we believe that they will allow to better separate rewriting from proving, and thus to give simpler proofs. The main results are summarized below:

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 67–83, 2009.

- We show that the theory of absolutely free constructors is local, and locality is preserved also in the presence of selectors. These results are consistent with existing decision procedures for this theory [13] which use a variant of bi-directional closure in a graph formed starting from the subterms of the set of clauses whose satisfiability is being checked.
- We show that, under certain assumptions, extensions of the theory of absolutely free constructors with functions satisfying a certain type of recursion axioms satisfy locality properties, and show that for functions with values in an ordered domain we can combine recursive definitions with boundedness axioms without sacrificing locality. We also address the problem of only considering models whose data part is the *initial* term algebra of such theories.
- We analyze conditions which ensure that similar results can be obtained if we relax some assumptions about the absolute freeness of the underlying theory of data types, and illustrate the ideas on an example from cryptography.

The locality results we establish allow us to reduce the task of reasoning about the class of recursive functions we consider to reasoning in the underlying theory of data structures (possibly combined with the theories associated with the co-domains of the recursive functions).

Structure of the paper. In Section 2 we present the results on local theory extensions and hierarchical reasoning in local theory extensions needed in the paper. We start Section 3 by considering theories of absolutely free data structures, and extensions of such theories with selectors. We then consider additional functions defined using a certain type of recursion axioms (possibly having values in a different – e.g. numeric – domain). We show that in these cases locality results can be established. In Section 4 we show that similar results can be obtained if we relax some assumptions about the absolute freeness of the underlying theory of data types, and illustrate the results on a simple example from cryptography.

2 Preliminaries

We will consider theories over possibly many-sorted signatures $\Pi = (S, \Sigma, \mathsf{Pred})$, where S is a set of sorts, Σ a set of function symbols, and Pred a set of predicate symbols. For each function $f \in \Sigma$ (resp. predicate $P \in \mathsf{Pred}$), we denote by $a(f) = s_1, \ldots, s_n \rightarrow s$ (resp. $a(P) = s_1, \ldots, s_n$) its arity, where $s_1, \ldots, s_n, s \in S$, and $n \geq 0$. In the one-sorted case we will simply write $a(f) = n$ (resp. $a(P) = n$).

First-order theories are sets of formulae (closed under logical consequence), typically the set of all consequences of a set of axioms. When referring to a theory, we can also consider the set of all its models. We here consider theories specified by their sets of axioms, but – usually when talking about local extensions of a theory – we will refer to a theory, and mean the set of all its models.

The notion of *local theory* was introduced by Givan and McAllester [9,10]. They studied sets $\mathcal{K}$ of Horn clauses with the property that, for any ground Horn clause C, $\mathcal{K} \models C$ only if already $\mathcal{K}[C] \models C$ (where $\mathcal{K}[C]$ is the set of instances of $\mathcal{K}$ in which all terms are subterms of ground terms in $\mathcal{K}$ or C).

Theory Extensions. We here also consider *extensions of theories*, in which the signature is extended by new *function symbols* (i.e. we assume that the set of predicate symbols remains unchanged in the extension). Let $\mathcal{T}_0$ be an arbitrary theory with signature $\Pi_0 = (S, \Sigma_0, \mathsf{Pred})$. We consider extensions $\mathcal{T}_1$ of $\mathcal{T}_0$ with signature $\Pi = (S, \Sigma, \mathsf{Pred})$, where the set of function symbols is $\Sigma = \Sigma_0 \cup \Sigma_1$. We assume that $\mathcal{T}_1$ is obtained from $\mathcal{T}_0$ by adding a set $\mathcal{K}$ of (universally quantified) clauses in the signature Π.

Partial Models. Let $\Pi = (S, \Sigma, \mathsf{Pred})$. A *partial Π-structure* is a structure $(\{A_s\}_{s\in S}, \{f_A\}_{f\in\Sigma}, \{P_A\}_{P\in\mathsf{Pred}})$ in which for every $f\in\Sigma$, with $a(f)=s_1, \ldots, s_n\to s$, f_A is a (possibly partially defined) function from $A_{s_1} \times \cdots \times A_{s_n}$ to A_s, and for every $P \in \mathsf{Pred}$ with arity $a(P) = s_1 \ldots s_n$, $P_A \subseteq A_{s_1} \times \cdots \times A_{s_n}$. A *weak Π-embedding* between partial structures $A = (\{A_s\}_{s\in S}, \{f_A\}_{f\in\Sigma}, \{P_A\}_{P\in\mathsf{Pred}})$ and $B = (\{B_s\}_{s\in S}, \{f_B\}_{f\in\Sigma}, \{P_B\}_{P\in\mathsf{Pred}})$ is an S-sorted family $i = (i_s)_{s\in S}$ of injective maps $i_s : A_s \to B_s$ which is an embedding w.r.t. Pred, s.t. if $a(f) = s_1, \ldots, s_n \to s$ and $f_A(a_1, \ldots, a_n)$ is defined then $f_B(i_{s_1}(a_1), \ldots, i_{s_n}(a_n))$ is defined and $i_s(f_A(a_1, \ldots, a_n))=f_B(i_{s_1}(a_1), \ldots, i_{s_n}(a_n))$.

We now define truth and satisfiability in partial structures of Π-literals and (sets of) clauses with variables in a set X. If A is a partial structure, $\beta : X \to A$ is a valuation[1] and $L = (\neg)P(t_1, \ldots, t_n)$ is a literal (with $P \in \mathsf{Pred}\cup\{=\}$) we say that $(A, \beta) \models_w L$ if (i) either $\beta(t_i)$ are all defined and $(\neg)P_A(\beta(t_1), \ldots, \beta(t_n))$ is true in A, or (ii) $\beta(t_i)$ is not defined for some argument t_i of P. Weak satisfaction of clauses ($(A, \beta) \models_w C$) is defined in the usual way. A is a *weak partial model* of a set $\mathcal{K}$ of clauses if $(A, \beta)\models_w C$ for every $\beta : X\to A$ and every clause $C \in \mathcal{K}$. A *weak partial model* of $\mathcal{T}_0 \cup \mathcal{K}$ is a weak partial model of $\mathcal{K}$ whose reduct to Π_0 is a total model of $\mathcal{T}_0$.

Local Theory Extensions. Consider the following condition (in what follows we refer to sets G of ground clauses and assume that they are in the signature $\Pi^c = (S, \Sigma \cup \Sigma_c, \mathsf{Pred})$, where Σ_c is a set of new constants):

(Loc) For every finite set G of ground clauses $\mathcal{T}_1\cup G \models \bot$ iff $\mathcal{T}_0\cup\mathcal{K}[G]\cup G$ has no weak partial model with all terms in $\mathsf{st}(\mathcal{K}, G)$ defined

where if T is a set of terms, $\mathcal{K}[T]$ is the set of instances of $\mathcal{K}$ in which all terms starting with a symbol in Σ_1 are in T, and $\mathcal{K}[G] := \mathcal{K}[\mathsf{st}(\mathcal{K}, G)]$, where $\mathsf{st}(\mathcal{K}, G)$ is the family of all subterms of ground terms in $\mathcal{K}$ or G.

We say that an extension $\mathcal{T}_0 \subseteq \mathcal{T}_1$ is *local* if it satisfies condition (Loc). We say that it is *local for clauses with a property P* if it satisfies the locality conditions for all ground clauses G with property P. A more general locality condition (ELoc) refers to situations when $\mathcal{K}$ consists of formulae $(\Phi(x_1, \ldots, x_n) \vee C(x_1, \ldots, x_n))$, where $\Phi(x_1, \ldots, x_n)$ is a *first-order Π_0-formula* with free variables $x_1, \ldots, x_n$, and $C(x_1, \ldots, x_n)$ is a *clause* in the signature Π. The free variables $x_1, \ldots, x_n$ of such an axiom are considered to be universally quantified [14].

[1] We denote the canonical extension to terms of a valuation $\beta : X\to A$ again by β.

(ELoc) For every formula $\Gamma = \Gamma_0 \cup G$, where Γ_0 is a Π_0^c-sentence and G is a finite set of ground Π^c-clauses, $\mathcal{T}_1 \cup \Gamma \models \bot$ iff $\mathcal{T}_0 \cup \mathcal{K}[G] \cup \Gamma$ has no weak partial model in which all terms in $\mathsf{st}(\mathcal{K}, G)$ are defined.

A more general notion, namely Ψ-locality of a theory extension (in which the instances to be considered are described by a closure operation Ψ) is introduced in [11]. Let $\mathcal{K}$ be a set of clauses. Let $\Psi_{\mathcal{K}}$ be a closure operation associating with any set T of ground terms a set $\Psi_{\mathcal{K}}(T)$ of ground terms such that all ground subterms in $\mathcal{K}$ and T are in $\Psi_{\mathcal{K}}(T)$. Let $\Psi_{\mathcal{K}}(G) := \Psi_{\mathcal{K}}(\mathsf{st}(\mathcal{K}, G))$. We say that the extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}$ is Ψ-local if it satisfies:

(Loc^{Ψ}) for every finite set G of ground clauses, $\mathcal{T}_0 \cup \mathcal{K} \cup G \models \bot$ iff $\mathcal{T}_0 \cup \mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G$ has no weak partial model in which all terms in $\Psi_{\mathcal{K}}(G)$ are defined.

(ELoc^{Ψ}) is defined analogously. In (Ψ-)local theories and extensions satisfying (ELoc^{Ψ}), hierarchical reasoning is possible.

Theorem 1 ([14,11]). *Let $\mathcal{K}$ be a set of clauses. Assume that $\mathcal{T}_0 \subseteq \mathcal{T}_1 = \mathcal{T}_0 \cup \mathcal{K}$ is a Ψ-local theory extension, and that for every finite set T of terms $\Psi_{\mathcal{K}}(T)$ is finite. For any set G of ground clauses, let $\mathcal{K}_0 \cup G_0 \cup \mathsf{Def}$ be obtained from $\mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G$ by flattening and purification[2]. Then the following are equivalent:*

(1) G is satisfiable w.r.t. $\mathcal{T}_1$.
(2) $\mathcal{T}_0 \cup \mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G$ has a partial model with all terms in $\mathsf{st}(\mathcal{K}, G)$ defined.
(3) $\mathcal{T}_0 \cup \mathcal{K}_0 \cup G_0 \cup \mathsf{Con}[G]_0$ has a (total) model, where

$$\mathsf{Con}[G]_0 = \{ \bigwedge_{i=1}^{n} c_i = d_i \rightarrow c = d \mid f(c_1, \ldots, c_n) = c, f(d_1, \ldots, d_n) = d \in \mathsf{Def} \}.$$

Theorem 1 allows us to transfer decidability and complexity results from the theory $\mathcal{T}_0$ to the theory $\mathcal{T}_1$:

Theorem 2 ([14]). *Assume that the extension $\mathcal{T}_0 \subseteq \mathcal{T}_1$ satisfies condition (Loc^{Ψ}) – where Ψ has the property that $\Psi(T)$ is finite for every finite T – and that every variable in any clause of $\mathcal{K}$ occurs below some function symbol from Σ_1. If testing satisfiability of ground clauses in $\mathcal{T}_0$ is decidable, then so is testing satisfiability of ground clauses in $\mathcal{T}_1$. Assume that the complexity of testing the satisfiability w.r.t. $\mathcal{T}_0$ of a set of ground clauses of size m can be described by a function $g(m)$. Let G be a set of $\mathcal{T}_1$-clauses such that $\Psi_{\mathcal{K}}(G)$ has size n. Then the complexity of checking the satisfiability of G w.r.t. $\mathcal{T}_1$ is of order $g(n^k)$, where k is the maximum number of free variables in a clause in $\mathcal{K}$ (but at least 2).*

[2] $\mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G$ can be flattened and purified by introducing, in a bottom-up manner, new constants c_t for subterms $t = f(g_1, \ldots, g_n)$ with $f \in \Sigma_1$, g_i ground $\Sigma_0 \cup \Sigma_c$-terms (where Σ_c is a set of constants which contains the constants introduced by flattening, resp. purification), together with corresponding definitions $c_t = t$. We obtain a set of clauses $\mathcal{K}_0 \cup G_0 \cup \mathsf{Def}$, where Def consists of ground unit clauses of the form $f(g_1, \ldots, g_n) = c$, where $f \in \Sigma_1$, c is a constant, $g_1, \ldots, g_n$ are ground $\Sigma_0 \cup \Sigma_c$-terms, and $\mathcal{K}_0$ and G_0 are $\Sigma_0 \cup \Sigma_c$-clauses. Flattening and purification preserve satisfiability and unsatisfiability w.r.t. total algebras, and w.r.t. partial algebras in which all ground subterms which are flattened are defined [14]. In what follows, we explicitly indicate the sorts of the constraints in Def by using indices, i.e. $\mathsf{Def} = \bigcup_{s \in S} \mathsf{Def}_s$.

Examples of Local Extensions. The locality of an extension can either be proved directly, or by proving embeddability of partial into total models.

Theorem 3 ([14,16,11,17]). *The following theory extensions are local:*

(1) Any extension of a theory with free function symbols;
(2) Extensions of any base theory $\mathcal{T}_0$ with functions satisfying axioms of the form

$$\mathsf{GBounded}(f) \qquad \bigwedge_{i=1}^{n}(\phi_i(\overline{x}) \rightarrow s_i \leq f(\overline{x}) \leq t_i)$$

where Π_0 contains a sort s for which a reflexive binary relation $\leq$ exists, s_i, t_i are Σ_0-terms of sort s and ϕ_i are Π_0-formulae s.t. for $i \neq j$, $\phi_i \wedge \phi_j \models_{\mathcal{T}_0} \bot$, and $\mathcal{T}_0 \models \forall \overline{x}(\phi_i(\overline{x}) \rightarrow s_i(\overline{x}) \leq t_i(\overline{x}))$.

3 Functions on Absolutely Free Data Structures

Let $\mathsf{AbsFree}_{\Sigma_0} = (\bigcup_{c\in\Sigma_0}(\mathsf{Inj}_c) \cup (\mathsf{Acyc}_c)) \cup \bigcup_{\substack{c,d\in\Sigma \\ c\neq d}} \mathsf{Disjoint}(c,d)$, where:

$$(\mathsf{Inj}_c) \qquad c(x_1,\ldots,x_n) = c(y_1,\ldots,y_n) \rightarrow \bigwedge_{i=1}^{n} x_i = y_i$$

$$(\mathsf{Acyc}_c) \qquad c(t_1,\ldots,t_n) \neq x \text{ if } x \text{ occurs in some } t_i$$

$$\mathsf{Disjoint}(c,d) \qquad c(x_1,\ldots,x_n) \neq d(y_1,\ldots,y_k) \qquad \text{if } c \neq d$$

Note that (Acyc_c) is an axiom schema (representing an infinite set of axioms).

Theorem 4. *The following theories are local:*

(a) The theory $\mathsf{AbsFree}_{\Sigma_0}$ of absolutely free constructors in Σ_0.
(b) Any theory $\mathsf{AbsFree}_{\Sigma_0\setminus\Sigma}$ obtained from $\mathsf{AbsFree}_{\Sigma_0}$ by dropping the acyclicity condition for a set $\Sigma \subseteq \Sigma_0$ of constructors.
(c) $\mathcal{T} \cup \mathsf{Sel}(\Sigma')$, where $\mathcal{T}$ is one of the theories in (a) or (b), and $\mathsf{Sel}(\Sigma') = \bigcup_{c\in\Sigma'}\bigcup_{i=1}^{n}\mathsf{Sel}(s_i^c, c)$ axiomatizes a family of selectors $s_1^c,\ldots,s_n^c$, where $n = a(c)$, corresponding to constructors $c \in \Sigma' \subseteq \Sigma_0$. Here,

$$\mathsf{Sel}(s_i, c) \qquad \forall x, x_1,\ldots,x_n \quad x = c(x_1,\ldots,x_n) \rightarrow s_i(x) = x_i.$$

In addition, $\mathcal{K} = \mathsf{AbsFree}_{\Sigma_0} \cup \mathsf{Sel}(\Sigma_0) \cup \mathsf{IsC}$, where

$$(\mathsf{IsC}) \quad \forall x \bigvee_{c\in\Sigma_0} x = c(s_1^c(x),\ldots,s_{a(c)}^c(x))$$

has the property that for every set G of ground $\Sigma_0 \cup \mathsf{Sel} \cup \Sigma_c$-clauses (where Σ_c is a set of additional constants), $\mathcal{K} \wedge G \models \bot$ iff $\mathcal{K}[\Psi(G)] \wedge G \models \bot$, where $\Psi(G) = \mathsf{st}(G) \cup \bigcup_{a\in\Sigma_c\cap\mathsf{st}(G)}\bigcup_{c\in\Sigma_0}(\{s_i^c(a) \mid 1\leq i\leq a(c)\}\cup\{c(s_1^c(a),\ldots,s_n^c(a))\})$.

Proof: This is proved by showing that every weak partial model of the axioms for (a)–(c) weakly embeds into a total model of the axioms. The locality then follows from the link between embeddability and locality established in [7]. □

The reduction to the pure theory of equality made possible by Theorem 4 is very similar to Oppen's method [13] for deciding satisfiability of ground formulae for

free recursive data structures by bi-directional closure. Quantifier elimination (cf. [13]) followed by the reduction enabled by Theorem 4 can be used to obtain a decision procedure for the first-order theory of absolutely free constructors axiomatized by $\mathsf{AbsFree}_{\Sigma_0} \cup \mathsf{Sel}(\Sigma_0) \cup \mathsf{IsC}$.

We consider extensions of $\mathsf{AbsFree}_{\Sigma_0}$ with new function symbols, possibly with codomain of a different sort, i.e. theories over the signature $S=\{d, s_1, \ldots, s_n\}$, where d is the "data" sort; we do not impose any restriction on the nature of the sorts in s_i (some may be equal to d). The function symbols are:

- constructors $c \in \Sigma$ (arity $d^n \rightarrow d$), and corresponding selectors s_i^c (arity $d \rightarrow d$);
- all functions Σ_{s_i} in the signature of the theory of sort s_i, for $i = 1, \ldots, n$;
- for every $1 \leq i \leq n$, a set Σ_i of functions of sort $d \rightarrow s_i$.

In what follows we will analyze certain such extensions for which decision procedures for ground satisfiability exist[3]. We assume for simplicity that $S = \{d, s\}$.

3.1 A Class of Recursively Defined Functions

Let $S = \{d, s\}$, where d is the "data" sort and s is a different sort (output sort for some of the recursively defined functions).

Let $\mathcal{T}_s$ be a theory of sort s. We consider extensions of the disjoint combination of $\mathsf{AbsFree}_{\Sigma_0}$ and $\mathcal{T}_s$ with functions in a set $\Sigma = \Sigma_1 \cup \Sigma_2$, where the functions in Σ_1 have arity $d \rightarrow d$ and those in Σ_2 have arity $d \rightarrow s$. If f has sort $d \rightarrow b$, with $b \in S$, we denote its output sort b by $o(f)$. Let $\Sigma_{o(f)}$ be Σ_0 if $o(f) = d$, or Σ_s if $o(f) = s$, and $\mathcal{T}_{o(f)}$ be the theory $\mathsf{AbsFree}_{\Sigma_0}$ if $o(f) = d$, or $\mathcal{T}_s$ if $o(f) = s$. For every $f \in \Sigma$ we assume that a subset $\Sigma_r(f) \subseteq \Sigma_0$ is specified (a set of constructors for which recursion axioms for f exist).

We consider theories of the form $\mathcal{T} = \mathsf{AbsFree}_{\Sigma_0} \cup \mathcal{T}_s \cup \mathsf{Rec}_\Sigma$, where $\mathsf{Rec}_\Sigma = \bigcup_{f \in \Sigma} \mathsf{Rec}_f$ is a set of axioms of the form:

$$\mathsf{Rec}_f \quad \left\{ \begin{array}{c} f(k) = k_f \\ f(c(x_1, \ldots, x_n)) = g^{c,f}(f(x_1), \ldots, f(x_n)) \end{array} \right.$$

where k, c range over all constructors in $\Sigma_r(f) \subseteq \Sigma_0$, with $a(k) = 0, a(c) = n$, k_f are ground $\Sigma_{o(f)}$-terms and the functions $g^{c,f}$ are expressible by $\Sigma_{o(f)}$-terms.

We also consider extensions with a new set of functions satisfying definitions by guarded recursion of the form $\mathsf{Rec}_\Sigma^g = \bigcup_{f \in \Sigma} \mathsf{Rec}_f^g$:

$$\mathsf{Rec}_f^g \quad \left\{ \begin{array}{rl} f(k) = k_f & \\ f(c(x_1, \ldots, x_n)) = & \left\{ \begin{array}{ll} g_1^{c,f}(f(x_1), \ldots, f(x_n)) & \text{if } \phi_1(f(x_1), \ldots, f(x_n)) \\ \ldots & \\ g_k^{c,f}(f(x_1), \ldots, f(x_n)) & \text{if } \phi_k(f(x_1), \ldots, f(x_n)) \end{array} \right. \end{array} \right.$$

[3] In this paper we only focus on the problem of checking the satisfiability of sets of ground clauses, although it appears that when adding axiom IsC decision procedures for larger fragments can be obtained using arguments similar to those used in [18].

where k, c range over all constructors in $\Sigma_r(f) \subseteq \Sigma_0$, with $a(k) = 0, a(c) = n$, k_f are ground $\Sigma_{o(f)}$-terms and the functions $g_i^{c,f}$ are expressible by $\Sigma_{o(f)}$-terms, and $\phi_i(x_1, \ldots, x_n)$ are $\Sigma_{o(f)}$-formulae with free variables $x_1, \ldots, x_n$, where $\phi_i \wedge \phi_j \models_{\mathcal{T}_{o(f)}} \bot$ for $i \neq j$.

Definition 1. *A definition of type* Rec_f *is* exhaustive *if* $\Sigma_r(f) = \Sigma_0$ *(i.e.* Rec_f *contains recursive definitions for terms starting with any* $c \in \Sigma_0$*). A definition of type* Rec_f^g *is* exhaustive *if* $\Sigma_r(f) = \Sigma_0$ *and for every definition, the disjoint guards* $\phi_1, \ldots, \phi_n$ *are exhaustive, i.e.* $\mathcal{T}_{o(f)} \models \forall \overline{x} \phi_1(\overline{x}) \vee \ldots \vee \phi_n(\overline{x})$. Quasi-exhaustive *definitions are defined similarly, by allowing that* $\Sigma_0 \setminus \Sigma_r(f)$ *may consist of constants.*

Example 5. *Let* $\Sigma_0 = \{c_0, c\}$ *with* $a(c_0) = 0, a(c) = n$. *Let* $\mathcal{T}_0 = \mathsf{AbsFree}_{\Sigma_0} \cup \mathcal{T}_s$ *be the disjoint, many-sorted combination of the theory* $\mathsf{AbsFree}_{\Sigma_0}$ *(sort* d*) and* $\mathcal{T}_{\mathsf{num}}$*, the theory of natural numbers with addition (sort* num*).*

(1) A size *function can be axiomatized by* $\mathsf{Rec}_{\mathsf{size}}$*:*

$$\begin{cases} \mathsf{size}(c_0) = 1 \\ \mathsf{size}(c(x_1, \ldots, x_n)) = 1 + \mathsf{size}(x_1) + \cdots + \mathsf{size}(x_n) \end{cases}$$

(2) A depth *function can be axiomatized by the following definition* $\mathsf{Rec}_{\mathsf{depth}}^g$ *(of type* Rec^g *due to* max*):*

$$\begin{cases} \mathsf{depth}(c_0) = 1 \\ \mathsf{depth}(c(x_1, \ldots, x_n)) = 1 + \mathsf{max}\{\mathsf{depth}(x_1), \ldots, \mathsf{depth}(x_n)\} \end{cases}$$

Example 6. *Let* $\Sigma_0 = \{c_0, d_0, c\}$ *with* $a(c_0) = a(d_0) = 0, a(c) = n$, *and let* $\mathcal{T}_0 = \mathsf{AbsFree}_{\Sigma_0} \cup \mathsf{Bool}$ *be the disjoint combination of the theories* $\mathsf{AbsFree}_{\Sigma_0}$ *(sort* d*) and* Bool*, having as model the two-element Boolean algebra* $\mathsf{B}_2 = (\{\mathsf{t}, \mathsf{f}\}, \sqcap, \sqcup, \neg)$ *(sort* bool*) with a function* has_{c_0} *with output of sort* bool*, defined by* $\mathsf{Rec}_{\mathsf{has}_{c_0}}$*:*

$$\begin{cases} \mathsf{has}_{c_0}(c_0) = \mathsf{t} \\ \mathsf{has}_{c_0}(d_0) = \mathsf{f} \\ \mathsf{has}_{c_0}(c(x_1, \ldots, x_n)) = \bigsqcup_{i=1}^n \mathsf{has}_{c_0}(x_i) \ (\textstyle\bigsqcup \textit{ is the supremum operation in } \mathsf{B}_2). \end{cases}$$

Problem. We analyze the problem of testing satisfiability of conjunctions G of ground unit $\Sigma_0 \cup \Sigma_1 \cup \Sigma_2 \cup \Sigma_c$-clauses, where Σ_c is a set of new constants:

$$(\mathsf{AbsFree}_{\Sigma_0} \cup \mathcal{T}_s \cup \mathsf{Rec}_{\Sigma_1}^{[g]} \cup \mathsf{Rec}_{\Sigma_2}^{[g]}) \wedge G \models \bot$$

(If $\Sigma_2 = \emptyset$, $\mathcal{T}_s$ can be omitted.) In what follows we use the abbreviations $\Sigma = \Sigma_1 \cup \Sigma_2$, $\mathsf{Rec}_{\Sigma}^g = \mathsf{Rec}_{\Sigma_1}^g \cup \mathsf{Rec}_{\Sigma_2}^g$, and $\mathsf{Rec}_{\Sigma} = \mathsf{Rec}_{\Sigma_1} \cup \mathsf{Rec}_{\Sigma_2}$.

The form of the ground formulae to be considered can be simplified as follows:

Lemma 7. *For every set* G *of ground unit* $\Sigma_0 \cup \Sigma \cup \Sigma_c$*-clauses there exists a set* G' *of* Σ*-flat ground unit* $\Sigma_0 \cup \Sigma \cup \Sigma_c'$*-clauses (where* $\Sigma_c \subseteq \Sigma_c'$*) of the form*

$$G' = C_s \wedge C_{\Sigma_0} \wedge C_{\Sigma} \wedge NC_{\Sigma_c'},$$

where C_s *is a set of (unit)* Σ_s*-clauses (if* $\Sigma_2 \neq \emptyset$*) and* $C_{\Sigma_0}, C_{\Sigma}, NC_{\Sigma_c'}$ *are (possibly empty) conjunctions of literals of the form:*

C_{Σ_0}: $c = c'$ *and* $c \neq c$, *where* $c, c' \in \Sigma_0$, *nullary;*
C_{Σ}: $(\neg)f(t_d) = t'$, *where* $f \in \Sigma_1 \cup \Sigma_2$, t_d *is a* $\Sigma_0 \cup \Sigma'_c$*-term,* t' *a* $\Sigma_{o(f)} \cup \Sigma'_c$*-term;*
$(\neg)f(t_d) = f'(t'_d)$, *where* $f, g \in \Sigma_2$, *and* t_d, t'_d *are* $\Sigma_0 \cup \Sigma'_c$*-terms;*
$NC_{\Sigma'_c}$: $t_d \neq t'_d$, *where* t_d, t'_d *are* $\Sigma_0 \cup \Sigma'_c$*-terms;*

such that G *and* G' *are equisatisfiable w.r.t.* $\mathsf{AbsFree}_{\Sigma_0} \cup \mathcal{T}_s \cup \mathcal{K}$ *for any set of clauses* $\mathcal{K}$ *axiomatizing the properties of the functions in* Σ.

Remark 8. *If* $\mathcal{K} = \mathsf{Rec}_\Sigma$ *we can ensure that, for every literal in* C_Σ, t_d (t'_d) *either starts with a constructor* $c \notin \Sigma_r(f)$ (*resp.* $c \notin \Sigma_r(f')$) *or is equal to some* $a \in \Sigma'_c$. *If the definition of* $f \in \Sigma$ *is exhaustive (resp. quasi-exhaustive), we can ensure that the only occurrence of* f *in* G' *is at the root of a term, in terms of the form* $f(a)$, *where* $a \in \Sigma_c$ (*resp., if* Rec_f *is quasi-exhaustive,* $a \in \Sigma_c \cup (\Sigma_0 \backslash \Sigma_r(f))$). *We can ensure that each such* $f(a)$ *occurs in at most one positive clause by replacing any conjunction* $f(a) = t_1 \wedge f(a) = t_2$ *with* $f(a) = t_1 \wedge t_1 = t_2$. $f(a) = t_1 \wedge f(a) \neq t_2$ *can also be replaced with the (equisatisfiable) conjunction:* $f(a) = t_1 \wedge t_1 \neq t_2$.

We make the following assumptions:

Assumption 1: Either $\Sigma_1 = \emptyset$, or else $\Sigma_1 \neq \emptyset$ and Rec_{Σ_1} is quasi-exhaustive.
Assumption 2: G is a set of ground unit clauses with the property that any occurrence of a function symbol in Σ_1 is in positive unit clauses of G of the form $f(a) = t$, with $a \in \Sigma_c \cup (\Sigma_0 \backslash \Sigma_r(f))$, and G does not contain any equalities between $\Sigma_0 \cup \Sigma_c$-terms. (By Remark 8, we can assume w.l.o.g. that for all $f \in \Sigma_1$ and $a \in \Sigma_c \cup (\Sigma_0 \backslash \Sigma_r(f))$, $f(a)$ occurs in at most one positive unit clause of G of the form $f(a) = t$.)

Theorem 9. *If Assumption 1 holds, then:*

(1) $\mathsf{AbsFree}_{\Sigma_0} \cup \mathcal{T}_s \cup \mathsf{Rec}_{\Sigma_2}$ *is a* Ψ*-local extension of* $\mathsf{AbsFree}_{\Sigma_0} \cup \mathcal{T}_s$;
(2) If Rec_{Σ_1} *is quasi-exhaustive, then* $\mathsf{AbsFree}_{\Sigma_0} \cup \mathcal{T}_s \cup \mathsf{Rec}_{\Sigma_1} \cup \mathsf{Rec}_{\Sigma_2}$ *satisfies the* Ψ*-locality conditions of an extension of* $\mathsf{AbsFree}_{\Sigma_0} \cup \mathcal{T}_s$ *for every set* G *of unit clauses which satisfy Assumption 2;*

where Ψ *associates with any set* T *of ground terms the smallest set which contains* T *and if* $f(c(t_1, \ldots, t_n)) \in \Psi(T)$ *and* $c \in \Sigma_r(f)$ *then* $f(t_i) \in \Psi(T)$ *for* $i = 1, \ldots, n$.

Similar results hold for extensions with Rec^g_Σ *(under similar assumptions) provided the guards* ϕ_i *in the recursive definitions of functions in* Σ_1 *are positive. The results can even be extended to recursive definitions of the form* $\mathsf{ERec}^{[g]}_f$:

$$\begin{cases} f(k, x) = k_f(x) \\ f(c(x_1, \ldots, x_n), x) = \begin{cases} g_1^{c,f}(f(x_1, x), \ldots, f(x_n, x), x) & \text{if } \phi_1(f(x_1), \ldots, f(x_n)) \\ \ldots \\ g_k^{c,f}(f(x_1, x), \ldots, f(x_n, x), x) & \text{if } \phi_k(f(x_1), \ldots, f(x_n)) \end{cases} \end{cases}$$

where k, c *range over* $\Sigma_r(f)$, $a(k) = 0, a(c) = n$, $k_f(x)$ *are* $\Sigma_{o(f)}$*-terms with free variable* x, $g_i^{c,f}$ *are functions expressible as* $\Sigma_{o(f)}$*-terms, and* $\phi_i(x_1, \ldots, x_n)$ *are* $\Sigma_{o(f)}$*-formulae with free variables* $x_1, \ldots, x_n$, *s.t.* $\phi_i \wedge \phi_j \models_{\mathcal{T}_{o(f)}} \bot$ *for* $i \neq j$.

Note: We can actually prove a variant of ELoc^{Ψ}, in which we can allow first-order Σ_s-constraints in $(\mathsf{E})\mathsf{Rec}^g_\Sigma$ and in G.

Example 10. *Let $\Sigma_0 = \{c_0, d_0, c\}$, where c is a binary constructor and c_0, d_0 are nullary. Consider the recursive definition $\mathsf{Rec}_{\mathsf{has}_{c_0}}$ of the function has_{c_0} in Example 6. We want to show that $\mathsf{AbsFree}_{\Sigma_0} \cup \mathsf{Bool} \cup \mathsf{Rec}_{\mathsf{has}_{c_0}} \models G_1$ where*

$$G_1 = \forall \overline{x}(\mathsf{has}_{c_0}(x){=}\mathsf{t} \wedge z_1{=}c(y_1, c(x_1, x)) \wedge z_1{=}c(y_2, y_3) \rightarrow \mathsf{has}_{c_0}(y_3){=}\mathsf{t})$$
$$G = \neg G_1 = (\mathsf{has}_{c_0}(a){=}\mathsf{t} \wedge c_1{=}c(b_1, c(a_1, a)) \wedge c_1{=}c(b_2, b_3) \wedge \mathsf{has}_{c_0}(b_3){=}\mathsf{f}),$$

where $\Sigma_c = \{a, a_1, b_1, b_2, b_3, c_1\}$. We transform G as explained in Lemma 7 by inferring all equalities entailed by the equalities between constructor terms in G; if $a_i = a_j$ (resp. $a_i = c(a_1, \ldots, a_n)$) is entailed we replace a_i with a_j (resp. with $c(a_1, \ldots, a_n)$). We obtain the equisatisfiable set of ground clauses:

$$G' = (\mathsf{has}_{c_0}(a){=}\mathsf{t} \wedge \mathsf{has}_{c_0}(c(a_1, a)){=}\mathsf{f}).$$

$(\mathsf{AbsFree}_{\Sigma_0} \cup \mathsf{Bool} \cup \mathsf{Rec}_{\mathsf{has}_{c_0}}) \cup G' \models \bot$ *iff* $(\mathsf{AbsFree}_{\Sigma_0} \cup \mathsf{Bool}) \cup \mathsf{Rec}_{\mathsf{has}_{c_0}}[\Psi(G')] \cup G' \models \bot$, *where* $\Psi(G') = \{\mathsf{has}_{c_0}(c(a_1, a)), \mathsf{has}_{c_0}(a_1), \mathsf{has}_{c_0}(a)\}$ *by Theorem 9. After purification we obtain:*

$\mathsf{Def}_{\mathsf{bool}}$	$G_0 \wedge \mathsf{Rec}_{\mathsf{has}_{c_0}}[\Psi(G)]_0$
$\mathsf{has}_{c_0}(a_1) = h_1 \wedge \mathsf{has}_{c_0}(a) = h_2 \wedge \mathsf{has}_{c_0}(c(a_1, a)) = h_3$	$h_2 = \mathsf{t} \wedge h_3 = \mathsf{f} \wedge h_3 = h_1 \sqcup h_2$

We immediately obtain a contradiction in Bool*, without needing to consider* Con_0 *or a further reduction to a satisfiability test w.r.t.* $\mathsf{AbsFree}_{\Sigma_0}$.

Combining Recursive Definitions with Boundedness. We analyze the locality of combinations of $\mathsf{Rec}^{[g]}_\Sigma$ with boundedness axioms, of the type:

$$\mathsf{Bounded}(f) \qquad \forall x(t_1 \leq f(x) \leq t_2)$$

Theorem 11. *Assume that $\leq$ is a partial order in all models of $\mathcal{T}_s$, $a(f) = d \rightarrow s$, t_1, t_2 are Σ_s-terms with $\mathcal{T}_s \models t_1 \leq t_2$, and all functions $g_i^{c,f}$ used in the definition of f have the property:*

$$\forall x_1, \ldots, x_n (\bigwedge_{i=1}^{n} t_1 \leq x_i \leq t_2 \rightarrow t_1 \leq g_i^{c,f}(x_1, \ldots, x_n) \leq t_2), \text{ where } n = a(c).$$

If Assumption 1 holds then $\mathsf{AbsFree}_{\Sigma_0} \cup \mathcal{T}_s \cup \mathsf{Rec}^{[g]}_f \cup \mathsf{Bounded}$ *is a Ψ-local extension of* $\mathsf{AbsFree}_{\Sigma_0} \cup \mathcal{T}_s$*, where Ψ is defined as in Theorem 9.*

Proof: The conditions on the functions $g_i^{c,f}$ ensure that in the completion process used in Theorem 9 the corresponding properties of f can be guaranteed. □

Example 12. *(1) We want to check whether* $\mathsf{AbsFree}_{\Sigma_0} \cup \mathbb{Z} \cup \mathsf{Rec}_{\mathsf{depth}}$ *entails*

$$G_1 = \forall x_1, x_2, x_3, x_4\ (\mathsf{depth}(x_1) \leq \mathsf{depth}(x_2) \wedge \mathsf{depth}(x_4) \leq \mathsf{depth}(x_3) \wedge x_4{=}c(x_2) \rightarrow \mathsf{depth}(d(x_1, e(x_2, c'))) \leq \mathsf{depth}(e(x_4, x_3))),$$

where Σ_0 contains the constructors c' (nullary), c (unary), and d, e (binary). By Ψ-locality, this can be reduced to testing the satisfiability of the following conjunction of ground clauses containing the additional constants:

$$\Sigma_c = \{a_1, a_2, a_3, a_4, d_1, d_2, d_3, d_4, e_1, e_2, e_3, g_1, g_2, g_3, c_2', d_2'\}$$

(below we present the flattened and purified form), where $G = \neg G_1$:

Def_d	$\mathsf{Def}_{\mathsf{num}}$	G_{0d}	$G_{0\mathsf{num}}$	$\mathsf{Rec}_{\mathsf{depth}}[\Psi(G)]_0$
$d(a_1, e_2) = e_1$	$\mathsf{depth}(a_i) = d_i (i = 1-4)$	$a_4 = c_2'$	$d_1 \leq d_2$	$g_1 = 1 + \mathsf{max}\{d_1, g_2\}$
$e(a_2, c') = e_2$	$\mathsf{depth}(e_i) = g_i (i = 1, 2, 3)$		$d_4 \leq d_3$	$g_2 = 1 + \mathsf{max}\{d_2, 1\}$
$e(a_4, a_3) = e_3$	$\mathsf{depth}(c_2') = d_2'$		$g_1 \not\leq g_3$	$g_3 = 1 + \mathsf{max}\{d_4, d_3\}$
$c(a_2) = c_2'$				$d_2' = 1 + d_2$

Let Con_0 consist of all the instances of congruence axioms for c, d, e and depth. $G_0 \cup \mathsf{Rec}_{\mathsf{depth}}[\Psi(G)]_0 \cup \mathsf{Con}_0$ is satisfiable in $\mathsf{AbsFree}_{\Sigma_0} \cup \mathbb{Z}$. A satisfying assignment is: $d_1 = d_2 = 0$ and $d_2' = d_4 = d_3 = 1$ (d_2' and d_4 need to be equal due to Con_0 because $c_2' = a_4$; and $d_4 \leq d_3$). $g_2 = 1 + \mathsf{max}\{0, 1\} = 2$, $g_1 = 1 + \mathsf{max}\{d_1, g_2\} = 3$ and $g_3 = 1 + \mathsf{max}\{d_4, d_3\} = 1 + d_4 = 2$. Thus, $\mathsf{AbsFree}_{\Sigma_0} \cup \mathbb{Z} \cup \mathsf{Rec}_{\mathsf{depth}} \not\models G_1$.

(2) We now show that $\mathsf{AbsFree}_{\Sigma_0} \cup \mathbb{Z} \cup \mathsf{Rec}_{\mathsf{depth}} \cup \mathsf{Bounded}(\mathsf{depth}) \models G_1$, where

$$\mathsf{Bounded}(\mathsf{depth}) \qquad \forall x(\mathsf{depth}(x) \geq 1).$$

By Theorem 11, we only need to consider the instances of $\mathsf{Bounded}(\mathsf{depth})$ containing terms in $\mathsf{Def}_{\mathsf{num}}$, i.e. the constraints $d_i \geq 1$ for $i \in \{1, \ldots, 4\}$; $g_i \geq 1$ for $i \in \{1, \ldots, 3\}$ and $d_2' \geq 1$. Con_0 can be used to derive $d_4 = d_2'$. We obtain:

$g_1 = 1 + \mathsf{max}\{d_1, g_2\} = 1 + \mathsf{max}\{d_1, 1 + \mathsf{max}\{d_2, 1\}\} = 1 + \mathsf{max}\{d_1, 1 + d_2\} = 2 + d_2$
$g_3 = 1 + \mathsf{max}\{d_4, d_3\} = 1 + d_3 \geq 1 + d_4 = 1 + d_2' = 2 + d_2.$
which together with $g_1 \not\leq g_3$ yields a contradiction.

3.2 Restricting to Term-Generated Algebras

The apparent paradox in the first part of Example 12 is due to the fact that the axiomatization of $\mathsf{AbsFree}_{\Sigma_0}$ makes it possible to consider models in which the constants in Σ_c are not interpreted as ground Σ_0-terms. We would like to consider only models for which the support A_d of sort d is the set $T_{\Sigma_0}(\emptyset)$ of ground Σ_0-terms (we will refer to them as *term generated models*)[4]. We will assume that the axiomatization of the recursive functions contains a family of constraints $\{C(a) \mid a \in \Sigma_c\}$ expressed in first order logic on the values the function needs to take on any element in Σ_c with the property:

(**TG**) $C(a)$ iff there exists $t \in T_{\Sigma_0}(\emptyset)$ such that for all $f \in \Sigma_2$, $f(a) = f(t)$.

[4] For expressing this, we can use axiom IsC (cf. Theorem 4) or the axiom used in [18]: $(\mathsf{IsConstr}) \forall x \bigvee_{c \in \Sigma_0} \mathsf{Is}_c(x)$ where $\mathsf{Is}_c(x) = \exists x_1, \ldots, x_n : x = c(x_1, \ldots, x_n)$.

Example 13. *Some examples are presented below:*

(1) Assume $\Sigma_2 = \{\mathsf{size}\}$ *(the size function over absolutely free algebras with set of constructors* $\{c_i \mid 1 \leq i \leq n\}$ *with arities* $a(c_i)$*). The following size constraints have the desired property (cf. also [18]):*

$$C(a) = \exists x_1, \ldots, x_n(\mathsf{size}(a) = (\sum_{i=1}^{n} a(c_i) * x_i) + 1).$$

To prove this, note that for every term t, $\mathsf{size}(t) = (\sum_{i=1}^{n} a(c_i) * n(c_i, t) + 1)$, *where* $n(c_i, t)$ *is the number of times* c_i *occurs in* t*. Thus, if there exists* t *such that* $\mathsf{size}(t) = \mathsf{size}(a)$*, then* $C(a)$ *is true. Conversely, if* $C(a)$ *is true* $\mathsf{size}(a) = \mathsf{size}(t)$ *for every term with* x_i *occurrences of the constructor* c_i *for* $i = 1, ..., n$.

(2) Consider the depth function (with output sort int*) over absolutely free algebras with set of constructors* $\{c_i \mid 1 \leq i \leq n\}$*. Then* $C(a) := \mathsf{depth}(a) \geq 1$.

In what follows we will assume that $\Sigma_1 = \emptyset$.

Theorem 14. *Assume that for every* $a \in \Sigma_c$*, a set* $C(a)$ *of constraints satisfying condition* **(TG)** *exists. Then* $\mathsf{AbsFree}_{\Sigma_0} \cup \mathcal{T}_s \cup \mathsf{Rec}^{[g]}_{\Sigma_2} \cup \bigcup_{a \in \Sigma_c} C(a)$ *is a* Ψ*-local extension of* $\mathsf{AbsFree}_c \cup \mathcal{T}_s$*, where* Ψ *is defined as in Theorem 9.*

Note: As in Theorem 9, we can prove, in fact, ELoc^{Ψ}-locality. Hence, the possibility that $C(a)$ may be a first-order formula of sort s is not a problem.

In order to guarantee that we test satisfiability w.r.t. term generated models, in general we have to add, in addition to the constraints $C(a)$, for every function symbol $f \in \Sigma_2$, additional counting constraints describing, for every $x \in A_s$, the maximal number of distinct terms t in $T_{\Sigma_0}(\emptyset)$ with $f(t) = x$. If Σ_0 contains infinitely many nullary constructors the number of distinct terms t in $T_{\Sigma_0}(\emptyset)$ with $f(t) = x$ is infinite, so no counting constraints need to be imposed.

Counting constraints are important if Σ_0 contains only finitely many nullary constructors and if the set G of ground unit clauses we consider contains negative (unit) $\Sigma_0 \cup \Sigma_c$-clauses. For the sake of simplicity, we here only consider sets G of unit ground clauses which contain only negative (unit) clauses of sort s.

Lemma 15. *Assume that* $\Sigma_1 = \emptyset$ *and for every* $a \in \Sigma_c$ *there exists a set* $C(a)$ *of constraints such that condition* **(TG)** *holds. The following are equivalent for any set* G *of unit* $\Sigma_0 \cup \Sigma_2 \cup \Sigma_c$*-clauses in which all negative literals have all sort* s*.*

(1) There exists a term-generated model $A = (T_{\Sigma_0}(\emptyset), A_s, \{f_A\}_{f \in \Sigma_2}, \{a_A\}_{a \in \Sigma_c})$ *of* $\mathsf{AbsFree}_{\Sigma_0} \cup \mathcal{T}_s \cup \mathsf{Rec}^{[g]}_{\Sigma_2}$ *and* G.

(2) There exists a model $F = (T_{\Sigma_0}(\Sigma_c), A_s, \{f_F\}_{f \in \Sigma_2}, \{a_F\}_{a \in \Sigma_c})$ *of* $\mathsf{AbsFree}_{\Sigma_0} \cup \mathcal{T}_s \cup \mathsf{Rec}^{[g]}_{\Sigma_2} \cup \bigcup_{a \in \Sigma_c} C(a)$ *and* G*, where for every* $a \in \Sigma_c$*,* $a_F = a$.

(3) There exists a model $A = (A_d, A_s, \{f_A\}_{f \in \Sigma_2}, \{a_A\}_{a \in \Sigma_c})$ *of* $\mathsf{AbsFree}_{\Sigma_0} \cup \mathcal{T}_s \cup \mathsf{Rec}^{[g]}_{\Sigma_2} \cup \bigcup_{a \in \Sigma_c} C(a)$ *and* G.

From Theorem 14 and Lemma 15 it follows that for every set G of ground unit clauses in which all negative (unit) clauses consist of literals of sort s, testing whether there exists a term-generated model of $\mathsf{AbsFree}_{\Sigma_0} \cup \mathcal{T}_s \cup \mathsf{Rec}^{[g]}_{\Sigma_2}$ and G can be done by computing $\mathsf{Rec}^{[g]}_{\Sigma_2}[\Psi(G)]$ and then reducing the problem hierarchically to a satisfiability test w.r.t. $\mathsf{AbsFree}_{\Sigma_0} \cup \mathcal{T}_s$.

Example 16. *Example 12 provides an example of a ground clause G for which $\mathsf{AbsFree}_{\Sigma_0} \cup \mathbb{Z} \cup \mathsf{Rec}_{\mathsf{depth}} \not\models G$, and $\mathsf{AbsFree}_{\Sigma_0} \cup \mathbb{Z} \cup \mathsf{Rec}_{\mathsf{depth}} \wedge \mathsf{Bounded}(\mathsf{depth}) \models G$. Example 12(2) shows that $\mathsf{AbsFree}_{\Sigma_0} \cup \mathbb{Z} \cup \mathsf{Rec}_{\mathsf{depth}} \cup \bigcup_{a \in Const(G)} C(a) \models G$, i.e. (by Lemma 15), G is true in every term-generated model of $\mathsf{AbsFree}_{\Sigma_0} \cup \mathbb{Z} \cup \mathsf{Rec}_{\mathsf{depth}}$.*

Similar results can be obtained if we relax the restriction on occurrences of negative clauses in G. If the set of nullary constructors in Σ_0 is infinite the extension is easy; otherwise we need to use equality completion and add counting constraints as done e.g. in [18] (assuming that there exist counting constraints expressible in first-order logic for the recursive definitions we consider).

4 More General Data Structures

We will now extend the results above to more general data structures. Consider a signature consisting of a set Σ_0 of constructors (including a set C of constants). Let E be an additional set of identities between Σ_0-terms.

Example 17. *Let $\Sigma_0 = \{c, c_0\}$, where c is a binary constructor and c_0 is a constant. We can impose that E includes one or more of the following equations:*

(A) $c(c(x,y),z) = c(x,c(y,z))$ *(associativity)*
(C) $c(x,y) = c(y,x)$ *(commutativity)*
(I) $c(x,x) = x$ *(idempotence)*
(N) $c(x,x) = c_0$ *(nilpotence)*

We consider many-sorted extensions of the theory defined by E with functions in $\Sigma = \Sigma_1 \cup \Sigma_2$, and sorts $S = \{d, s\}$, where the functions in Σ_1 have sort $d \to d$, those in Σ_2 have sort $d \to s$, and the functions in Σ satisfy additional axioms of the form Rec_Σ and ERec_Σ as defined in Section 3.1.[5] We therefore consider two-sorted theories of the form $E \cup \mathcal{T}_s \cup (\mathsf{E})\mathsf{Rec}_\Sigma$, where $\mathcal{T}_s$ is a theory of sort s. We make the following assumptions:

Assumption 3: We assume that:
(a) The equations in E only contain constructors c with $c \in \bigcap_{f \in \Sigma} \Sigma_r(f)$.
(b) For every $\forall \overline{x} \;\; t(\overline{x}) = s(\overline{x}) \in E$ and every $f \in \Sigma_1 \cup \Sigma_2$ let $t'(\overline{x})$ (resp. $s'(\overline{x})$) be the $\Sigma_{o(f)}$-term obtained by replacing every constructor $c \in \Sigma_0$ with the term-generated function[6] $g^{c,f}$. Then for every $f \in \Sigma_1$, $E \models \forall \overline{x} \;\; t'(\overline{x}) = s'(\overline{x})$, and for every $f \in \Sigma_2$, $\mathcal{T}_s \models \forall \overline{x} \;\; t'(\overline{x}) = s'(\overline{x})$.

[5] We restrict to unguarded recursive definitions of type Rec_Σ and ERec_Σ to simplify the presentation. Similar results can be obtained for definitions of the type Rec^g_Σ and ERec^g_Σ, with minor changes in Assumption 3.

[6] $g^{c,f}$ is the function (expressible as a $\Sigma_{o(f)}$-term) from the definition $f(c(x_1,\ldots,x_n)) = g^{c,f}(f(x_1),\ldots,f(x_n))$ in Rec_f.

Example 18. *Consider the extension of the theory of one binary associative and/or commutative function c with the size function defined as in Example 5(1). Then*

$$\mathsf{size}(c(x,y)) = g^c_{\mathsf{size}}(\mathsf{size}(x),\mathsf{size}(y)), \text{ where } g^c_{\mathsf{size}}(x,y) = 1+x+y.$$

Note that g^c_{size} is associative and commutative, so Assumption 3 holds.

$$g^c_{\mathsf{size}}(g^c_{\mathsf{size}}(x,y),z) = 1+(1+x+y)+z = 1+x+(1+y+z) = g^c_{\mathsf{size}}(x,g^c_{\mathsf{size}}(y,z));$$
$$g^c_{\mathsf{size}}(x,y) = 1+x+y = 1+y+x = g^c_{\mathsf{size}}(y,x).$$

Example 19. *Assume that Σ_0 only contains the binary constructor c satisfying a set E of axioms containing some of the axioms $\{(\mathbf{A}),(\mathbf{C}),(\mathbf{I})\}$ in Example 17. Let enc_k be a new function symbol (modeling encoding with key k) satisfying*

$$\mathsf{Rec}_{\mathsf{enc}} \qquad \mathsf{enc}_k(c(x,y)) = c(\mathsf{enc}_k(x),\mathsf{enc}_k(y)).$$

It is easy to see that $g^c_{\mathsf{enc}} = c$ and hence Assumption 3 is satisfied.

In what follows we assume that Assumption 3 holds, and that Rec_{Σ_1} is quasi-exhaustive. Note that in the presence of axioms such as associativity, the universal (Horn) theory of E itself may be undecidable. We will therefore only consider the simpler proof task of checking whether

$$E \cup [\mathsf{E}]\mathsf{Rec}^{[\mathsf{g}]}_{\Sigma_1} \cup [\mathsf{E}]\mathsf{Rec}^{[\mathsf{g}]}_{\Sigma_2} \models G_1,$$

where G_1 is a ground $\Sigma \cup \Sigma_1 \cup \Sigma_2$-clause of the form

$$\bigwedge_{k=1}^{l} g_k(c_k) = t^d_k \wedge \bigwedge_{i=1}^{n} f_i(t^d_i) = t^s_i \wedge \bigwedge_{j=1}^{m} f_j(t^d_j) = f'_j(t'^d_j) \rightarrow f(t_d) = t_s \qquad (1)$$

where $g_k \in \Sigma_1$, $c_k \in \Sigma_0 \backslash \Sigma_r(g_k)$, f_i, f'_i, f are functions in Σ_2 (with output sort s different from d), t^d_k, t'^d_k, t_d are ground Σ_0-terms, and t^s_k, t'^s_k, t_s are Σ_s-terms. We additionally assume that for every $g \in \Sigma_1$ and every $c \in \Sigma_0 \backslash \Sigma_r(g)$, $g(c)$ occurs at most once in the premise of G.

Remark. If Rec_{Σ_2} is quasi-exhaustive, G is equisatisfiable with a clause in which every occurrence of $f \in \Sigma_2$ is in a term of the form $f(c)$, with $c \in \Sigma_0 \backslash \Sigma_r(f)$.

Theorem 20. *Assume that Rec_{Σ_1}, Rec_{Σ_2} are quasi-exhaustive and Assumption 3 holds. The following are equivalent for any set G of $\Sigma_0 \cup \Sigma$-clauses of form (1):*

(1) $E \cup \mathsf{Rec}_{\Sigma_1} \cup \mathcal{T}_s \cup \mathsf{Rec}_{\Sigma_2} \models G$.
(2) G is true in all models $A = (A_d, A_s, \{f_A\}_{f\in\Sigma})$ of $E \cup \mathsf{Rec}_{\Sigma_1} \cup \mathcal{T}_s \cup \mathsf{Rec}_{\Sigma_2}$.
(3) G is true in all models $F = (T_{\Sigma_0}(\emptyset)/\equiv_E, A_s, \{f_A\}_{f\in\Sigma})$ of $E \cup \mathcal{T}_s \cup \mathsf{Rec}_{\Sigma_1} \cup \mathsf{Rec}_{\Sigma_2}$.
(4) G is true in all weak partial models $F = (T_{\Sigma_0}(\emptyset)/\equiv_E, A_s, \{f_A\}_{f\in\Sigma})$ of $E \cup \mathcal{T}_s \cup (\mathsf{Rec}_{\Sigma_1} \cup \mathsf{Rec}_{\Sigma_1})[\Psi(G)]$ in which all terms in $\Psi(G)$ are defined.

Similar results can also be obtained for definitions of type Rec^q_Σ or $\mathsf{ERec}^{[g]}_\Sigma$.

Note: We can impose boundedness conditions on the recursively defined functions without affecting locality (as for absolutely free constructors). [7]

4.1 An Example Inspired from Cryptography

In this section we illustrate the ideas on an example inspired by the treatment of a Dolev-Yao security protocol considered in [4] (cf. also Examples 17 and 19). Let $\Sigma_0 = \{c\} \cup C$, where c is a binary constructor, and let enc be a binary function. We analyze the following situations:

(1) c satisfies a set E of axioms and enc is a free binary function. By Theorem 3, the extension of E with the free function enc is a local extension of E.
(2) c is an absolutely free constructor, and enc satisfies the recursive definition:

$$(\mathsf{ERec}_{\mathsf{enc}}) \quad \forall x, y, z \;\; \mathsf{enc}(c(x, y), z) = c(\mathsf{enc}(x, z), \mathsf{enc}(y, z)).$$

By Theorem 9, the extension $\mathsf{AbsFree}_c \subseteq \mathsf{AbsFree}_c \cup \mathsf{ERec}_{\mathsf{enc}}$ satisfies the Ψ-locality condition for all clauses satisfying Assumption 2 (with Ψ as in Theorem 9).
(3) If c is associative (resp. commutative) and enc satisfies axiom $\mathsf{ERec}_{\mathsf{enc}}$ then Assumption 3 is satisfied, so, by Theorem 20, $E \cup \mathsf{ERec}_{\mathsf{enc}}$ satisfies the condition of a Ψ-local extension of E for all clauses of type (1).

Formalizing the Intruder Deduction Problem. We now formalize the version of the deduction system of the Dolev and Yao protocol given in [4]. Let E be the set of identities which specify the properties of the constructors in Σ_0. We use the following chain of successive theory extensions:

$$E \subseteq E \cup \mathsf{ERec}_{\mathsf{enc}} \subseteq E \cup \mathsf{ERec}_{\mathsf{enc}} \cup \mathsf{Bool} \cup \mathsf{Rec}^g_{\mathsf{known}},$$

where known has sort $d \to \mathsf{bool}$ and $\mathsf{Rec}^g_{\mathsf{known}}$ consists of the following axioms:

$$\begin{array}{ll} \forall x, y & \mathsf{known}(c(x, y)) = \mathsf{known}(x) \sqcap \mathsf{known}(y) \\ \forall x, y & \mathsf{known}(y) = \mathsf{t} \to \mathsf{known}(\mathsf{enc}(x, y)) = \mathsf{known}(x) \end{array}$$

Intruder deduction problem. The general statement of the intruder deduction problem is: "Given a finite set T of messages and a message m, is it possible to retrieve m from T?".

Encoding the intruder deduction problem. The finite set of known messages, $T = \{t_1, \ldots, t_n\}$, where t_i are ground $\Sigma_0 \cup \{\mathsf{enc}\}$-terms, is encoded as $\bigwedge_{i=1}^n \mathsf{known}(t_i) = \mathsf{t}$. With this encoding, the intruder deduction problem becomes:
"Test whether $E \cup \mathsf{Rec}_{\mathsf{enc}} \cup \mathsf{Bool} \cup \mathsf{Rec}_{\mathsf{known}} \models \bigwedge_{i=1}^n \mathsf{known}(t_i) = \mathsf{t} \to \mathsf{known}(m) = \mathsf{t}$."

[7] We can also consider axioms which link the values of functions $f_2 \in \Sigma_2$ and $f_1 \in \Sigma_1$ on the constants, such as e.g. "$f_2(f_1(c)) = t_s$" if we consider clauses G in which if $f_1(c) = t$ occurs then $t = c'$, where c' is a constant constructor not in $\Sigma_r(f_2)$. In the case of Σ_1-functions defined by ERec we can consider additional axioms of the form: $\phi(f_2(x)) \to f_2(f_1(c, x)) = t'_s$, where t'_s is a ground term of sort s either containing f_2 (and of the form $f_2(c')$) or a pure Σ_s-term.

Example 21. *We illustrate the hierarchical reasoning method we propose on the following example: Assume that* $E = \{(\mathbf{C})\}$ *and the intruder knows the messages* $c(a,b)$ *and* $\mathsf{enc}(c(c(e,f),e),c(b,a))$*. We check if he can retrieve* $c(f,e)$*, i.e. if*

$$G : (\mathsf{known}(c(a,b))=\mathsf{t}) \wedge (\mathsf{known}(\mathsf{enc}(c(c(e,f),e),c(b,a)))=\mathsf{t}) \wedge (\mathsf{known}(c(f,e))=\mathsf{f})$$

is unsatisfiable w.r.t. $E \cup \mathsf{Bool} \cup \mathsf{ERec}_{\mathsf{enc}} \cup \mathsf{Rec}^{g}_{\mathsf{known}}$*.* G *is equisatisfiable with a set* G' *of clauses obtained by applying all the definitions in* $\mathsf{ERec}_{\mathsf{enc}}$ *and* $\mathsf{Rec}^{g}_{\mathsf{known}}$*:*

$$G' : \begin{array}{l}(\mathsf{known}(\mathsf{enc}(e,c(b,a))) \sqcap \mathsf{known}(\mathsf{enc}(f,c(b,a))) \sqcap \mathsf{known}(\mathsf{enc}(e,c(b,a)))=\mathsf{t}) \\ \wedge\ (\mathsf{known}(a) \sqcap \mathsf{known}(b)=\mathsf{t}) \ \wedge\ (\mathsf{known}(f) \sqcap \mathsf{known}(e)=\mathsf{f}).\end{array}$$

By Theorem 20, we know that $E \cup \mathsf{Rec}_{\mathsf{enc}} \cup \mathsf{Bool} \cup \mathsf{Rec}_{\mathsf{known}} \wedge G' \models \bot$ *iff* $E \cup \mathsf{Rec}_{\mathsf{enc}} \cup \mathsf{Bool} \cup \mathsf{Rec}_{\mathsf{known}}[\Psi(G')] \wedge G' \models \bot$*. The reduction is illustrated below:*

$\mathsf{Def}_{\mathsf{bool}}$		$G'_0 \quad \wedge$	$\mathsf{Rec}_{\mathsf{known}}[\Psi(G')]_0$
$k_1 = \mathsf{known}(a)$	$k_5 = \mathsf{known}(\mathsf{enc}(e,c(b,a)))$	$k_1 \sqcap k_2 = \mathsf{t}$	$k_7 = k_2 \sqcap k_1$
$k_2 = \mathsf{known}(b)$	$k_6 = \mathsf{known}(\mathsf{enc}(f,c(b,a)))$	$k_3 \sqcap k_4 = \mathsf{f}$	$k_7 = \mathsf{t} \rightarrow k_5 = k_3$
$k_3 = \mathsf{known}(e)$	$k_7 = \mathsf{known}(c(b,a))$	$k_5 \sqcap k_6 \sqcap k_5 = \mathsf{t}$	$k_7 = \mathsf{t} \rightarrow k_6 = k_4$
$k_4 = \mathsf{known}(f)$			

(We ignored Con_0*.) The contradiction in* Bool *can be detected immediately.*

5 Conclusion

We showed that many extensions with recursive definitions (which can be seen as generalized homomorphism properties) satisfy locality conditions. This allows us to reduce the task of reasoning about the class of recursive functions we consider to reasoning in the underlying theory of data structures (possibly combined with the theories attached to the co-domains of the additional functions). We illustrated the ideas on several examples (including one inspired from cryptography). The main advantage of the method we use consists in the fact that it has the potential of completely separating the task of reasoning about the recursive definitions from the task of reasoning about the underlying data structures. We believe that these ideas will make the automatic verification of certain properties of recursive programs or of cryptographic protocols much easier, and we plan to make a detailed study of applications to cryptography in future work. An implementation of the method for hierarchical reasoning in local theory extensions is available at `www.mpi-inf.mpg.de/~ihlemann/software/index.html` (cf. also [12]). In various test runs it turned out to be extremely efficient, and can be used as a decision procedure for local theory extensions. We plan to extend the program to handle the theory extensions considered in this paper; we expect that this will not pose any problems. There are other classes of bridging functions – such as, for instance, cardinality functions for finite sets and measure functions for subsets of $\mathbb{R}$ (for instance intervals) – which turn out to satisfy similar locality properties. We plan to present such phenomena in a separate paper.

Acknowledgments. Many thanks to the referees for their helpful comments. This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, `www.avacs.org`).

References

1. Armando, A., Ranise, S., Rusinowitch, M.: A rewriting approach to satisfiability procedures. Information and Computation 183(2), 140–164 (2003)
2. Barrett, C., Shikanian, I., Tinelli, C.: An abstract decision procedure for satisfiability in the theory of inductive data types. Journal on Satisfiability, Boolean Modeling and Computation 3, 1–17 (2007)
3. Bonacina, M.P., Echenim, M.: Rewrite-based decision procedures. Electronic Notes in Theoretical Computer Science 174(11), 27–45 (2007)
4. Comon-Lundh, H., Treinen, R.: Easy intruder deductions. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 225–242. Springer, Heidelberg (2004)
5. Comon-Lundh, H.: Challenges in the automated verification of security protocols. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS, vol. 5195, pp. 396–409. Springer, Heidelberg (2008)
6. Delaune, S.: Easy intruder deduction problems with homomorphisms. Information Processing Letters 97(6), 213–218 (2006)
7. Ganzinger, H.: Relating semantic and proof-theoretic concepts for polynomial time decidability of uniform word problems. In: Sixteenth Annual IEEE Symposium on Logic in Computer Science, Boston, MA, USA, pp. 81–90. IEEE Computer Society, Los Alamitos (2001)
8. Ganzinger, H., Sofronie-Stokkermans, V., Waldmann, U.: Modular proof systems for partial functions with Evans equality. Information and Computation 204(10), 1453–1492 (2006)
9. Givan, R., McAllester, D.: New results on local inference relations. In: Principles of Knowledge Representation and reasoning: Proceedings of the Third International Conference (KR 1992), pp. 403–412. Morgan Kaufmann Press, San Francisco (1992)
10. Givan, R., McAllester, D.A.: Polynomial-time computation via local inference relations. ACM Transactions on Computational Logic 3(4), 521–541 (2002)
11. Ihlemann, C., Jacobs, S., Sofronie-Stokkermans, V.: On local reasoning in verification. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 265–281. Springer, Heidelberg (2008)
12. Ihlemann, C., Sofronie-Stokkermans, V.: System description. H-PiLOT. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 131–139. Springer, Heidelberg (2009)
13. Oppen, D.C.: Reasoning about recursively defined data structures. Journal of the ACM 27(3), 403–411 (1980)
14. Sofronie-Stokkermans, V.: Hierarchic reasoning in local theory extensions. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 219–234. Springer, Heidelberg (2005)
15. Sofronie-Stokkermans, V.: Hierarchical and modular reasoning in complex theories: The case of local theory extensions. In: Konev, B., Wolter, F. (eds.) FroCos 2007. LNCS, vol. 4720, pp. 47–71. Springer, Heidelberg (2007)

16. Sofronie-Stokkermans, V., Ihlemann, C.: Automated reasoning in some local extensions of ordered structures. Journal of Multiple-Valued Logics and Soft Computing 13(4-6), 397–414 (2007)
17. Sofronie-Stokkermans, V.: Efficient hierarchical reasoning about functions over numerical domains. In: Dengel, A.R., Berns, K., Breuel, T.M., Bomarius, F., Roth-Berghofer, T.R. (eds.) KI 2008. LNCS (LNAI), vol. 5243, pp. 135–143. Springer, Heidelberg (2008)
18. Zhang, T., Sipma, H., Manna, Z.: Decision procedures for term algebras with integer constraints. Information and Computation 204(10), 1526–1574 (2006)

Axiom Pinpointing in Lightweight Description Logics via Horn-SAT Encoding and Conflict Analysis*

Roberto Sebastiani and Michele Vescovi

DISI, Università di Trento, Italy
{rseba,vescovi}@disi.unitn.it

Abstract. The recent quest for tractable logic-based languages arising from the field of bio-medical ontologies has raised a lot of attention on *lightweight* (i.e. less expressive but tractable) description logics, like $\mathcal{EL}$ and its family. To this extent, automated reasoning techniques in these logics have been developed for computing not only concept subsumptions, but also to pinpoint the set of axioms causing each subsumption. In this paper we build on previous work from the literature and we propose and investigate a simple and novel approach for axiom pinpointing for the logic $\mathcal{EL}^+$. The idea is to encode the classification of an ontology into a Horn propositional formula, and to exploit the power of Boolean Constraint Propagation and Conflict Analysis from modern SAT solvers to compute concept subsumptions and to perform axiom pinpointing. A preliminary empirical evaluation confirms the potential of the approach.

1 Motivations and Goals

The recent quest for tractable logic-based languages arising from the field of bio-medical ontologies has attracted a lot of attention on *lightweight* (i.e. less expressive but tractable) description logics, like $\mathcal{EL}$ and its family [1,3,5,2]. In particular, the logic $\mathcal{EL}^+$ [3,5] extends $\mathcal{EL}$ and is of particular relevance due to its algorithmic properties and due to its capability of expressing several important and widely-used bio-medical ontologies, such as SNOMED-CT [16], NCI [15], GENEONTOLOGY [7] and the majority of GALEN [12]. In fact in $\mathcal{EL}^+$ not only standard logic problems such as *concept subsumption* (e.g., "is `Amputation-of-Finger` a subconcept of `Amputation-of-Arm` in the ontology SNOMED-CT?" [6]), but also more sophisticated logic problems such as *axiom pinpointing* (e.g., "Find a minimal set of axioms in SNOMED-CT which are responsible of the fact that `Amputation-of-Finger` is a subconcept of `Amputation-of-Arm`" [6]) are tractable. Importantly, the problem of axiom pinpointing in $\mathcal{EL}^+$ is of great interest for debugging complex bio-medical ontologies (see, e.g., [6]). To this extent, the problems of concept subsumption and axiom pinpointing in $\mathcal{EL}^+$ have been thoroughly investigated, and efficient algorithms

* The first author is partly supported by SRC under GRC Custom Research Project 2009-TJ-1880 WOLFLING, and by MIUR under PRIN project 20079E5KM8_002.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 84–99, 2009.

for these two functionalities have been implemented and tested with success on large ontologies, including SNOMED-CT (see e.g. [3,5,6]).

In this paper we build on previous work from the literature of $\mathcal{EL}^+$ reasoning [3,5,6] and of SAT/SMT [11,17,8,9], and propose a simple and novel approach for (concept subsumption and) axiom pinpointing in $\mathcal{EL}^+$ (and hence in its sub-logics $\mathcal{EL}$ and $\mathcal{ELH}$). In a nutshell, the idea is to generate *polynomial-size* Horn propositional formulas representing part or all the deduction steps performed by the classification algorithms of [3,5], and to manipulate them by exploiting the functionalities of modern conflict-driven SAT/SMT solvers —like *Boolean Constraint Propagation (BCP)* [11], *conflict analysis under assumptions* [11,8], and *all-SMT* [9]. In particular, we show that from an ontology $\mathcal{T}$ it is possible to generate in polynomial time Horn propositional formulas $\phi_{\mathcal{T}}$, $\phi_{\mathcal{T}}^{one}$ and $\phi_{\mathcal{T}(po)}^{all}$ of increasing size s.t., for every pair of primitive concepts C_i, D_i:

(i) concept subsumption is performed by one run of BCP on $\phi_{\mathcal{T}}$ or $\phi_{\mathcal{T}}^{one}$;
(ii) *one* (non-minimal) set of axioms responsible for the derivation of $C_i \sqsubseteq_{\mathcal{T}} D_i$ (nMinA) is computed by one run of BCP and conflict analysis on $\phi_{\mathcal{T}}^{one}$ or $\phi_{\mathcal{T}(po)}^{all}$;
(iii) *one minimal* such set (MinA) is computed by iterating process (ii) on $\phi_{\mathcal{T}(po)}^{all}$ for an amount of times up-to-linear in the size of the first nMinA found;
(iv) the same task of (iii) can also be computed by iteratively applying process (ii) on an up-to-linear sequence of increasingly-smaller formulas $\phi_{\mathcal{T}}^{one}$,$\phi_{\mathcal{S}_1}^{one}$,...$\phi_{\mathcal{S}_k}^{one}$;
(v) *all* MinAs can be enumerated by means of all-SMT techniques on $\phi_{\mathcal{T}(po)}^{all}$, using step (iii) as a subroutine.

It is worth noticing that (i) and (ii) are instantaneous even with huge $\phi_{\mathcal{T}}$, $\phi_{\mathcal{T}}^{one}$ and $\phi_{\mathcal{T}(po)}^{all}$, and that (v) requires building a *polynomial-size* formula $\phi_{\mathcal{T}(po)}^{all}$, in contrast to the exponential-size formula required by the *all-MinAs* process of [5].

We have implemented a prototype tool and performed a preliminary empirical evaluation on the available ontologies, whose results confirm the potential of our novel approach. For lack of space we omit many details and optimizations of our procedures, which can be found in an extended version of this paper [14].

Content. In §2 we provide the necessary background on $\mathcal{EL}^+$ reasoning and on conflict-driven SAT solving; in §3 we present our SAT-based procedures for concept subsumption, one-MinA extraction and all-MinAs enumeration; in §4 we discuss our techniques and compare them with those in [5]; in §5 we present our preliminary empirical evaluation, in §6 we draw some conclusions and outline directions for future research.

2 Background

2.1 Classification, Subsumption and Axiom Pinpointing in $\mathcal{EL}^+$

The Logic $\mathcal{EL}^+$. The description logic $\mathcal{EL}^+$ belongs to the $\mathcal{EL}$ family, a group of lightweight description logics which allow for conjunctions, existential restrictions and support TBox of GCIs (general concept inclusions) [3]; $\mathcal{EL}^+$ extends $\mathcal{EL}$ adding *complex role inclusion axioms*. In more details, the *concept descriptions* in $\mathcal{EL}^+$ are inductively defined through the constructors listed in the upper

half of Table 1, starting from a set of *primitive concepts* and a set of *primitive roles*. (We use the uppercase letters X, X_i, Y, Y_i, to denote generic concepts, the uppercase letters C, C_i, D, D_i, E, E_i to denote concept names and the lowercase letters r, r_i, s to denote role names.) An $\mathcal{EL}^+$ *TBox* (or *ontology*) is a finite set of general concept inclusion (GCI) and role inclusion (RI) axioms as defined in the lower half of Table 1. Given a TBox $\mathcal{T}$, we denote with $\mathsf{PC}_\mathcal{T}$ the set of the *primitive concepts* for $\mathcal{T}$, i.e. the smallest set of concepts containing: (i) the top concept $\top$; (ii) all concept names used in $\mathcal{T}$. We denote with $\mathsf{PR}_\mathcal{T}$ the set of the *primitive roles* for $\mathcal{T}$, i.e. the set of all the role names used in $\mathcal{T}$. We use the expression $X \equiv Y$ as an abbreviation of the two GCIs $X \sqsubseteq Y$ and $Y \sqsubseteq X$.

The semantics of $\mathcal{EL}^+$ is defined in terms of *interpretations*. An interpretation $\mathcal{I}$ is a couple $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$, where $\Delta^\mathcal{I}$ is the domain, i.e. a non-empty set of individuals, and $\cdot^\mathcal{I}$ is the interpretation function which maps each concept name C to a set $C^\mathcal{I} \subseteq \Delta^\mathcal{I}$ and maps each role name r to a binary relation $r^\mathcal{I} \subseteq \Delta^\mathcal{I} \times \Delta^\mathcal{I}$. In the right-most column of Table 1 the inductive extensions of $\cdot^\mathcal{I}$ to arbitrary concept descriptions are defined. An interpretation $\mathcal{I}$ is a *model* of a given TBox $\mathcal{T}$ if and only if the conditions in the Semantics column of Table 1 are respected for every GCI and RI axiom in $\mathcal{T}$. A TBox $\mathcal{T}'$ is a *conservative extension* of the TBox $\mathcal{T}$ if every model of $\mathcal{T}'$ is also a model of $\mathcal{T}$, and every model of $\mathcal{T}$ can be extended to a model of $\mathcal{T}'$ by appropriately defining the interpretations of the additional concept and role names.

A concept Y *subsumes* a concept X w.r.t. the TBox $\mathcal{T}$, written $X \sqsubseteq_\mathcal{T} Y$, iff $X^\mathcal{I} \subseteq Y^\mathcal{I}$ for every model $\mathcal{I}$ of $\mathcal{T}$. The computation of all subsumption relations between concept names occurring in $\mathcal{T}$ is called *classification* of $\mathcal{T}$. Subsumption and classification in $\mathcal{EL}^+$ can be performed in polynomial time [1,5].

Normalization. In $\mathcal{EL}^+$ it is convenient to establish and work with a *normal form* of the input problem, which helps to make explanations, proofs, reasoning rules and algorithms simpler and more general. Usually the following normal form for the $\mathcal{EL}^+$ TBoxes is considered [1,3,4,5]:

$$(C_1 \sqcap ... \sqcap C_k) \sqsubseteq D, \quad k \geq 1 \qquad C \sqsubseteq \exists r.D \qquad \exists r.C \sqsubseteq D \tag{1}$$

$$r_1 \circ \cdots \circ r_n \sqsubseteq s, \quad n \geq 1 \tag{2}$$

s.t. $C_1, ..., C_k, D \in \mathsf{PC}_\mathcal{T}$ and $r_1, ..., r_n, s \in \mathsf{PR}_\mathcal{T}$. A TBox $\mathcal{T}$ can be turned into a normalized TBox $\mathcal{T}'$ that is a conservative extension of $\mathcal{T}$ [1], by introducing new concept names. In a nutshell, normalization consists in substituting all instances

Table 1. Syntax and semantics of $\mathcal{EL}^+$

	Syntax	Semantics
top	$\top$	$\Delta^\mathcal{I}$
conjunction	$X \sqcap Y$	$X^\mathcal{I} \cap Y^\mathcal{I}$
existential restriction	$\exists r.X$	$\{x \in \Delta^\mathcal{I} \mid \exists y \in \Delta^\mathcal{I} : (x,y) \in r^\mathcal{I} \wedge y \in X^\mathcal{I}\}$
general concept inclusion	$X \sqsubseteq Y$	$X^\mathcal{I} \subseteq Y^\mathcal{I}$
role inclusion	$r_1 \circ \cdots \circ r_n \sqsubseteq s$	$r_1^\mathcal{I} \circ \cdots \circ r_k^\mathcal{I} \subseteq s^\mathcal{I}$

of complex concepts of the forms $\exists r.C$ and $C_1 \sqcap ... \sqcap C_k$ with fresh concept names (namely, C' and C''), and adding the axioms $C' \sqsubseteq \exists r.C$ [resp. $\exists r.C \sqsubseteq C'$] and $C'' \sqsubseteq C_1, ..., C'' \sqsubseteq C_k$ [resp. $(C_1 \sqcap ... \sqcap C_k) \sqsubseteq C''$] for every substitution in the right [resp. left] part of an axiom. This transformation can be done in linear time and the size of $\mathcal{T}'$ is linear w.r.t. that of $\mathcal{T}$ [1]. We call *normal concept* of a normal TBox $\mathcal{T}'$ every non-conjunctive concept description occurring in the concept inclusions of $\mathcal{T}'$; we call $\mathsf{NC}_{\mathcal{T}'}$ the set of all the normal concepts of $\mathcal{T}'$.

Concept Subsumption in $\mathcal{EL}^+$. Given a normalized TBox $\mathcal{T}$ over the set of primitive concepts $\mathsf{PC}_{\mathcal{T}}$ and the set of primitive roles $\mathsf{PR}_{\mathcal{T}}$, the subsumption algorithm for $\mathcal{EL}^+$ [5] generates and extends a set $\mathcal{A}$ of *assertions* through the completion rules defined in Table 2. (By "*assertion*" we mean every known or deduced subsumption relation between normal concepts of the TBox $\mathcal{T}$.) The algorithm starts with the initial set $\mathcal{A} = \{a_i \in \mathcal{T} | a_i \text{ is a GCI}\} \cup \{C \sqsubseteq C | C \in \mathsf{PC}_{\mathcal{T}}\} \cup \{C \sqsubseteq \top | C \in \mathsf{PC}_{\mathcal{T}}\}$ and extends $\mathcal{A}$ using the rules of Table 2 until no more assertions can be added. (Notice that a rule is applied only if it extends $\mathcal{A}$.)

In [1] the soundness and completeness of the algorithm are proved, together with the fact that the algorithm terminates after polynomially-many rule applications, each of which can be performed in polynomial time.

Once a complete classification of the normalized TBox is computed and stored in some ad-hoc data structure, if $C, D \in \mathsf{PC}_{\mathcal{T}}$, then $C \sqsubseteq_{\mathcal{T}} D$ iff the pair C, D can be retrieved from the latter structure. The problem of computing $X \sqsubseteq_{\mathcal{T}} Y$ s.t. $X, Y \notin \mathsf{PC}_{\mathcal{T}}$ can be reduced to that of computing $C \sqsubseteq_{\mathcal{T} \cup \{C \sqsubseteq X, Y \sqsubseteq D\}} D$, s.t. C and D are two new concept names.

Axiom Pinpointing in $\mathcal{EL}^+$. We consider $C_i, D_i \in \mathsf{PC}_{\mathcal{T}}$ s.t. $C_i \sqsubseteq_{\mathcal{T}} D_i$. We call $\mathcal{S}$ s.t. $\mathcal{S} \subseteq \mathcal{T}$ a (possibly non-minimal) *axiom set for $\mathcal{T}$ wrt. $C_i \sqsubseteq D_i$*, written *nMinA*, if $C_i \sqsubseteq_{\mathcal{S}} D_i$; we call an nMinA $\mathcal{S}$ a *minimal axiom set for $C_i \sqsubseteq D_i$*, written *MinA*, if $C_i \not\sqsubseteq_{\mathcal{S}'} D_i$ for every $\mathcal{S}'$ s.t. $\mathcal{S}' \subset \mathcal{S}$.

Baader et al. [5] proposed a technique for computing all MinAs for $\mathcal{T}$ wrt. $C_i \sqsubseteq_{\mathcal{T}} D_i$, which is based on building from a classification of $\mathcal{T}$ a *pinpointing formula* (namely $\Phi^{C_i \sqsubseteq_{\mathcal{T}} D_i}$), which is a monotone propositional formula on the set of propositional variables $\mathcal{P}_{\mathcal{T}} \stackrel{\text{def}}{=} \{s_{[ax_j]} | ax_j \in \mathcal{T}\}$ s.t., for every $\mathcal{O} \subseteq \mathcal{T}$, $\mathcal{O}$ is a MinA wrt. $C_i \sqsubseteq_{\mathcal{T}} D_i$ iff $\{s_{[ax_k]} | ax_k \in \mathcal{O}\}$ is a minimal valuation of $\Phi^{C_i \sqsubseteq_{\mathcal{T}} D_i}$. Thus,

Table 2. Completion rules of the concept subsumption algorithm for $\mathcal{EL}^+$. A rule reads as follows: if the assertions/axioms in the left column belong to $\mathcal{A}$, the GCI/RI of the central column belongs to $\mathcal{T}$, and the assertion of the right column is not already in $\mathcal{A}$, then the assertion of the right column is added to $\mathcal{A}$.

Subsumption assertions ($\ldots \in \mathcal{A}$)	**TBox's axioms** ($\ldots \in \mathcal{T}$)	... added to $\mathcal{A}$
$X \sqsubseteq C_1$, $X \sqsubseteq C_2$, ... $X \sqsubseteq C_k$ $\quad k \geq 1$	$C_1 \sqcap \cdots \sqcap C_k \sqsubseteq D$	$X \sqsubseteq D$
$X \sqsubseteq C$	$C \sqsubseteq \exists r.D$	$X \sqsubseteq \exists r.D$
$X \sqsubseteq \exists r.E$, $E \sqsubseteq C$	$\exists r.C \sqsubseteq D$	$X \sqsubseteq D$
$X \sqsubseteq \exists r.D$	$r \sqsubseteq s$	$X \sqsubseteq \exists s.D$
$X \sqsubseteq \exists r_1.E_1$, ..., $E_{n-1} \sqsubseteq \exists r_n.D$ $\quad n \geq 1$	$r_1 \circ \cdots \circ r_n \sqsubseteq s$	$X \sqsubseteq \exists s.D$

the all-MinAs algorithm in [5] consists of (i) building $\Phi^{C_i \sqsubseteq_\mathcal{T} D_i}$ and (ii) computing all minimal valuations of $\Phi^{C_i \sqsubseteq_\mathcal{T} D_i}$. According to [5], however, this algorithm has serious limitations in terms of complexity: first, the algorithm for generating $\Phi^{C_i \sqsubseteq_\mathcal{T} D_i}$ requires intermediate logical checks, each of them involving the solution of an NP-complete problem; second, the size of $\Phi^{C_i \sqsubseteq_\mathcal{T} D_i}$ can be exponential wrt. that of $\mathcal{T}$. More generally, [5] proved also that there is no output-polynomial algorithm for computing all MinAs (unless P=NP). (To the best of our knowledge, there is no publicly-available implementation of the all-MinAs algorithm above.) Consequently, [5] concentrated the effort on finding polynomial algorithms for finding *one* MinA at a time, proposing a linear-search minimization algorithm which allowed for finding MinAs for FULL-GALEN efficiently. This technique was further improved in [6] by means of a binary-search minimization algorithm, and by a novel algorithm exploiting the notion of *reachability-modules*, which allowed to find efficiently MinAs for the much bigger SNOMED-CT ontology. We refer the readers to [5,6] for a detailed description.

2.2 Basics on Conflict-Driven SAT Solving

Notation. We assume the standard syntactic and semantic notions of propositional logic (including the standard notions of formula, atom, literal, CNF formula, Horn formula, truth assignment, clause, unit clause). We represent a truth assignment μ as a conjunction of literals $\bigwedge_i l_i$ (or analogously as a set of literals $\{l_i\}_i$) with the intended meaning that a positive [resp. negative] literal p_i means that p_i is assigned to true [resp. false]. Notationally, we often write clauses as implications: "$(\bigwedge_i l_i) \rightarrow (\bigvee_j l_j)$" for "$\bigvee_i \neg l_i \vee \bigvee_j l_j$"; also, if η is a conjunction of literals $\bigwedge_i l_i$, we write $\neg\eta$ for the clause $\bigvee_i \neg l_i$, and vice versa.

Conflict-Driven SAT Solving. The schema of a modern conflict-driven DPLL SAT solver is shown in Figure 1 [11,17]. The propositional formula φ is in CNF; the assignment μ is initially empty, and it is updated in a stack-based manner.

In the main loop, `decide_next_branch`(φ, μ) (line `12.`) chooses an unassigned literal l from φ according to some heuristic criterion, and adds it to μ.

```
1.      SatValue DPLL (formula φ, assignment μ)
2.          while (1)
3.              while (1)
4.                  status = bcp(φ, μ);
5.                  if (status == sat)
6.                      return sat;
7.                  else if (status == conflict)
8.                      blevel = analyze_conflict(φ, μ);
9.                      if (blevel == 0) return unsat;
10.                     else backtrack(blevel,φ, μ);
11.                 else break;
12.             decide_next_branch(φ, μ);
```

Fig. 1. Schema of a conflict-driven DPLL SAT solver

(This operation is called *decision*, l is called *decision literal* and the number of decision literals in μ after this operation is called the *decision level* of l.) In the inner loop, $\texttt{bcp}(\varphi, \mu)$ iteratively deduces literals l from the current assignment and updates φ and μ accordingly; this step is repeated until either μ satisfies φ, or μ falsifies φ, or no more literals can be deduced, returning sat, conflict and unknown respectively. In the first case, DPLL returns sat. In the second case, $\texttt{analyze\_conflict}(\varphi, \mu)$ detects the subset η of μ which caused the conflict (*conflict set*) and the decision level blevel to backtrack. (This process is called *conflict analysis*, and is described in more details below.) If blevel is 0, then a conflict exists even without branching, so that DPLL returns unsat. Otherwise, $\texttt{backtrack}(\texttt{blevel}, \varphi, \mu)$ adds the *blocking clause* $\neg\eta$ to φ (*learning*) and backtracks up to blevel (*backjumping*), popping out of μ all literals whose decision level is greater than blevel, and updating φ accordingly. In the third case, DPLL exits the inner loop, looking for the next decision.

bcp is based on *Boolean Constraint Propagation. (BCP)*, that is, the iterative application of *unit propagation*: if a unit clause l occurs in φ, then l is added to μ, all negative occurrences of l are declared false and all clauses with positive occurrences of l are declared satisfied. Current SAT solvers include extremely fast implementations of bcp based on the *two-watched-literal scheme* [11]. Notice that a complete run of bcp requires an amount of steps which is at most linear in the number of clauses containing the negation of some of the propagated literals.

analyze_conflict works as follows (see, e.g., [11,17]). Each literal is tagged with its decision level, that is, the literal corresponding to the nth decision and the literals derived by unit-propagation after that decision are labeled with n; each non-decision literal l in μ is also tagged by a link to the clause ψ_l causing its unit-propagation (called the *antecedent clause* of l). When a clause ψ is falsified by the current assignment —in which case we say that a *conflict* occurs and ψ is the *conflicting clause*— a *conflict clause* ψ' is computed from ψ s.t. ψ' contains only one literal l_u which has been assigned at the last decision level. ψ' is computed starting from $\psi' = \psi$ by iteratively resolving ψ' with the antecedent clause ψ_l of some literal l in ψ' (typically the last-assigned literal in ψ', see [17]), until some stop criterion is met. E.g., with the *Decision Scheme*, ψ' must contain only decision literals, including the last-assigned one.

If φ is a Horn formula, then one single run of bcp is sufficient to decide the satisfiability of φ. In fact, if $\texttt{bcp}(\varphi, \{\})$ returns conflict, then φ is unsatisfiable; otherwise φ is satisfiable because, since all unit clauses have been removed from φ, all remaining clauses contain at least one negative literal, so that assigning all unassigned literals to false satisfies φ.

Conflict-Driven SAT Solving Under Assumptions. The schema in Figure 1 can be adapted to check also the satisfiability of a CNF propositional formula φ *under a set of assumptions* $\mathcal{L} \stackrel{def}{=} \{l_1, ..., l_k\}$. (From a purely-logical viewpoint, this corresponds to check the satisfiability of $\bigwedge_{l_i \in \mathcal{L}} l_i \wedge \varphi$.) This works as follows: $l_1, ..., l_k$ are initially assigned to true, they are tagged as decision literals and added to μ, then the decision level is reset to 0 and DPLL enters the external loop. If $\bigwedge_{l_i \in \mathcal{L}} l_i \wedge \varphi$ is consistent, then DPLL returns sat; otherwise, DPLL

eventually backtracks up to level 0 and then stops, returning conflict. Importantly, if `analyze_conflict` uses the Decision Scheme mentioned above, then the final conflict clause will be in the form $\bigvee_{l_j \in \mathcal{L}'} \neg l_j$ s.t. $\mathcal{L}'$ is the (possibly much smaller) subset of $\mathcal{L}$ which actually caused the inconsistency revealed by the SAT solver (i.e., s.t. $\bigwedge_{l_j \in \mathcal{L}'} l_j \wedge \varphi$ is inconsistent). In fact, at the very last branch, `analyze_conflict` will iteratively resolve the conflicting clause with the antecedent clauses of the unit-propagated literals until only decision literals are left: since this conflict has caused a backtrack up to level 0, these literals are necessarily all part of $\mathcal{L}$.

This technique is very useful in some situations. First, sometimes one needs checking the satisfiability of a (possibly very big) formula φ under many different sets of assumptions $\mathcal{L}_1, ..., \mathcal{L}_N$. If this is the case, instead of running DPLL on $\bigwedge_{l_i \in \mathcal{L}_j} l_i \wedge \varphi$ for every $\mathcal{L}_j$ —which means parsing the formulas and initializing DPLL from scratch each time— it is sufficient to parse φ and initialize DPLL only once, and run the search under the different sets of assumptions $\mathcal{L}_1, ..., \mathcal{L}_N$. This is particularly important when parsing and initialization times are relevant wrt. solving times. In particular, if φ is a Horn formula, solving φ under assumptions requires only one run of bcp, whose computational cost depends linearly only on the clauses where the unit-propagated literals occur.

Second, this technique can be used in association with the use of *selector variables*: all the clauses ψ_i of φ can be substituted by the corresponding clauses $s_i \rightarrow \psi_i$, all s_is being fresh variables, which are initially assumed to be true (i.e., $\mathcal{L} = \{s_i \mid \psi_i \in \varphi\}$). If φ is unsatisfiable, then the final conflict clause will be of the form $\bigvee_{s_j \in \mathcal{L}'} \neg s_j$, s.t. $\{\psi_j \mid s_j \in \mathcal{L}'\}$ is the actual subset of clauses which caused the inconsistency of φ. This technique is used to compute unsatisfiable cores of CNF propositional formulas [10].

3 Axiom Pinpointing via Horn SAT and Conflict Analysis

We assume that $\mathcal{T}$ is the result of a normalization process, as described in §2.1. (We will consider the issue of normalization at the end of §3.2.)

3.1 Classification and Concept Subsumption via Horn SAT Solving

We consider the problem of concept subsumption. We build a Horn propositional formula $\phi_\mathcal{T}$ representing the classification of the input ontology $\mathcal{T}$. A basic encoding works as follows. For every normalized concept X in $\mathsf{NC}_\mathcal{T}$ we introduce exactly one uniquely-associated fresh Boolean variable $p_{[X]}$. We initially set $\phi_\mathcal{T}$ to the empty set of clauses. We run the classification algorithm of §2.1: for every non-trivial[1] axiom or assertion a_i of the form (1) which is added to $\mathcal{A}$, we add to $\phi_\mathcal{T}$ one clause $\mathcal{EL}^+2sat(a_i)$ of the form

$$(p_{[C_1]} \wedge ... \wedge p_{[C_k]}) \rightarrow p_{[D]}, \quad k \geq 1 \qquad p_{[C]} \rightarrow p_{[\exists r.D]} \qquad p_{[\exists r.C]} \rightarrow p_{[D]} \tag{3}$$

[1] We do not encode axioms of the form $C \sqsubseteq C$ and $C \sqsubseteq \top$ because they generate valid clauses $p_{[C]} \rightarrow p_{[C]}$ and $p_{[C]} \rightarrow \top$.

respectively. Notice that (3) are non-unit Horn clauses with one positive literal (hereafter *definite* Horn clauses). It follows straightforwardly that $C \sqsubseteq_{\mathcal{T}} D$ if and only if the Horn formula $\phi_{\mathcal{T}} \wedge p_{[C]} \wedge \neg p_{[D]}$ is unsatisfiable, for every pair of concepts C, D in $\mathsf{PC}_{\mathcal{T}}$. In fact, by construction, the clause $p_{[C]} \rightarrow p_{[D]}$ is in $\phi_{\mathcal{T}}$ if and only if $C \sqsubseteq_{\mathcal{T}} D$. Notice that $\phi_{\mathcal{T}}$ is polynomial wrt. the size of $\mathcal{T}$, since the algorithm of §2.1 terminates after a polynomial number of rule applications. A more compact encoding is described in [14].

Once $\phi_{\mathcal{T}}$ has been generated, in order to perform concept subsumption we exploit the techniques of conflict-driven SAT solving under assumptions described in §2.2: once $\phi_{\mathcal{T}}$ is parsed and `DPLL` is initialized, each subsumption query $C_i \sqsubseteq_{\mathcal{T}} D_i$ corresponds to solving $\phi_{\mathcal{T}}$ under the assumption list $\mathcal{L}_i \stackrel{\text{def}}{=} \{\neg p_{[D_i]}, p_{[C_i]}\}$. This corresponds to one run of `bcp`: since $\phi_{\mathcal{T}}$ contains the clause $p_{[C_i]} \rightarrow p_{[D_i]}$, `bcp` stops as soon as $\neg p_{[D_i]}$ and $p_{[C_i]}$ are unit-propagated.

3.2 Computing Single and All MinAs via Conflict Analysis

We consider the general problem of generating MinAs. We build another Horn propositional formula $\phi^{all}_{\mathcal{T}}$ representing the complete classification DAG of the input normalized ontology $\mathcal{T}$.[2] The size of $\phi^{all}_{\mathcal{T}}$ is polynomial wrt. that of $\mathcal{T}$.

Building the Formula $\phi^{all}_{\mathcal{T}}$. For every normalized concept X in $\mathsf{NC}_{\mathcal{T}}$ we introduce exactly one uniquely-associated fresh Boolean variable $p_{[X]}$; further (*selector*) Boolean variables will be introduced, through the steps of the algorithm, to uniquely represent axioms and assertions. We initially set $\phi^{all}_{\mathcal{T}}$ to the empty set of clauses. Then we run an extended version of the classification algorithm of §2.1:

1. for every RI axiom a_i we introduce the axiom selector variable $s_{[a_i]}$; for every GCI axiom a_i of the form $C \sqsubseteq C$ or $C \sqsubseteq \top$, $s_{[a_i]}$ is the "true" constant $\top$;
2. for every non-trivial GCI axiom a_i we add to $\phi^{all}_{\mathcal{T}}$ a clause of the form

$$s_{[a_i]} \rightarrow \mathcal{EL}^+2sat(a_i) \tag{4}$$

   s.t. $s_{[a_i]}$ is the axiom selector variable for a_i and $\mathcal{EL}^+2sat(a_i)$ is the encoding in (3);
3. for every application of a rule (namely r) generating some assertion $gen(r)$ (namely a_i) which was not yet present in $\mathcal{A}$ (and thus adding a_i to $\mathcal{A}$), we add to $\phi^{all}_{\mathcal{T}}$ a clause (4) and a clause of the form

$$\Big(\bigwedge_{a_j \in\ ant(r)} s_{[a_j]}\Big) \rightarrow s_{[a_i]} \tag{5}$$

   s.t. $s_{[a_i]}$ (that is $s_{[gen(r)]}$) is the selector variable for a_i and $ant(r)$ are the *antecedents* of a_i wrt. rule r (that is, the assertions and the RI or GCI axiom in the left and central columns of Table 2 for rule r respectively);

[2] Here "complete" means "including also the rule applications generating already-generated assertions".

4. for every application of a rule (namely r) generating some assertion $gen(r)$ (namely a_i) which was already present in $\mathcal{A}$ (and thus not adding a_i to $\mathcal{A}$), we add to $\phi^{all}_{\mathcal{T}}$ only a clause of the form (5).

In order to ensure termination, we perform step 3. and 4. in a queue-based manner, which assures that every possible distinct (i.e. with different antecedents) rule application is applied only once. Following this idea, in [14] we show that the extended algorithm requires a polynomial amount of steps wrt. the size of $\mathcal{T}$ and that $\phi^{all}_{\mathcal{T}}$ is polynomial in the size of $\mathcal{T}$.

Notice that (4) and (5) are definite Horn clauses since all (3) are definite Horn clauses. (We call (4) and (5) *assertion clauses* and *rule clauses* respectively.) Notice also that step 4. is novel wrt. the classification algorithm of §2.1.

It follows straighforwardly that, for every $\mathcal{S} \subseteq \mathcal{T}$ and for every pair of concepts C, D in $\mathsf{PC}_{\mathcal{T}}$, $C \sqsubseteq_{\mathcal{S}} D$ if and only if $\phi^{all}_{\mathcal{T}} \wedge \bigwedge_{a_i \in \mathcal{S}} s_{[a_i]} \wedge p_{[C]} \wedge \neg p_{[D]}$ is unsatisfiable. In fact, $C \sqsubseteq_{\mathcal{S}} D$ if and only if there exists a sequence or rule applications $r_1, ..., r_k$ generating $C \sqsubseteq D$ from $\mathcal{S}$. If (and only if) this is the case, by construction $\phi^{all}_{\mathcal{T}}$ contains the clause $s_{[C \sqsubseteq D]} \rightarrow (p_{[C]} \rightarrow p_{[D]})$ (4) and all the clauses (5) corresponding to all rule applications $r \in \{r_1, ..., r_k\}$. This means that $\bigwedge_{a_i \in \mathcal{S}} s_{[a_i]} \wedge p_{[C]} \wedge \neg p_{[D]}$ forces the unit-propagation of $s_{[C \sqsubseteq D]}$, $p_{[C]}$ and $\neg p_{[D]}$, which falsify the clause (4) above. (See [14] for details.)

Computing One MinA. Once $\phi^{all}_{\mathcal{T}}$ is generated, in order to compute one MinA, we can exploit the techniques of conflict-driven SAT solving under assumptions described in §2.2. After $\phi^{all}_{\mathcal{T}}$ is parsed and `DPLL` is initialized, each query $C_i \sqsubseteq_{\mathcal{T}} D_i$ corresponds to solving $\phi^{all}_{\mathcal{T}}$ under the assumption list $\mathcal{L}_i \stackrel{\text{def}}{=} \{\neg p_{[D_i]}, p_{[C_i]}\} \cup \{s_{[a_i]} \mid a_i \in \mathcal{T}\}$. This corresponds to a single run of `bcp` and one run of `analyze_conflict`, whose cost depends linearly only on the clauses where the unit-propagated literals occur. (Actually, if `bcp` does not return conflict, then `sat` is returned without even performing conflict analysis.) If `bcp` returns conflict, as explained in §2.2, then `analyze_conflict` produces a conflict clause $\psi^{C_i,D_i}_{\mathcal{T}^*} \stackrel{\text{def}}{=} p_{[D_i]} \vee \neg p_{[C_i]} \vee \bigvee_{a_i \in \mathcal{T}^*} \neg s_{[a_i]}$ s.t. $\mathcal{T}^*$ is an nMinA wrt. $C_i \sqsubseteq_{\mathcal{T}} D_i$. In fact, the presence of both $\neg p_{[D_i]}$ and $p_{[C_i]}$ in $\mathcal{L}_i$ is necessary for causing the conflict, so that, due to the Decision Scheme, the conflict set necessarily contains both of them.

Notice that $\mathcal{T}^*$ may not be minimal. In order to minimize it, we can apply the SAT-based variant of the linear minimization algorithm of [5] in Figure 2. (We assume that $\phi^{all}_{\mathcal{T}}$ has been parsed and `DPLL` has been initialized, and that $\phi^{all}_{\mathcal{T}}$ has been solved under the assumption list $\mathcal{L}_i$ above, producing the conflict clause $\psi^{C_i,D_i}_{\mathcal{T}^*}$ and hence the nMinA $\mathcal{T}^*$; then `lin-extract-MinA`$_{DPLL}$`(`C_i, D_i`, `$\mathcal{T}^*$`, `$\phi^{all}_{\mathcal{T}}$`)` is invoked.) In a nutshell, the algorithm tries to remove one-by-one the axioms a_js in $\mathcal{T}^*$, each time checking whether the reduced axiom set $\mathcal{S} \setminus \{a_j\}$ is still such that $C_i \sqsubseteq_{S \setminus \{a_j\}} D_i$. As before, each call to `DPLLUnderAssumptions` requires only one run of `bcp`. This schema can be improved as follows: if `DPLLUnderAssumptions` performs also conflict analysis and returns (the conflict clause corresponding to) an nMinA $\mathcal{S}'$ s.t. $\mathcal{S}' \subset \mathcal{S} \setminus \{a_i\}$, then $\mathcal{S}$ is assigned to $\mathcal{S}'$ and all axioms in $(\mathcal{S} \setminus \{a_j\}) \setminus \mathcal{S}'$ will not be selected in next loops. As

an alternative choice, one can implement instead (a SAT-based version of) the binary-search variant of the minimization algorithm (see e.g. [6]).

It is important to notice that the formula $\phi^{all}_{\mathcal{T}}$ is never updated: in order to check $C_i \sqsubseteq_{S\setminus\{a_j\}} D_i$, it suffices to drop $s_{[a_j]}$ from the assumption list. The latter fact makes (the encoding of) the axiom a_j useless for `bcp` to falsify the clause encoding $C_i \sqsubseteq_{\mathcal{T}} D_i$, so that `DPLLUnderAssumptions` returns `unsat` if and only if a different falsifying chain of unit-propagations can be found, corresponding to a different sequence of rule applications generating $C_i \sqsubseteq_{\mathcal{T}} D_i$. Notice that this fact is made possible by step 4. of the encoding, which allows for encoding all alternative sequences of rule applications generating the same assertions.

We also notice that one straightforward variant to this technique, which is feasible since typically $|\mathcal{T}^*| \ll |\mathcal{T}|$, is to compute another formula $\phi^{all}_{\mathcal{T}^*}$ from scratch and to feed it to the algorithm of Figure 2 instead of $\phi^{all}_{\mathcal{T}}$.

One very important remark is in order. During pinpointing the only clause of type (4) in $\phi^{all}_{\mathcal{T}}$ which is involved in the conflict analysis process is $s_{[C_i \sqsubseteq_{\mathcal{T}} D_i]} \rightarrow (p_{[C_i]} \rightarrow p_{[D_i]})$, which reduces to the unit clause $\neg s_{[C_i \sqsubseteq_{\mathcal{T}} D_i]}$ after the unit-propagation of the assumption literals $\neg p_{[D_i]}, p_{[C_i]}$. Thus, one may want to decouple pinpointing from classification/subsumption, and produce a reduced "pinpointing-only" version of $\phi^{all}_{\mathcal{T}}$, namely $\phi^{all}_{\mathcal{T}(po)}$. The encoding of $\phi^{all}_{\mathcal{T}(po)}$ works like that of $\phi^{all}_{\mathcal{T}}$, except that no clause (4) is added to $\phi^{all}_{\mathcal{T}(po)}$. Thus each query $C_i \sqsubseteq_{\mathcal{T}} D_i$ corresponds to solving $\phi^{all}_{\mathcal{T}(po)}$ under the assumption list $\mathcal{L}_i \stackrel{\text{def}}{=} \{\neg s_{[C_i \sqsubseteq_{\mathcal{T}} D_i]}\} \cup \{s_{[a_i]} \mid a_i \in \mathcal{T}\}$, so that the algorithm for pinpointing is changed only in the fact that $\phi^{all}_{\mathcal{T}(po)}$ and $\{\neg s_{[C_i \sqsubseteq_{\mathcal{T}} D_i]}\}$ are used instead of $\phi^{all}_{\mathcal{T}}$ and $\{\neg p_{[D_i]}, p_{[C_i]}\}$ respectively. Thus, w.l.o.g. in the remaining part of this section we will reason using $\phi^{all}_{\mathcal{T}(po)}$ and $\{\neg s_{[C_i \sqsubseteq_{\mathcal{T}} D_i]}\}$. (The same results, however, can be obtained using $\phi^{all}_{\mathcal{T}}$ and $\{\neg p_{[D_i]}, p_{[C_i]}\}$ instead.)

Computing All MinAs. We describe a way of generating *all* MinAs wrt. $C_i \sqsubseteq_{\mathcal{T}} D_i$ from $\phi^{all}_{\mathcal{T}(po)}$ and $\{\neg s_{[C_i \sqsubseteq_{\mathcal{T}} D_i]}\}$. In a nutshell, the idea is to assume $\{\neg s_{[C_i \sqsubseteq_{\mathcal{T}} D_i]}\}$ and to enumerate all possible minimal truth assignments on the axiom selector variables in $\mathcal{P}_{\mathcal{T}} \stackrel{\text{def}}{=} \{s_{[ax_j]} \mid ax_j \in \mathcal{T}\}$ which cause the inconsistency of the formula $\phi^{all}_{\mathcal{T}(po)}$. This is implemented by means of a variant of the all-SMT technique in [9]. A naive version of this technique is described as follows.

We consider a propositional CNF formula φ on the variables in $\{s_{[C_i \sqsubseteq_{\mathcal{T}} D_i]}\} \cup \mathcal{P}_{\mathcal{T}}$. φ is initially set to $\top$. One top-level instance of `DPLL` (namely `DPLL1`) is used to enumerate a complete set of truth assignments $\{\mu_k\}_k$ on the axiom selector variables in $\mathcal{P}_{\mathcal{T}}$ which satisfy φ under the assumption of $\neg s_{[C_i \sqsubseteq_{\mathcal{T}} D_i]}$. Every time that a novel assignment μ_k is generated, $\{\neg s_{[C_i \sqsubseteq_{\mathcal{T}} D_i]}\} \cup \mu_k$ is passed to an ad-hoc "$\mathcal{T}$-solver" checking whether it causes the inconsistency of the formula $\phi^{all}_{\mathcal{T}(po)}$. If this is the case, then the $\mathcal{T}$-solver returns `conflict` and a minimal subset $\{\neg s_{[C_i \sqsubseteq_{\mathcal{T}} D_i]}\} \cup \{s_{[ax_j]} \mid ax_j \in \mathcal{T}^*_k\}$, s.t. $\mathcal{T}^*_k$ is a MinA, which caused such inconsistency. $\psi^*_k \stackrel{\text{def}}{=} s_{[C_i \sqsubseteq_{\mathcal{T}} D_i]} \vee \bigvee_{ax_j \in \mathcal{T}^*_k} \neg s_{[ax_j]}$ is then added to φ as a blocking clause and it is used as a conflict clause for driving next backjumping step. Otherwise,

```
AxiomSet lin-extract-MinA_DPLL(Concept C_i, D_i, AxiomSet T*, formula φ^all_T)
1.    S = T*;
2.    for each axiom a_j in T*
3.       L = {¬p_[D_i], p_[C_i]} ∪ {s_[a_i] | a_i ∈ S \ {a_j}};
4.       if (DPLLUnderAssumptions(φ^all_T, L) == unsat)
5.          S = S \ {a_j};
6.    return S;
```

Fig. 2. SAT-based variant of the linear MinA-extracting algorithm in [5]

$\mathcal{T}$-solver returns sat, and DPLL1 uses $s_{[C_i \sqsubseteq_\mathcal{T} D_i]} \vee \neg\mu_k$ as a "fake" conflict clause, which is added to φ as a blocking clause and is used as a conflict clause for driving next backjumping step. The process terminates when backtrack back-jumps to blevel zero. The set of all MinAs $\mathcal{T}_k^*$ are returned as output.

The $\mathcal{T}$-solver is the procedure described in the previous paragraph "Compute one MinA" (with $\phi^{all}_{\mathcal{T}(po)}$, $\{\neg s_{[C_i \sqsubseteq_\mathcal{T} D_i]}\}$ instead of $\phi^{all}_{\mathcal{T}}$, $\{\neg p_{[D_i]}, p_{[C_i]}\}$), using a second instance of DPLL, namely DPLL2. As before, we assume $\phi^{all}_{\mathcal{T}(po)}$ is parsed and DPLL2 is initialized only once, before the whole process starts.

In [14] we explain this naive procedure with more details and show that it returns all MinAs wrt. $C_i \sqsubseteq_\mathcal{T} D_i$.

One important improvement to the naive procedure above is that of exploiting *early pruning* and *theory propagation*, two well-known techniques from SMT (see, e.g., [13]). The $\mathcal{T}$-solver is invoked also on partial assignments μ_k on $\mathcal{P}_\mathcal{T}$: if this causes the unit-propagation of one (or more) $\neg s_{[ax_j]}$ s.t. $s_{[ax_j]} \in \mathcal{P}_\mathcal{T}$ and $s_{[ax_j]}$ is unassigned, then the antecedent clause of $\neg s_{[ax_j]}$ can be fed to analyze_conflict in DPLL2, which returns the clause $s_{[C_i \sqsubseteq_\mathcal{T} D_i]} \vee \neg\mu'_k$ s.t. $\mu'_k \subseteq \mu_k$ and $\neg s_{[C_i \sqsubseteq_\mathcal{T} D_i]} \wedge \mu'_k$ causes the propagation of $\neg s_{[ax_j]}$. (As before, we assume that analyze_conflict uses the Decision Scheme.) Intuitively, this is equivalent to say that, if $\neg s_{[C_i \sqsubseteq_\mathcal{T} D_i]} \wedge \mu_k \wedge s_{[ax_j]}$ is passed to the $\mathcal{T}$-solver, then it would return conflict and the $\mathcal{T}$-conflict clause $\psi_k^* \stackrel{\text{def}}{=} s_{[C_i \sqsubseteq_\mathcal{T} D_i]} \vee \neg\mu'_k \vee \neg s_{[ax_j]}$. Thus $\mu'_k \wedge s_{[ax_i]}$ represents a non-minimal set of axioms causing the inconsistency of $\phi^{all}_{\mathcal{T}(po)}$, which can be further minimized by the algorithm of Figure 2, as described above.

One problem of the naive procedure above, regardless of early pruning and theory propagation, is that adding to φ a "fake" blocking clause (namely $\neg\eta_k$) each time a new satisfying truth assignment η_k is found may cause an exponential blowup of φ. As shown in [9], this problem can be overcome by exploiting conflict analysis techniques. Each time a model η_k is found, it is possible to consider $\neg\eta_k$ as a conflicting clause to feed to analyze_conflict and to perform conflict-driven backjumping as if the blocking clause $\neg\eta_k$ belonged to the clause set; importantly, *it is not necessary to add permanently the conflicting clause* $\neg\eta_k$ *to* φ *as a blocking clause*, and it is sufficient to keep the conflict clause resulting from conflict analysis only as long as it is active.[3]

[3] We say that a clause is currently *active* if it occurs in the implication graph, that is, if it is the antecendent clause of some literal in the current assignment. (See [17]).

In [9] it is proved that this technique terminates and allows for enumerating all models. (Notice that the generation of blocking clauses ψ_k^* representing MinAs is not affected, since in this case we add ψ_k^* to φ as blocking clause.) The only potential drawback of this technique is that some models may be found more than once. However, according to the empirical evaluation in [9], this events appears to be rare and it has very low impact on performances, which are much better than those of the naive version.

We refer the reader to [9] and [14] for more detailed explanations of all-SMT and of our procedure respectively.

Computing One MinA Using a Much Smaller Formula. Although polynomial, $\phi^{all}_{\mathcal{T}}/\phi^{all}_{\mathcal{T}(po)}$ may be huge for very-big ontologies $\mathcal{T}$ like SNOMED-CT. For these situations, we propose here a variant of the one-MinA procedure using the much smaller formula $\phi^{one}_{\mathcal{T}}$(which is an improved SAT-based version of the simplified one-MinA algorithm of [5]).[4] $\phi^{one}_{\mathcal{T}}$ is computed like $\phi^{all}_{\mathcal{T}}$, except that step 4. is never performed, so that only one deduction of each assertion is computed. This is sufficient, however, to compute *one* non-minimal axiom set $\mathcal{T}^*$ by one run of `bcp` and `analyze_conflict`, as seen before. Since $\phi^{one}_{\mathcal{T}}$ does not represent all deductions of $C_i \sqsubseteq_{\mathcal{T}} D_i$, we cannot use the algorithm in Figure 2 to minimize it. However, since typically $\mathcal{T}^* \ll \mathcal{T}$, one can cheaply compute $\phi^{one}_{\mathcal{T}^*}$ and run a variant of the algorithm in Figure 2 in which at each loop a novel formula $\phi^{one}_{\mathcal{S}\setminus\{a_i\}}$ is computed and fed to `DPLLUnderAssumptions` together with the updated $\mathcal{L}$. One further variant is to compute instead $\phi^{all}_{\mathcal{T}^*(po)}$ and feed it to the algorithm in Figure 2.

Handling Normalization. The normalized TBox $\mathcal{T} \stackrel{\text{def}}{=} \{ax_1, ..., ax_N\}$ can result from normalizing the non-normal one $\hat{\mathcal{T}} \stackrel{\text{def}}{=} \{\hat{ax}_1, ..., \hat{ax}_{\hat{N}}\}$ by means of the process hinted in §2.1. $|\mathcal{T}|$ is $O(|\hat{\mathcal{T}}|)$. Each original axiom $\hat{ax}_i$ is converted into a set of normalized axioms $\{ax_{i1}, ..., ax_{ik_i}\}$, and each axiom ax_{ik_i} can be reused in the conversion of several original axioms $\hat{ax}_{j1}, ..., \hat{ax}_{jk_j}$. In order to handle non-normal TBoxes $\hat{\mathcal{T}}$, we adopt one variant of the technique in [5]: for every $\hat{ax}_i$, we add to $\phi^{all}_{\mathcal{T}(po)}$ [resp. $\phi^{all}_{\mathcal{T}}$] the set of clauses $\{s_{[\hat{ax}_i]} \rightarrow s_{[ax_{i1}]}, ..., s_{[\hat{ax}_i]} \rightarrow s_{[ax_{ik_i}]}\}$, and then we use $\mathcal{P}_{\hat{\mathcal{T}}} \stackrel{\text{def}}{=} \{s_{[\hat{ax}_1]}, ..., s_{[\hat{ax}_{\hat{N}}]}\}$ as the novel set of axiom selector variables for the one-MinA and all-MinAs algorithms described above. Thus `analyze_conflict` finds conflict clauses in terms of variables in $\mathcal{P}_{\hat{\mathcal{T}}}$ rather than in $\mathcal{P}_{\mathcal{T}}$. Since $\mathcal{P}_{\hat{\mathcal{T}}}$ is typically smaller than $\mathcal{P}_{\mathcal{T}}$, this may cause a significant reduction in search for `DPLL1` in the all-MinAs procedure. (Hereafter we will call $\mathcal{T}$ the input TBox, no matter whether normal or not.)

4 Discussion

By comparing the pinpointing formula $\Phi^{C_i \sqsubseteq_{\mathcal{T}} D_i}$ of [5] (see also §2.1) with $\phi^{all}_{\mathcal{T}(po)}$, and by analyzing the way they are built and used, we highlight the

[4] We prefer considering $\phi^{one}_{\mathcal{T}}$ rather the corresponding formula $\phi^{one}_{\mathcal{T}(po)}$ since it fits better with an optimization described in [14].

following differences: (i) $\Phi^{C_i \sqsubseteq_\mathcal{T} D_i}$ is built only on axiom selector variables in $\mathcal{P}_\mathcal{T} \stackrel{\text{def}}{=} \{s_{[ax_j]} \mid ax_j \in \mathcal{T}\}$, whilst $\phi^{all}_{\mathcal{T}(po)}$ is build on *all* selector variables in $\mathcal{P}_\mathcal{A} \stackrel{\text{def}}{=} \{s_{[a_j]} \mid a_j \in \mathcal{A}\}$ (i.e., of both axioms and inferred assertions); (ii) the size of $\Phi^{C_i \sqsubseteq_\mathcal{T} D_i}$ and the time to compute it are worst-case *exponential* in $|\mathcal{T}|$ [5], whilst the size of $\phi^{all}_{\mathcal{T}(po)}$ and the time to compute it are worst-case *polynomial* in $|\mathcal{T}|$; (iii) the algorithm for generating $\Phi^{C_i \sqsubseteq_\mathcal{T} D_i}$ in [5] requires intermediate logical checks, whilst the algorithm for building $\phi^{all}_{\mathcal{T}(po)}$ does not; (iv) each MinA is a *model* of $\Phi^{C_i \sqsubseteq_\mathcal{T} D_i}$, whilst it is (the projection to $\mathcal{P}_\mathcal{T}$ of) a *counter-model* of $\phi^{all}_{\mathcal{T}(po)}$. Moreover, our process can reason directly in terms of (the selector variables of) the input axioms, no matter whether normal or not.

In accordance with Theorem 5 in [5], also our approach is not output-polynomial, because in our proposed all-MinAs procedure even the enumeration of a polynomial amount of MinAs may require exploring an exponential amount of possible truth assignments. In our proposed approach, however, the potential exponentiality is completely relegated to the final step of our approach, i.e. to our variant of the all-SMT search, since the construction of the SAT formula is polynomial. Thus we can build $\phi^{all}_{\mathcal{T}(po)}$ once and then, for each $C_i \sqsubseteq_\mathcal{T} D_i$ of interest, run the all-SMT procedure until either it terminates or a given timeout is reached: in the latter case, we can collect the MinAs generated so far. (Notice that the fact that `DPLL1` selects *positive* axiom selector variables first tends to anticipate the enumeration of over-constrained assignments wrt. to that of under-constrained ones, so that it is more likely that counter-models, and thus MinAs, are enumerated during the first part of the search.) With the all-MinAs algorithm of [5], it may take an exponential amount of time to build the pinpointing formula $\Phi^{C_i \sqsubseteq_\mathcal{T} D_i}$ before starting the enumeration of the MinAs.

As far as the generation of each single MinA of §3.2 is concerned, another interesting feature of our approach relates to the minimization algorithm of Figure 2: we notice that, once $\phi^{all}_{\mathcal{T}(po)}$ is generated, in order to evaluate different subsets $\mathcal{S} \setminus \{a_j\}$ of the axiom sets, it suffices to assume different selector variables, without modifying the formula, and perform one run of `bcp`. Similarly, if we want to compute one or all MinAs for different deduced assertion, e.g. $C_1 \sqsubseteq_\mathcal{T} D_1, \ldots, C_j \sqsubseteq_\mathcal{T} D_j, \ldots$, we do not need recomputing $\phi^{all}_{\mathcal{T}(po)}$ each time, we just need assuming (i.e. querying) each time a different axiom selector variable, e.g. respectively: $\neg s_{[C_1 \sqsubseteq_\mathcal{T} D_1]}, \ldots, \neg s_{[C_j \sqsubseteq_\mathcal{T} D_j]}, \ldots$.

5 Empirical Evaluation

In order to test the feasibility of our approach, we have implemented an early-prototype version of the procedures of §3 (hereafter referred as $\mathcal{EL}^+$SAT) which does not yet include all optimizations described here and in [14], and we performed a preliminary empirical evaluation of $\mathcal{EL}^+$SAT on the ontologies of §1.[5]

[5] The first four ontologies are available at `http://lat.inf.tu-dresden.de/~meng/toyont.html`; SNOMED-CT'09 is courtesy of IHTSDO `http://www.ihtsdo.org/`

We have implemented $\mathcal{EL}^+$SAT in C++, modifying the code of the SAT solver MiniSat2.0 070721 [8]. All tests have been run on a biprocessor dual-core machine `Intel Xeon 3.00GHz` with 4GB RAM on `Linux RedHat 2.6.9-11`, except for $\phi^{one}_{\mathcal{T}(po)}$ of Snomed-CT'09 which is processed on a `Intel Xeon 2.66 GHz` machine with 16 GB RAM on `Debian Linux 2.6.18-6-amd64`.[6]

The results of the evaluation are presented in Table 3. The first block reports the data of each ontology. The second and third blocks report respectively the size of the encoded formula, in terms of variable and clause number, and the CPU

Table 3. "XeN" is "$X \cdot 10^N$". CPU times are in seconds.

Ontology	notGalen	GeneOnt.	NCI	fullGalen	Snomed09
# of prim. concepts	2748	20465	27652	23135	310075
# of orig. axioms	4379	20466	46800	36544	310025
# of norm. axioms	8740	29897	46800	81340	857459
# of role names	413	1	50	949	62
# of role axioms	442	1	0	1014	12
Size (var#|clause#)					
$\phi_{\mathcal{T}}$	5.4e3|1.8e4	2.2e4|4.2e4	3.2e4|4.7e4	4.8e4|7.3e5	5.3e5|8.4e6
$\phi^{one}_{\mathcal{T}}$	2.3e4|2.7e4	5.5e4|5.4e4	7.8e4|4.7e4	7.3e5|1.4e6	8.4e6|1.6e7
$\phi^{all}_{\mathcal{T}(po)}$	1.7e5|2.2e5	2.1e5|2.6e5	2.9e5|3.0e5	5.3e6|1.2e7	2.6e7|8.4e7
Encode time					
$\phi_{\mathcal{T}}$	0.65	2.37	2.98	35.28	3753.04
$\phi^{one}_{\mathcal{T}}$	2.06	4.15	6.19	68.94	4069.84
$\phi^{all}_{\mathcal{T}(po)}$	1.17	1.56	2.37	178.41	198476.59
Load time					
$\phi_{\mathcal{T}}$	0.11	0.37	1.01	1.93	21.16
$\phi^{one}_{\mathcal{T}}$	0.18	0.55	1.17	5.95	59.88
Subsumption (on 10^5*)*					
$\phi_{\mathcal{T}}$	0.00002	0.00002	0.00003	0.00003	0.00004
$\phi^{one}_{\mathcal{T}}$	0.00003	0.00002	0.00003	0.00004	0.00008
nMinA $\phi^{one}_{\mathcal{T}}$ *(on* 5000*)*	0.00012	0.00027	0.00042	0.00369	0.05938
MinA $\phi^{one}_{\mathcal{T}}$ *(on* 100*)*					
– *Load time*	0.175	0.387	0.694	6.443	63.324
– *Extract time*	0.066	0.082	0.214	0.303	3.280
– *DPLL Search time*	0.004	0.004	0.002	0.010	0.093
MinA $\phi^{all}_{\mathcal{T}(po)}$ *(on* 100*)*					
– *Load time*	1.061	1.385	1.370	39.551	150.697
– *DPLL Search time*	0.023	0.027	0.036	0.331	0.351
allMinA $\phi^{all}_{\mathcal{T}(po)}$*(on* 30*)*					
– 50% *#MinA/time*	1/1.50	1/1.76	4/1.79	3/53.40	15/274.70
– 90% *#MinA/time*	2/1.59	4/2.11	6/1.86	9/63.61	32/493.61
– 100% *#MinA/time*	2/1.64	8/2.79	9/2.89	15/150.95	40/588.33

[6] $\mathcal{EL}^+$SAT is available from `http://disi.unitn.it/~rseba/elsat/`

time taken to compute them.[7] The fourth block reports the time taken to load the formulas and to initialize DPLL. The fifth block reports the average time (on 100000 sample queries) required by computing subsumptions.[8] (Notice that $\phi_{\mathcal{T}}$ and $\phi_{\mathcal{T}}^{one}$ must be loaded and DPLL must be initialized only once for all queries.) The sixth block reports the same data for the computation of one nMinA, on 5000 sample queries.[9] (Loading times are the same as above.) The seventh block reports the average times on 100 samples required to compute one MinA with $\phi_{\mathcal{T}}^{one}$.[10] The eighth block reports the average times on 100 samples required to compute one MinA with $\phi_{\mathcal{T}(po)}^{all}$. The ninth block reports the results (50th, 90th and 100th percentiles) of running the all-MinAs procedure on 30 samples, each with a timeout of 1000s (loading included), and counting the number of MinAs generated and the time taken until the last MinA is generated.[11]

Although still very preliminary, these empirical results allow us to notice a few facts: (i) once the formulas are loaded, concept subsumption and computation of nMinAs are instantaneous, even with very-big formulas $\phi_{\mathcal{T}}$ and $\phi_{\mathcal{T}}^{one}$; (ii) in the computation of single MinAs, with both $\phi_{\mathcal{T}}^{one}$ and $\phi_{\mathcal{T}(po)}^{all}$, DPLL search times are very low or even negligible: most time is taken by loading the main formula (which can be performed only once for all) and by extracting the information from intermediate results. Notice that $\mathcal{EL}^+$SAT succeeded in computing some MinAs even with the huge ontology SNOMED-CT'09; (iii) although no sample concluded the full enumeration within the timeout of 1000s, the all-MinAs procedure allowed for enumerating a set of MinAs. Remarkably, all MinAs are all found in the very first part of the search, as expected.

6 Ongoing and Future Work

The current implementation of $\mathcal{EL}^+$SAT is still very naive to many extents. We plan to implement an optimized version of $\mathcal{EL}^+$SAT, including all techniques and optimizations presented here and in [14]. (We plan to investigate and implement also a SAT-based versions of the techniques based on reachability modules of [6].) Then we plan to perform a very-extensive empirical analysis of the optimized tools; we also plan to implement a user-friendly GUI for $\mathcal{EL}^+$SAT so that to make

[7] The classification alone (excluding the time taken in encoding the problem and in computing the additional *rule clauses* for pinpointing) required respectively: 0.60, 2.24, 2.84, 34.06 and 3738.82 seconds for $\phi_{\mathcal{T}}$, 0.99, 2.63, 4.13, 41.19 and 3893.20 seconds for $\phi_{\mathcal{T}}^{one}$. In the case of $\phi_{\mathcal{T}(po)}^{all}$ the times are not distinguishable.

[8] The queries have been generated randomly, extracting about 2000 primitive concept names from each ontology and then randomly selecting 100000 queries from all the possible combinations of these concept names.

[9] We chose the first 5000 "unsatisfiable" queries we encounter when analyzing all the possible pairwise combinations of primitive concept names of each ontology.

[10] The queries are selected randomly from the 5000 samples introduced above.

[11] First, we sort the assertions computed for each ontology wrt. the number of occurrences as implicate in *rule clauses* then, following this order, we pick with a probability of 0.25 (to avoid queries which are too similar) the 30 sample assertions to be queried.

it usable by domain experts. Research-wise, we plan to investigate alternative sets of completion rules, which may be more suitable for producing smaller $\phi^{all}_{\mathcal{T}(po)}$ formulas, and to extend our techniques to richer logics.

References

1. Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ Envelope. In: Proc. of IJCAI 2005, pp. 364–369. Morgan-Kaufmann Publishers, San Francisco (2005)
2. Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ Envelope Further. In: Proc. of OWLED 2008, DC Workshop (2008)
3. Baader, F., Lutz, C., Suntisrivaraporn, B.: Efficient Reasoning in $\mathcal{EL}^+$. In: Proc. of DL 2006. CEUR-WS, vol. 189 (2006)
4. Baader, F., Penaloza, R.: Axiom pinpointing in general tableaux. In: Olivetti, N. (ed.) TABLEAUX 2007. LNCS (LNAI), vol. 4548, pp. 11–27. Springer, Heidelberg (2007)
5. Baader, F., Peñaloza, R., Suntisrivaraporn, B.: Pinpointing in the Description Logic $\mathcal{EL}^+$. In: Hertzberg, J., Beetz, M., Englert, R. (eds.) KI 2007. LNCS, vol. 4667, pp. 52–67. Springer, Heidelberg (2007)
6. Baader, F., Suntisrivaraporn, B.: Debugging SNOMED CT Using Axiom Pinpointing in the Description Logic $\mathcal{EL}^+$. In: Proc. of KR-MED 2008: Representing and Sharing Knowledge Using SNOMED. CEUR-WS, vol. 410 (2008)
7. T. G. O. Consortium. Gene ontology: Tool for the unification of biology. Nature Genetics 25, 25–29 (2000)
8. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
9. Lahiri, S.K., Nieuwenhuis, R., Oliveras, A.: SMT techniques for fast predicate abstraction. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 424–437. Springer, Heidelberg (2006)
10. Lynce, I., Silva, J.P.M.: On Computing Minimum Unsatisfiable Cores. In: Proc. of SAT 2004 (2004), `http://www.satisfiability.org/SAT04/programme/110.pdf`
11. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proc. of DAC 2001, pp. 530–535. ACM, New York (2001)
12. Rector, A., Horrocks, I.: Experience Building a Large, Re-usable Medical Ontology using a Description Logic with Transitivity and Concept Inclusions. In: Proc. of the Workshop on Ontological Engineering, AAAI 1997. AAAI Press, Menlo Park (1997)
13. Sebastiani, R.: Lazy Satisfiability Modulo Theories. Journal on Satisfiability, Boolean Modeling and Computation, JSAT 3, 141–224 (2007)
14. Sebastiani, R., Vescovi, M.: Axiom Pinpointing in Lightweight Description Logics via Horn-SAT Encoding and Conflict Analysis. Technical Report DISI-09-014, University of Trento (2009), `http://disi.unitn.it/~rseba/elsat/`
15. Sioutos, N., de Coronado, S., Haber, M.W., Hartel, F.W., Shaiu, W., Wright, L.W.: NCI Thesaurus: A semantic model integrating cancer-related clinical and molecular information. Journal of Biomedical Informatics 40(1), 30–43 (2007)
16. Spackman, K.A.: Managing clinical terminology hierarchies using algorithmic calculation of subsumption: Experience with SNOMED RT. Journal of American Medical Informatics Association (Fall Symposium Special Issue) (2000)
17. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In: Proc. of ICCAD, pp. 279–285 (2001)

Does This Set of Clauses Overlap with at Least One MUS?

Éric Grégoire, Bertrand Mazure, and Cédric Piette

Université Lille-Nord de France, Artois
CRIL-CNRS UMR 8188
F-62307 Lens Cedex, France
{gregoire,mazure,piette}@cril.fr

Abstract. This paper is concerned with the problem of checking whether a given subset Γ of an unsatisfiable Boolean CNF formula Σ takes part in the basic causes of the inconsistency of Σ. More precisely, an original approach is introduced to check whether Γ overlaps with at least one minimally unsatisfiable subset (MUS) of Σ. In the positive case, it intends to compute and deliver one such MUS. The approach re-expresses the problem within an evolving coarser-grained framework where clusters of clauses of Σ are formed and examined according to their levels of mutual conflicts when they are interpreted as basic interacting entities. It then progressively refines the framework and the solution by splitting most promising clusters and pruning the useless ones until either some maximal preset computational resources are exhausted, or a final solution is discovered. The viability and the usefulness of the approach are illustrated through benchmarks experimentations.

1 Introduction

These last years, various research studies have concentrated on explaining *why* a SAT instance is inconsistent in terms of minimal subsets of clauses that are actually conflicting. Indeed, although some SAT solvers (e.g. [1,2]) can deliver the trace of a proof of inconsistency, this trace is often not guaranteed to deliver a set of conflicting clauses that would become consistent if any of its clauses was dropped. Accordingly, several recent contributions have investigated various computational issues about computing Minimal Unsatisfiable Subsets (in short, MUSes) of unsatisfiable SAT instances ([3,4], see [5] for a recent survey). From a worst-case complexity analysis, several major issues arise. First, an n-clauses SAT instance Σ can exhibit $C_n^{n/2}$ MUSes in the worst case. Then, checking whether a given formula belongs to the set of MUSes of another CNF formula or not is a $\sum_2^p$-hard problem [6]. However, approaches to compute one MUS that appear viable in many circumstances have been proposed [7,8]. In order to circumvent the possibly exponential number of MUSes in Σ, variant problems have been defined and addressed in the literature, like the problem of computing a so-called cover of MUSes, which is a subset of clauses that contains enough minimal sources of conflicts to explain all unrelated reasons leading to the inconsistency of Σ, without computing all MUSes of Σ [8]. Finally, algorithms to compute the complete sets of

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 100–115, 2009.

MUSes of Σ have also been proposed and shown viable [12,11], at least to some extent due to the possible combinatorial blow-up.

When a user is faced with an unsatisfiable SAT instance, she (he) can have some beliefs about which subset Γ of clauses is actually causing the conflicts within Σ. In this respect, she (he) might want to check whether her (his) beliefs are grounded or not. Apart from the situation where $\Sigma \setminus \Gamma$ is satisfiable, she (he) might wish to check whether Γ shares a non-empty set-theoretical intersection with at least one MUS of Σ. In the positive case, she (he) might wish to be delivered such an MUS as well. Current techniques to find and compute MUSes do not accommodate those wishes without computing all MUSes of Σ.

An original approach is introduced in the paper to address those issues, without computing all MUSes of Σ. To circumvent the high worst-case complexity (at least, to some extent), it re-expresses the problem within an evolving coarser-grained framework where clusters of clauses of Σ are formed and examined according to their levels of mutual conflicts when they are interpreted as basic interacting entities. It then progressively refines the framework and the solution by splitting most promising clusters and pruning useless ones until either some maximal preset computational resources are exhausted, or a final solution is discovered. Interestingly, the levels of mutual conflicts between clusters are measured using a form of Shapley's values [9].

The paper is organized as follows. In the next Section, formal preliminaries are provided, together with basic definitions and properties about MUSes and Shapley's measure of conflicting information. The main principles guiding the approach are described in Section 3. The sketch of the any-time algorithm is provided in Section 4, while the empirical results are given and discussed in Section 5. Main other relevant works are then described before paths for future research are provided.

2 Logical Preliminaries, MUSes and Shapley's Measure

2.1 Logical Preliminaries and SAT

Let L be the propositional language of formulas defined in the usual inductive way from a set P of propositional symbols (represented using plain letters like a, b, c, etc.), the Boolean constants $\top$ and $\bot$, and the standard connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$. A SAT instance is a propositional formula in conjunctive normal form (CNF for short), i.e. a conjunction (often represented through a set) of clauses, where a clause is a disjunction (also often represented as a set) of literals, a literal being a possibly negated propositional variable. In the following, plain letters like l, m, n, etc. will be used to represent formulas of L. Upper-case Greek letters like Δ, Γ, etc. will be used to represent sets of clauses, and lower-case ones like α, β, γ etc. to represent clauses.

SAT is the canonic NP-complete decision problem consisting in checking whether a SAT instance is satisfiable or not. In the following, the words satisfiable (resp. unsatisfiable) and consistent (resp. inconsistent) are used indifferently. Along the paper, Σ is an unsatisfiable SAT instance and Γ is a set of clauses s.t. $\Gamma \subset \Sigma$.

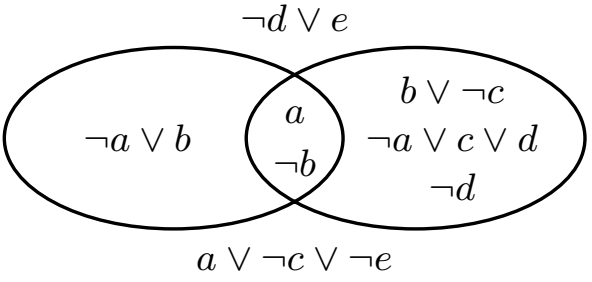


Fig. 1. MUSes of Σ from Example 1

2.2 MUSes of a SAT Instance

An MUS of an unsatisfiable SAT instance Σ is an unsatisfiable subset of clauses of Σ that cannot be made smaller without restoring its satisfiability. An MUS thus represents a minimal cause of inconsistency, expressed by means of conflicting clauses.

Definition 1. *A set of clauses Γ is a Minimally Unsatisfiable Subset (MUS) of Σ iff*

1. $\Gamma \subseteq \Sigma$
2. *Γ is unsatisfiable*
3. $\forall \Delta \subset \Gamma$, *$\Delta$ is satisfiable.*

The following example illustrates that MUSes can overlap one another.

Example 1. *Let* $\Sigma = \{\neg d \vee e,\ b \vee \neg c,\ \neg d,\ \neg a \vee b,\ a,\ a \vee \neg c \vee \neg e,\ \neg a \vee c \vee d,\ \neg b\}$. *$\Sigma$ is unsatisfiable and contains 2 MUSes which are illustrated in Figure 1, namely* $\{a,\ \neg a \vee b,\ \neg b\}$ *and* $\{b \vee \neg c,\ \neg d,\ a,\ \neg a \vee c \vee d,\ \neg b\}$.

This example also illustrates how clauses in Σ can play various roles w.r.t. the unsatisfiability of Σ. Following [10], clauses that belong to all MUSes of Σ are *necessary* (w.r.t. the unsatisfiability of Σ). Removing any necessary clause from Σ restores consistency. *Potentially necessary* clauses of Σ belong to at least one MUS of Σ but not to all of them: removing one potentially necessary clause does not restore the consistency of Σ. Clauses that do not belong to any of those two categories (i.e. clauses belonging to no MUS at all) are called *never necessary*. Removing any combination of never necessary clauses cannot restore consistency. Obviously enough, an unsatisfiable SAT instance does not always contain necessary clauses since it can exhibit non-overlapping MUSes.

Example 1 (cont'd). *In this example, both clauses "a" and "$\neg b$" are necessary w.r.t. the inconsistency of Σ, "$\neg d \vee e$" and "$a \vee \neg c \vee \neg e$" are never necessary whereas all the other clauses of Σ are potentially necessary.*

Numerous approaches have been proposed to extract one MUS from Σ [5]. However, these approaches cannot take into account *a priori* conditions stating that this MUS

should overlap with Γ. Accordingly, in order to find such an overlapping MUS using the best current available techniques, one must resort to tools like [11,12] that compute all MUSes of Σ. Note that the most powerful of these techniques compute the Maximal Satisfiable Subsets (MSSes) of Σ [11] as a first necessary step, and then derive MUSes as hitting-sets of all the MSSes. Accordingly, these techniques are hardly tractable and cannot be used for many large and difficult SAT instances, even when the actual number of MUSes is not too large.

2.3 Shapley's Measure of Inconsistency

There exist various studies about the possible levels of inconsistency within a set of formulas. In the following, one form of Shapley's measure [9] of inconsistency proposed in [13] will be used. It can be formulated as follows.

Shapley's inconsistency measure of a clause α in a SAT instance Σ delivers a score that takes the number of MUSes of Σ containing α into account, as well as the size of each of these MUSes, enabling the involvement or "importance" of the clauses within each source of inconsistency to be considered, too.

Definition 2. *[13] Let Σ be a SAT instance and $\alpha \in \Sigma$. Let $\cup_{MUS_\Sigma}$ be the set of all MUSes of Σ. The measure of inconsistency MI of α in Σ, noted $MI(\Sigma, \alpha)$, is defined as*

$$MI(\Sigma, \alpha) = \sum_{\{\Delta s.t. \Delta \in \cup_{MUS_\Sigma} \text{ and } \alpha \in \Delta\}} \frac{1}{|\Delta|}$$

Properties of such a measure are investigated in [13]. Roughly, a clause that takes part in many conflicts is assigned a higher score, while at the same time a clause that occurs in a large MUS is given a lower score than a clause that occurs in an MUS containing a smaller number of clauses.

Example 2. *Considering the CNF Σ of Example 1, we have:*

- $MI(\Sigma, \neg a \vee b) = 1/3$
- $MI(\Sigma, b \vee \neg c) = 1/5$
- $MI(\Sigma, \neg d \vee e) = 0$
- $MI(\Sigma, a) = MI(\Sigma, \neg b) = 1/3 + 1/5 = 8/15$

In the following, this measure will be extended to characterize the conflict levels between conjunctive formulas.

3 Main Principles Underlying the Approach

3.1 Problem Definition

A set of Boolean clauses Γ *actually* participates in the inconsistency of a SAT instance Σ when Γ contains at least one clause that belongs to at least one MUS of Σ, namely a clause that is necessary or potentially necessary w.r.t. the inconsistency of Σ. In the positive case, Γ is said to *participate in the inconsistency* of Σ and the goal is to deliver one such MUS. More formally:

Definition 3. *Let $\cup_{MUS_\Sigma}$ be the set of MUSes of a set of Boolean clauses Σ and let Γ be a set of Boolean clauses, too.* Γ participates in the inconsistency *of Σ iff $\exists \alpha \in \Gamma, \exists \Delta \in \cup_{MUS_\Sigma}$ s.t. $\alpha \in \Delta$.*

Clearly enough, if Γ contains at least one necessary clause w.r.t. the inconsistency of Σ then Γ participates in the inconsistency of Σ. In this specific case, checking participation in inconsistency requires at most k calls to a SAT solver, where k is the number of clauses of Γ, since there exists α in Γ s.t. $\Sigma \backslash \{\alpha\}$ is satisfiable. Moreover, any MUS of Σ overlaps with Γ and provides a solution to the problem. However, in the general case, whether at least one necessary clause w.r.t. the satisfiability of Σ exists or not within Γ is unknown.

3.2 Using Clusters to Move to a Simplified Level

Accordingly, and in order to circumvent to some extent the high-level computational complexity of the problem, the approach in this paper resorts to a strategy that consists in partitioning Σ into a prefixed m number of subsets of clauses, called *clusters*, as a first step.

Definition 4. *Let Σ be a SAT instance. An* m-clustering of clauses *Π of Σ is a partition of Σ into m subsets Π_j where $j \in [1..m]$. Π_j is called the j^{th} cluster of Σ (w.r.t. the clustering Π of Σ).*

For convenience reasons, the same notation Π_j will be used to represent both the set of clauses forming the j^{th} cluster of Σ and the conjunctive formula made of those clauses. Also, a set-theoretical union of clusters will be identified as the conjunction of the clauses that they contain.

All the clauses inside a cluster are then considered conjunctively to deliver a formula that is interpreted as being an indivisible interacting entity within Σ. The clauses of Γ are treated in the same way to form an additional conjunctive formula. At this point the initial problem is moved to a coarser-grained one that consists in checking how the conjunctive formula of Γ treated as a whole takes part in the inconsistency of the new representation of Σ that is now made of m individual subformulas that are considered as basic interacting entities.

3.3 From MUSes to MUSCes

As the problem is moved to a framework where inconsistency is analysed using clusters as basic interacting entities, the MUS concept should be adapted accordingly, giving rise to a concept of Minimally Unsatisfiable Set of Clusters for a clustering Π ($MUSC_\Pi$ for short) of Σ.

Definition 5. *Let Σ be a SAT instance and Π an m-clustering of clauses of Σ. A* Minimally Unsatisfiable Set of Clusters *for Π ($MUSC_\Pi$ for short) M_Π of Σ is a set of clusters s.t.:*

1. *$M_\Pi \subseteq \Pi$*
2. *M_Π is unsatisfiable*
3. *$\forall \Phi \in M_\Pi$, $M_\Pi \setminus \{\Phi\}$ is satisfiable*

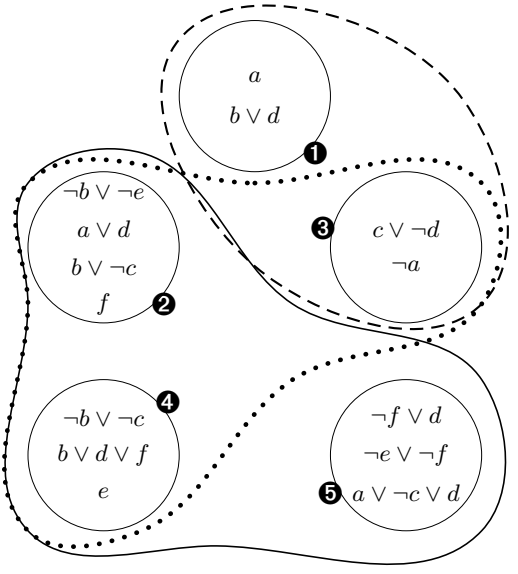


Fig. 2. MUSes of Σ from Example 3

The set of $MUSC_\Pi$ of Σ is noted $\cup_{MUSC_\Pi}$.

The standard MUS concept is a particular instantiation of the MUSC one, where each cluster is actually one clause. Accordingly, MUSes represent the finest-grained level of abstraction in order to explain inconsistency of Σ by means of contradicting clauses. The MUSC concept allows more coarser-grained clusterings to be used, as ensured by the following property.

Let us consider the clauses that we want to prove they are involved in the inconsistency of Σ. When those clauses appear in an $MUSC_\Pi$ of Σ (for some clustering Π), they also participate in the inconsistency of Σ when this issue is represented at the lowest level of abstraction, namely at the level of clauses. More formally:

Proposition 1. *Let Σ be a SAT instance, Π an m-clustering of Σ and M_Π an $MUSC_\Pi$ of Σ. If $\Pi_i \in M_\Pi$ then $\exists \alpha \in \Pi_i, \exists \Psi \in \cup_{MUS_\Sigma}$ s.t. $\alpha \in \Psi$.*

Such a problem transformation exhibits interesting features from a computational point of view that will be exploited by the approach. But there is no free lunch. As the new problem is a simplification of the initial one, some MUSes might disappear in the process. However, this simplification remains compatible with the initial goal and an algorithm will be proposed that is sound and complete in the sense that it always delivers an MUS of Σ overlapping with Γ, whenever this one exists and enough computational resources are provided.

Example 3. *Let Σ consists of 14 clauses, as depicted in Figure 2, forming 9 MUSes. For example, when the participation of the clauses "$\neg a$" and "$c \vee \neg d$" in the inconsistency of Σ needs to be checked and, in the positive case, when a corresponding MUS*

needs to be extracted, computing the exhaustive set of MUSes can appear necessary in the worst case. Let us cluster the clauses in the way depicted in Figure 2, giving rise to 5 clusters; the cluster ❸ is the set of clauses Γ accordingly to our previous notation. At this level of abstraction, only 3 $MUSC_\Pi$ remain, i.e. $M_1 = \{❶,❸\}$, $M_2 = \{❷,❸,❹\}$ and $M_3 = \{❷,❹,❺\}$, using the labels of clusters represented in the Figure. This clustering limits the number of entities that are sources of inconsistency and that need to be considered in order to prove the membership of one of the two clauses "$\neg a$" and "$c \vee \neg d$" to an MUS of Σ.

This example illustrates the interest in clustering clauses in that it can avoid the need of extracting all MUSes in order to answer the question of the existence of an overlapping MUS. However, if the clauses are clustered and do not appear in an $MUSC$ (for some clustering Π), this does not entail that they do not participate in the inconsistency of the formula at the clauses level.

Proposition 2. *Let Σ be a SAT instance, Π an m-clustering of Σ and $\Pi_i \in \Pi$. Proposition 1 ensures that:*

$$\textit{if } (\exists M_\Pi \in \cup_{MUSC_\Pi} \textit{ s.t. } \Pi_i \in M_\Pi) \textit{ then } (\exists \alpha \in \Pi_i, \exists \Psi \in \cup_{MUS_\Sigma} \textit{ s.t. } \alpha \in \Psi)$$

The converse does not hold.

Accordingly, the fact that a cluster does not appear in any MUSC does not imply that no one of its clauses is not involved in an MUS of the considered instance. This is illustrated by the following example:

Example 4. *Let $\Sigma = \{\neg c, \neg a, b \vee c, a, \neg b\}$. Σ exhibits 2 MUSes: $MUS_\Sigma^1 = \{\neg a, a\}$ and $MUS_\Sigma^2 = \{\neg b, b \vee c, \neg c\}$. Let us now consider a clustering Π of Σ s.t. $\Pi_1 = \{\neg b, a\}$, $\Pi_2 = \{\neg a, b \vee c\}$ and $\Pi_3 = \{\neg c\}$. Considering a table where each column represents a cluster and each line represents an MUS of Σ, we get:*

	Π_1	Π_2	Π_3
MUS_Σ^1	a	$\neg a$	
MUS_Σ^2	$\neg b$	$b \vee c$	$\neg c$

At this level of abstraction, the only $MUSC_\Pi$ is $\{\Pi_1, \Pi_2\}$, capturing MUS_Σ^1. However, MUS_Σ^2 is not represented in this unique $MUSC_\Pi$, since its clauses are splitted in each cluster of Π. Particularly, this clustering does not permit the involvement of "$\neg c$" in the inconsistency of Σ to be shown.

Accordingly, a form of "subsumption" phenomenon between causes of inconsistency can occur when clauses are clustered in such a way. On the one hand, the number of detectable MUSes in Σ can decrease accordingly, and a subsequent process could be allowed to focus on selected remaining ones only. On the other hand, it should be ensured that too much information relevant to the inconsistency of Σ is not hidden in the process, making the goal of finding an overlapping MUS impossible to reach, although such an MUS would exist. To some extent, the last example also illustrates that the

way according to which the clusters are formed and thus how MUSes are disseminated amongst clusters can drastically influence the efficiency of the approach.

3.4 Measuring Conflicts amongst MUSCses

The measure of inconsistency is based on a form of Shapley's values described in Section 2.3, which needs to be adapted as follows to consider clusters as basic interacting entities.

Definition 6. *Let Σ be a SAT instance and Π an m-clustering of Σ. The measure of inconsistency MI of a cluster Π_i of Π is defined as follows:*

$$MI(\Pi_i) = \sum_{\{M | M \in \cup_{MUSC_\Pi} \text{ and } \Pi_i \in M\}} \frac{1}{|M|}$$

Once the set of MUSCes has been extracted, this measure can be computed in polynomial time.

Example 5. *Let Σ be the SAT instance and Π be a clustering corresponding to Example 3. We have:*

- $MI(\Pi_1) = 1/2$
- $MI(\Pi_2) = MI(\Pi_4) = 2 \times 1/3 = 2/3$
- $MI(\Pi_3) = 1/2 + 1/3 = 5/6$
- $MI(\Pi_5) = 1/3$

The cluster that exhibits the highest score is Π_3. This is partly due to the fact that Π_3 belongs to 2 MUSCes. Π_2 and Π_4 also appear inside two MUSCes, but each of them only represents a third of both sources of conflict, whereas Π_3 is involved in an MUSC made of 2 clusters. The role of this latter cluster is thus very important in this source of conflict.

3.5 Initiating the Process

In order to form the initial clustering of clauses, a cheap scoring heuristic is used to estimate for each clause α of Σ its probability of belonging to at least one MUS of Σ. This heuristic exploits a failed local-search for satisfiability of Σ and a so-called concept of *critical clause* that takes a relevant partial neighborhood of each explored interpretation into account [8].

Σ is divided into a preset number m of clusters in the following way. The $\frac{|\Sigma \setminus \Gamma|}{m}$ clauses of $\Sigma \setminus \Gamma$ that exhibit the highest scores are (probably) taking part in a number of conflicts and their presence in Σ could be the cause of many MUSes. They are assembled to form the Π_1 cluster. The remaining clauses of $\Sigma \setminus \Gamma$ are then sorted in the same manner according to their scores to deliver the remaining $m-1$ clusters of Π and to provide a m-clustering of $\Sigma \setminus \Gamma$. The clauses of Γ are then conjuncted together to yield a $m+1^{th}$ cluster, noted Π_Γ. Accordingly, a $m+1$-clustering of Σ is obtained.

3.6 Analysing the Conflicts at the Clusters Level

Each cluster is thus interpreted as a conjunctive formula and the interaction between these entities is analysed in terms of mutually contradicting (sets of) clusters. This conflict analysis is intended to prune the problem by rejecting clusters that are not conflicting with Γ, and in concentrating on the clusters that are most conflicting with Γ. The exhaustive set of MUSCes of this clustering (i.e. $\cup_{MUSC_\Pi}$) is thus computed and three different cases might occur:

1. *Global inconsistency:* a form of global inconsistency occurs at the level of clusters when for every cluster Π_i ($i \in [1..m]$) of the m-clustering Π, $\Pi \setminus \Pi_i$ is consistent. Such a situation occurs when at least one clause of each cluster is needed to form a conflict. The clustering does not provide any other useful information. The parameter m is increased to obtain a finer-grained clustering. Note that this situation does not require a lot of computational ressources to be spent, since the current clustering contains only one MUSC and a linear number of maximal satisfiable sets of cluster need to be checked.

2. *Γ appears in at least one conflict:* some minimal conflicts can be detected between clusters, and at least one such conflict involves the Π_Γ cluster formed with Γ. More precisely, $\exists M \in \cup_{MUSC_\Pi}$ s.t. $\Pi_\Gamma \in M$. The idea is to focus on such an MUSC since it involves the proof that some clauses in Γ minimally conflict with Σ. More precisely, there exists an MUS of Σ that necessarily contains clauses of Γ and clauses occurring in the other clusters of the MUSC. Accordingly, the current set of clauses can be pruned by dropping all clauses that are occurring in clusters not present in this particular MUSC.

3. *Γ does not appear in any conflict:* some minimal conflicts can be detected between clusters but no such conflict involves the cluster formed with Γ. More precisely, $\nexists M \in \cup_{MUSC_\Pi}$ s.t. $\Pi_\Gamma \in M$. In such a situation, a refinement of the clustering is undertaken by splitting its *most conflicting* current cluster, using a measure that is described in paragraph 3.4. A prefixed number of clusters with the highest scores are splitted, motivated by the heuristic that they should be the most probable ones that could include and hide a number of other sources of inconsistency, and hopefully MUSes in which Γ is involved.

3.7 Iterating and Ending the Process

The clustering is thus modified following one of the three possible situations described above, allowing the approach to increase the number of clusters, split and focus on most promising subparts of Σ, mainly. This process is iterated. Accordingly, a $|\Sigma|$-clustering (i.e. a clustering where each cluster is actually a single clause) can be obtained in a finite time, corresponding to the "classical" computation of exhaustive approaches, on a reduced number of clauses of Σ. Actually, when the current size of the cluster that is most conflicting with Γ becomes manageable to envision the computation of all the MUSes of the set-theoretical union of Γ with Σ, such a computation is performed.

Algorithm 1. The `look4MUS` algorithm.

Input:
Σ: a CNF
Γ : a CNF s.t. $\Gamma \subset \Sigma$
m: size of the initial clustering
inc: size of increment of the clustering (global inconsistency case)
s: number of clusters to split (local inconsistency case)
Output: An MUS of Σ overlapping Γ when such a MUS exists, $\emptyset$ otherwise

```
1  begin
2      S_Σ ⟵ scoreClauses(Σ) ;
3      Π ⟵ clusterClauses(Σ \ Γ, S_Σ, m) ∪{Γ} ;
4      ∪MUSC_Π ⟵ HYCAM*(Π) ;
5      if Π is a |Σ|-clustering then
6          if ∃M ∈ ∪MUSC_Π s.t. Γ ⊆ M then return M ;
7          else return ∅ ;
8      else
9          if ∀Π_i ∈ Π, Σ \ Π_i is satisfiable then
               // global inconsistency
10             return look4MUS(Σ, Γ, m + inc, inc, s) ;
11         else
12             while ∄M ∈ ∪MUSC_Π s.t. Γ ∈ M do
                   // local inconsistency
13                 Π' ⟵ ∅ ;
14                 for j ∈ [1..s] do
15                     Π_best ⟵ ∅ ;
16                     foreach Π_i ∈ Π s.t. Π ≠ Γ do
17                         if MI(Π_i,∪MUSC_Π)>MI(Π_best,∪MUSC_Π) then
                               Π_best ⟵ Π_i ;
18                     Π ⟵ Π \ Π_best ;
19                     Π' ⟵ Π' ∪ clusterClauses(Π_best, S_Σ, 2) ;
20                 Π ⟵ Π ∪ Π' ;
21                 ∪MUSC_Π ⟵ HYCAM*(Π) ;
22             M_Γ ⟵ selectOneMUSC(∪MUSC_Π,Γ) ;
23             return look4MUS(M_Γ, Γ, m, inc, s) ;
24 end
```

One interesting feature of this approach thus lies in its any-time property: whenever a preset amount of computing resources is exhausted, it can provide the user with the current solution, namely the currently focused subpart of Σ that is conflicting with Γ. The proposed approach based on those ideas is described in the next Section and sketched in Algorithm 1.

Function `scoreClauses`

Input: Σ: a CNF
Output: A score vector of clauses of Σ

```
1  begin
2      for each clause c of Σ do
3          S_Σ(c) ⟵ 0;
4      I ⟵ a random assignment of each variable of Σ ;
5      while a preset maximum of steps is reached do
6          for each clause c of Σ do
7              if c is critical w.r.t I in Σ then
8                  S_Σ(c) + + ;
9          I ⟵ I' s.t. I and I' differs exactly by one assignment of a Boolean variable ;
10     return S_Σ ;
11 end
```

Function `clusterClauses`

Input: Σ: a CNF, S_Σ: a score vector, m: the size of the clustering
Output: A m-clustering of Σ

```
1  begin
2      Π ⟵ ∅ ;
3      for i ∈ [1..m − 1] do
4          Π_i ⟵ the (|Σ|/m)^th clauses with the highest scores in Σ ;
5          Π ⟵ Π ∪ {Π_i} ;
6          Σ ⟵ Σ \ Π_i ;
7      Π ⟵ Π ∪ {Σ} ;
8      return Π ;
9  end
```

4 Main Algorithm

In this section, the reader is provided with a sketch of the Algorithm. For convenience reasons, specific cases that are handled in the actual implementation (e.g. situations where Γ is inconsistent or where Σ is consistent) are not described. Also, it does not include the presentation of a syntactic preprocessing that removes all occurrences of identical clauses of Σ but one (left in Γ whenever possible), allowing the number of MUSes to be reduced exponentially. Such situations are important to handle in practice in order to avoid pathological cases: they are simple to implement and do not offer original algorithmic interest. The algorithm also refers to the `HYCAM` technique, that allows the exhaustive set of MUSes of a SAT instance to be computed. Actually, a new version of `HYCAM` that is able to deal with clusters of clauses has been implemented and used in the experimental study. We do detail the internal algorithmic techniques of `HYCAM*` which is just a cluster-oriented version of the algorithm presented in [12]. Let

Function `selectOneMUSC`

Input: $\cup_{MUSC_\Pi}$: the set of MUSC w.r.t. Π, Γ: a CNF
Output: An MUSC of $\cup_{MUSC_\Pi}$ containing Γ

```
1 begin
2     M_Γ ⟵ Π ;
3     foreach M s.t. M ∈ ∪_MUSC_Π do
4         if Γ ⊆ M and |M| < |M_Γ| then
5             M_Γ ⟵ M ;
6     return M_Γ ;
7 end
```

Function `MI`

Input: Π_i: an element of the clustering Π, $\cup_{MUSC_\Pi}$: the set of $MUSC_\Pi$
Output: The measure of inconsistency of Π_i w.r.t. Π

```
1 begin
2     mi ⟵ 0 ;
3     foreach MUSC_Π ∈ ∪_MUSC_Π do
4         if Π_i ∈ MUSC_Π then
5             mi ⟵ mi + 1/|MUSC_Π| ;
6     return mi ;
7 end
```

us focus on the different steps of the approach depicted in Algorithm 1. The approach takes as input a CNF Σ together with a subpart Γ of Σ. Moreover, it also needs 3 positive integer parameters, namely m, inc and s.

First, a local search is run for heuristically scoring the clauses w.r.t. their "constraintness" within Σ. More precisely, scores of all clauses are initially set to 0. During the local search process, scores of clauses which are *critical* [8] w.r.t. the current explored interpretation are increased. Roughly, a clause is critical if it is falsified by the current interpretation and if any neighbor interpretation that satisfies this clause falsifies another clause previously satisfied. Hence, a prefixed number of interpretations is explored by the local search, giving rise to a score for each clause of the CNF (see function `scoreClause` for more details).

Then, clauses are clustered (line 1) into Π, invoking the `clusterClauses` function. This function consists in conjuncting clauses that exhibit the highest scores to form a first cluster, then the remaining highest-scored clauses to form a second cluster, etc. until m clusters are delivered. Next, the modified version of `HYCAM`, noted `HYCAM*` (line 1), is called to compute all MUSCs of Π.

Now, $\cup_{MUSC_\Pi}$ has been computed and several cases are possible. Either Π is globally inconsistent (lines 1-1): at this stage, no information can be exploited to find a subset of conflicting clauses with Γ (see Section 3.6.1) and the clustering is refined by adding inc more clusters (in the sketch of the algorithm, this is done through a recursive call to `look4MUS`). Another case occurs when some local inconsistency within the

clustering is detected. Here, either Γ is involved in at least one local inconsistency exhibited by the procedure (see Section 3.6.2), or no one of them contains Γ (see Section 3.6.3). Actually, while Γ is not involved in any local MUSC of Π (lines 1-1), the s most conflicting clusters of Π (w.r.t. the Shapley value in `MI` function) are split to attempt to reveal sources of conflict involving Γ. When such local inconsistencies are exhibited, one MUSC M_Γ containing Γ is selected (function `selectOneMUSC`, which chooses the MUSC that contains the least possible number of clauses). The computation continues only considering the M_Γ CNF, which is a subformula of Σ that contains one MUS involving Γ.

Unless the preset amount of computing time is exhausted, the algorithm can only terminate in lines 1 or 1, when each cluster is actually a single clause ($|\Sigma|$-clustering). At this point, a "classical" call to HYCAM is performed, delivering either MUSes (the "MUSC" line 1 is actually an MUS since each cluster is a clause) containing information from Γ, or proving that no such MUS exists. In the actual implementation, a $|\Sigma|$-clustering does not need to be reached in order to call the `HYCAM` procedure: when the current number of clusters is close enough to the number of clauses (i.e. when $m < 2 \times |\Sigma|$), then the procedure is ended up with an `HYCAM` call.

Moreover, let us note that once line 22 is reached, an approximation of the original problem has been computed. Consequently, the algorithm could be terminated at any time with an approximated solution. The actual implementation is provided with an anytime feature in the sense that the approach delivers the current considered subformula conflicting with Γ when the preset amount of computing resources is exhausted.

5 Experimental Results

A `C` implementation of the algorithm has been completed. As a case study, the following values have been selected as parameters: $m = 30$, $inc = 10$ and $s = \frac{m}{10}$. The algorithm has been run on various SAT benchmarks from `http://www.satcompetition.org`. All the experimentations have been conducted on Intel Xeon 3GHz under Linux CentOS 4.1. (kernel 2.6.9) with a RAM limit of 2GB.

As a case study again, for each SAT instance it has been attempted to show that the subformula made of the clauses that are occurring at the 1^{st}, 3^{rd}, 5^{th} and 7^{th} positions within the standard DIMACS CNF-format file was participating in the inconsistency of the instance, and find an MUS overlapping with this set. A sample of the results are reported in Table 1. For each result, we report the name of the tested benchmark together with its number of variables (#var) and clauses (#cla). In addition, the size of the specific wanted MUS (#cla$_{MUS}$) in term of number of clauses and the time (in second) needed to extract it are also proposed.

This table shows that specific large (and thus difficult to extract) MUSes can be efficiently extracted from SAT instances. Note that the exhaustive set of MSSes of most of these instances cannot be delivered in reasonable time with neither `CAMUS` [11] nor `HYCAM` [12]. Thus, preexisting approaches cannot answer the question about the involvement of Γ in the inconsistency of Σ.

On the contrary, clustering clauses in order to hide sources of inconsistency and focus on the ones involving the wanted clauses proved valuable in practice. While no

Table 1. Extracting one MUS of Σ overlapping with Γ: sample of experimental results

name	#var	#cla	#cla$_{MUS}$	time
1dlx_c_mc_ex_bp_f	776	3725	1440	705
bf0432-007	1040	3668	1223	119
bf1355-075	2180	6778	151	22
bf1355-638	2177	6768	154	21
bf2670-001	1393	3434	134	8
dp05u04	1571	3902	820	1254
dp06u05	2359	6053	1105	2508
ezfact16_1	193	1113	319	28
ezfact16_2	193	1113	365	43
ezfact16_3	193	1113	297	31
ezfact16_10	193	1113	415	54

approach was able to answer this query for various realistic problems, the approach in this paper drastically reduced the computational effort by limiting the combinatorial explosion of the number of MUSes. For instance, the `dp05u04` and `dp06u05` problems (encoding the dining philosophers problem *à la* bounded model checking) contain a number of MUSes large enough to prevent them from being computed (or even enumerated) using the current available technology. `look4MUS` succeeds to compute one of the wished minimal sources of inconsistency -without extracting all of them- in 20 and 42 minutes, respectively. The situation is similar for other considered CNFs. For the `ezfact16_*` factorization circuits family, an explanation involving parlicular clauses can be delivered in less than 1 minute while exhaustive approaches show their limits with such problems. Obviously enough, the considered benchmarks are smaller than the very large CNFs containing tens of thousand clauses that can now be solved by the best SAT solvers. Nevertheless, for some of those instances, thanks to its any-time feature, the procedure was able to iterate several times and compute MUSCes, delivering a subformula conflicting with Γ.

6 Other Relevant Works

Existing approaches to compute MUSes and address the specific problem investigated in this paper have been described in previous sections. Let us simply emphasize here that this piece of work involves a form of *abstraction* to reduce the computational cost of the initial problem. Moreover, due to its any-time property, it can be used as an *approximation* technique. In this respect, a large number of abstraction and approximation techniques have been proposed to address various problems related to propositional logic but that are different from the issue addressed in this paper. Let us mention some of them.

A seminal paper about the approximation of inference in propositional logic using limited ressources is [14]. Also, approximate clausal entailment relationships where any step of the computation is decided in polytime have been proposed in [15,16], and extended to full classical logic in [17]. For an overview about propositional

approximations, the reader is refered to [18]. Finally, let us note that approximation of coherent-based reasoning has also been studied extensively in e.g. [19] and [20].

Finally, the idea of clustering clauses has already been proposed in the context of high-level constraints encoded within clauses [11]. Moreover, cases where no clause of Γ is involved in the inconsistency of the instance appear to be the worst case of the algorithm. However, if Γ is exclusively composed of never necessary clauses, then Γ can be satisfied by an *autarky* [21], defined as a partial interpretation satisfying each clause having one literal assigned. An algorithm [22] has been recently proposed for finding autarkies by modifying the CNF formula and considering an optimization problem.

7 Conclusions and Future Works

In this paper, an original "abstract-and-refine" technique for checking whether a set of clauses overlaps with at least one minimal unsatisfiable subset of a SAT instance has been described. Such an issue can prove useful when a user needs to restore the consistency of a CNF formula and has (or can get) some hints about where the conflicting clauses are located in the formula.

The approach proves orders of magnitude more efficiently than existing approaches, thanks to its step-by-step abstraction and reformulation paradigm. Especially, it does not require the whole set of MUSes of the instance to be computed. Another feature of the approach lies in its any-time character, allowing "rough" solutions to be provided when the preset computational resources are exhausted.

Interesting paths for future reseach include the investigation of several possible variants for the approach. For instance, a crucial point lies in the way according to which the clauses are clustered. Although the failed local-search heuristic and the concept of critical clauses proved to be efficient, it could be fruitful to investigate other forms of interactions between clauses in order to form clusters. Particularly, various techniques have been proposed to divide SAT instances into subinstances, and it would be interesting to study the viability of such approaches to form clusters. In addition, this study was focused on the "flat" Boolean framework where all clauses share a same importance. Users may have some qualitative or quantitative preferences about the information contained in Σ. In this case, they might want to extract the MUSC that best fits those preferences. In the future, we plan to investigate those issues.

Acknowledgement

The authors gratefully thank the anonymous reviewers for their insightful comments. This work is partly supported by the "IUT de Lens".

References

1. Eén, N., Sörensson, N.: Minisat, `http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat`
2. Biere, A.: Boolforce, `http://fmv.jku.at/booleforce`

3. Huang, J.: MUP: A minimal unsatisfiability prover. In: ASP-DAC 2005, Shanghai, China, pp. 432–437 (2005)
4. van Maaren, H., Wieringa, S.: Finding guaranteed MUSes fast. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 291–304. Springer, Heidelberg (2008)
5. Grégoire, É., Mazure, B., Piette, C.: On approaches to explaining infeasibility of sets of Boolean clauses. In: ICTAI 2008, vol. 1, pp. 74–83 (2008)
6. Eiter, T., Gottlob, G.: On the complexity of propositional knowledge base revision, updates and counterfactual. Artificial Intelligence 57, 227–270 (1992)
7. Zhang, L., Malik, S.: Extracting small unsatisfiable cores from unsatisfiable Boolean formula. In: SAT 2003, Portofino, Italy (2003)
8. Grégoire, É., Mazure, B., Piette, C.: Local-search extraction of MUSes. Constraints Journal 12(3), 325–344 (2007)
9. Shapley, L.: A value for n-person games. Contributions to the Theory of Games II (Annals of Mathematics Studies 28), 307–317 (1953)
10. Kullmann, O., Lynce, I., Marques Silva, J.: Categorisation of clauses in conjunctive normal forms: Minimally unsatisfiable sub-clause-sets and the lean kernel. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 22–35. Springer, Heidelberg (2006)
11. Liffiton, M., Sakallah, K.: Algorithms for computing minimal unsatisfiable clause sets. Journal of Automated Reasoning 40(1), 1–33 (2008)
12. Grégoire, É., Mazure, B., Piette, C.: Boosting a complete technique to find MSS and MUS thanks to a local search oracle. In: IJCAI 2007, vol. 2, pp. 2300–2305. AAAI Press, Menlo Park (2007)
13. Hunter, A., Konieczny, S.: Measuring inconsistency through minimal inconsistent sets. In: KR 2008, Sydney, Australia, pp. 358–366 (2008)
14. Schaerf, M., Cadoli, M.: Tractable reasoning via approximation. Artificial Intelligence 74, 249–310 (1995)
15. Dalal, M.: Anytime families of tractable propositional reasoners. In: AI/MATH 1996, pp. 42–45 (1996)
16. Dalal, M.: Semantics of an anytime family of reasponers. In: ECAI 1996, pp. 360–364 (1996)
17. Finger, M.: Towards polynomial approximations of full propositional logic. In: Bazzan, A.L.C., Labidi, S. (eds.) SBIA 2004. LNCS (LNAI), vol. 3171, pp. 11–20. Springer, Heidelberg (2004)
18. Finger, M., Wassermann, R.: The universe of propositional approximations. Theoretical Computer Science 355(2), 153–166 (2006)
19. Koriche, F., Sallantin, J.: A logical toolbox for knowledge approximation. In: TARK 2001, pp. 193–205. Morgan Kaufmann Publishers Inc., San Francisco (2001)
20. Koriche, F.: Approximate coherence-based reasoning. Journal of Applied Non-Classical Logics 12(2), 239–258 (2002)
21. Kullmann, O.: Investigations on autark assignments. Discrete Applied Mathematics 107, 99–137 (2000)
22. Liffiton, M., Sakallah, K.: Searching for autarkies to trim unsatisfiable clause sets. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 182–195. Springer, Heidelberg (2008)

Progress in the Development of Automated Theorem Proving for Higher-Order Logic*

Geoff Sutcliffe[1], Christoph Benzmüller[2], Chad E. Brown[3], and Frank Theiss[3,**]

[1] University of Miami, USA***
[2] International University, Germany
[3] Saarland University, Germany

Abstract. The Thousands of Problems for Theorem Provers (TPTP) problem library is the basis of a well established infrastructure supporting research, development, and deployment of first-order Automated Theorem Proving (ATP) systems. Recently, the TPTP has been extended to include problems in higher-order logic, with corresponding infrastructure and resources. This paper describes the practical progress that has been made towards the goal of TPTP support for higher-order ATP systems.

1 Motivation and History

There is a well established infrastructure that supports research, development, and deployment of first-order Automated Theorem Proving (ATP) systems, stemming from the Thousands of Problems for Theorem Provers (TPTP) problem library [38]. This infrastructure includes the problem library itself, the TPTP language [36], the SZS ontologies [35], the Thousands of Solutions from Theorem Provers (TSTP) solution library, various tools associated with the libraries [34], and the CADE ATP System Competition (CASC) [37]. This infrastructure has been central to the progress that has been made in the development of high performance first-order ATP systems.

Until recently there has been no corresponding support in higher-order logic. In 2008, work commenced on extending the TPTP to include problems in higher-order logic, and developing the corresponding infrastructure and resources. These efforts aim to have an analogous impact on the development of higher-order ATP systems. The key steps have been:

* This research has received funding from the European Community's Seventh Framework Programme FP7/2007-2013, under grant agreement PIIF-GA-2008-219982.

** Supported by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft XT project under grant 01 IS 07 008.

*** Most of the work was done while hosted as a guest of the Automation of Logic group in the Max Planck Institut für Informatik.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 116–130, 2009.

- Development of the Typed Higher-order Form (THF) part of the TPTP language, compatible with the existing first-order forms (FOF and CNF).
- Collecting THF problems, for the TPTP.
- Building TPTP infrastructure for THF problems.
- Finding and implementing ATP systems for higher-order logic.
- Collecting ATP systems' solutions to the THF problems, for the TSTP.
- Planning for a THF division of CASC.

These topics are described in this paper.

2 The Higher-Order TPTP

The development of the higher-order part of the TPTP involved three of the steps identified in the introduction: development of the Typed Higher-order Form (THF) part of the TPTP language, collecting THF problems for the TPTP, and building TPTP infrastructure for THF problems. These steps are described in this section.

2.1 The Typed Higher-Order Form (THF) Language

The TPTP language is a human-readable, easily machine-parsable, flexible and extensible language, suitable for writing both ATP problems and solutions. A particular feature of the TPTP language, which has been maintained in the THF part, is Prolog compatibility. As a result the development of reasoning software (for TPTP data) in Prolog has a low entry barrier [36]. The top level building blocks of the TPTP language are *include directives* and *annotated formulae*. Include directives are used mainly in problem files to include the contents of axiom files. Annotated formulae have the form:

language(*name*, *role*, *formula*, *source*, *useful_info*).

The *language*s supported are first-order form (`fof`), clause normal form (`cnf`), and now typed higher-order form (`thf`). The *role* gives the user semantics of the *formula*, e.g., `axiom`, `lemma`, `type`, `definition`, `conjecture`, and hence defines its use in an ATP system. The logical *formula* uses a consistent and easily understood notation, and uses only standard ASCII characters.

The THF language for logical formulae is a syntactically conservative extension of the existing first-order TPTP language, adding constructs for higher-order logic. Maintaining a consistent style between the first-order and higher-order languages allows easy adoption of the new higher-order language, through reuse or adaptation of existing infrastructure for processing TPTP format data, e.g., parsers, pretty-printing tools, system testing harnesses, and output processors. The THF language has been divided into three layers: THF0, THF, and THFX. The THF0 core language is based on Church's simple type theory, and provides the commonly used and accepted aspects of a higher-order logic language [13]. The full THF language drops the differentiation between terms and types, thus providing a significantly richer type system, and adds the ability to reason about types. It additionally offers more term constructs, more

type constructs, and more connectives. The extended THFX language adds constructs that are "syntactic sugar", but are usefully expressive. Initially the TPTP will contain higher-order problems in only THF0, to allow users to adopt the language without being swamped by the richness of the full THF language. The full THF language definition is available from the TPTP web site, `www.tptp.org`.

Figures 1 and 2 show an example of a TPTP problem file in THF0. The `thf` annotated formulae in Figure 1 illustrate common constructs in the THF0 language. The constructs that are not part of the first-order TPTP language are:

- The typing of constants and quantified variables. THF requires that all symbols be typed. Constants are globally typed in an annotated formula with role `type`, and variables are locally typed in their quantification (thus requiring all variables to be quantified).
- `$tType` for the collection of all types.
- `$i` and `$o` for the types of individuals and propositions. `$i` is non-empty, and may be finite or infinite.
- `>` for (right associative) function types.
- `^` as the lambda binder.
- `@` for (left associative) application.

Additional THF constructs, not shown in the example, are:

- The use of connectives as terms (THF0), e.g.,
  `(& @ $false) = (^ [P:$o] : $false)`
- `!!` and `??` for the Π (forall) and Σ (exists) operators (THF0), e.g.,
  `((!! (p)) & (!! (q))) = (! [X:$i] : (( p @ X) & (q @ X)))`
- `!>` and `?*` for Π (dependent product) and Σ (sum) types (THF), e.g.,
  `cons: !> [N:nat] : ($i > (list @ N) > (list @ (succ @ N)))`
- `[ ]` for tuples (THF), e.g.,
  `make_triple = ^ [X:$i,Y:$i:,Z:$i] : [X,Y,Z]`
- `*` and `+` for simple product and sum (disjoint union) types (THF), e.g.,
  `roots: quadratic > (($real * $real) + $real + undef)`
- `:=` as a connective for global definitions, and as a binder and separator for local definitions (ala `letrec`) (THFX), e.g.,
  `apply_twice:= ^ [F:$o > $o,X:$o] : (F @ (F @ X))`
  defines `apply_twice` with global scope, and
  `:= [NN:= (apply_twice @ ~)] : (NN = (apply_twice @ NN))`
  has `NN` defined with local (the formula) scope.
- `-->` as the sequent connective (THFX), e.g., `[p,q,r] --> [s,t]`

The choice of semantics for THF problems is of interest, as, unlike the first-order case, there are different options [6,7]. For THF0 the default is Henkin semantics with extensionality (without choice or description). The default semantics of the higher THF layers have not been fixed yet, since this choice will be dependent on which systems adopt the higher-order TPTP.

```
%------------------------------------------------------------------------------
%----Include simple maths definitions and axioms
include('Axioms/LCL008^0.ax').
%------------------------------------------------------------------------------
thf(a,type,(
    a: $tType )).

thf(p,type,(
    p: ( a > $i > $o ) > $i > $o )).

thf(g,type,(
    g: a > $i > $o )).

thf(e,type,(
    e: ( a > $i > $o ) > a > $i > $o )).

thf(r,type,(
    r: $i > $i > $o )).

thf(mall_aio,type,(
    mall_aio: ( ( a > $i > $o ) > $i > $o ) > $i > $o )).

thf(mall_a,type,(
    mall_a: ( a > $i > $o ) > $i > $o )).

thf(mall_aio,definition,
    ( mall_aio
    = ( ^ [P: ( a > $i > $o ) > $i > $o,W: $i] :
        ! [X: a > $i > $o] :
          ( P @ X @ W ) ) )).

thf(mall_a,definition,
    ( mall_a
    = ( ^ [P: a > $i > $o,W: $i] :
        ! [X: a] :
          ( P @ X @ W ) ) )).

thf(positiveness,axiom,
    ( mvalid
    @ ( mall_aio
      @ ^ [X: a > $i > $o] :
          ( mimpl @ ( mnot @ ( p @ X ) )
          @ ( p
            @ ^ [Z: a] :
                ( mnot @ ( X @ Z ) ) ) ) ) ) )).

thf(g,definition,
    ( g
    = ( ^ [Z: a] :
          ( mall_aio
          @ ^ [X: a > $i > $o] :
              ( mimpl @ ( p @ X ) @ ( X @ Z ) ) ) ) ) )).

thf(e,definition,
    ( e
    = ( ^ [X: a > $i > $o,Z: a] :
          ( mall_aio
          @ ^ [Y: a > $i > $o] :
              ( mimpl @ ( Y @ Z )
              @ ( mbox @ r
                @ ( mall_a
                  @ ^ [W: a] :
                      ( mimpl @ ( X @ W ) @ ( Y @ W ) ) ) ) ) ) ) ) )).

thf(thm,conjecture,
    ( mvalid
    @ ( mall_a
      @ ^ [Z: a] :
          ( mimpl @ ( g @ Z ) @ ( e @ g @ Z ) ) ) ) )).

%------------------------------------------------------------------------------
```

Fig. 1. LCL634^1 formulae

```
%------------------------------------------------------------------------------
% File     : LCL634^1 : TPTP v3.7.0. Released v3.6.0.
% Domain   : Logical Calculi
% Problem  : Goedel's ontological argument on the existence of God
% Version  : [Ben08] axioms : Especial.
% English  :

% Refs     : [Fit00] Fitting (2000), Higher-Order Modal Logic - A Sketch
%          : [Ben08] Benzmueller (2008), Email to G. Sutcliffe
% Source   : [Ben08]
% Names    : Fitting-HOLML-Ex-God-alternative-b [Ben08]

% Status   : Theorem
% Rating   : 1.00 v3.7.0
% Syntax   : Number of formulae    :   48 (   3 unit;  27 type;  19 defn)
%            Number of atoms       :  323 (  19 equality;  60 variable)
%            Maximal formula depth :   13 (   5 average)
%            Number of connectives :   71 (   3   ~;   1   |;   2   &;  64   @)
%                                         (   0 <=>;   1  =>;   0  <=;   0 <~>)
%                                         (   0  ~|;   0  ~&;   0  !!;   0  ??)
%            Number of type conns  :  118 ( 118   >;   0   *;   0   +)
%            Number of symbols     :   28 (  27   :;   0  :=)
%            Number of variables   :   51 (   2 sgn;   6   !;   4   ?;  41   ^)
%                                         (  51   :;   0  :=;   0  !>;   0  ?*)

% Comments :
%------------------------------------------------------------------------------
```

Fig. 2. `LCL634^1` header

The first section of each TPTP problem file is a header that contains information for the user. Figure 2 shows the header for the annotated formulae of Figure 1. This information is not for use by ATP systems. It is divided into four parts. The first part identifies and describes the problem, the second part provides information about occurrences of the problem in the literature and elsewhere, the third part gives the problem's status as an SZS ontology value [35] and a table of syntactic measurements made on the problem, and the last part contains general comments about the problem. The status value is for the default semantics – Henkin semantics with extensionality. If the status is known to be different for other semantics, e.g., without functional/Boolean extensionality, or with addition of choice or description, this is provided on subsequent lines, with the modified semantics noted.

2.2 Collecting THF Problems, for the TPTP

The THF problems collected in the second half of 2008 and first quarter of 2009 were part of TPTP v3.7.0, which was released on 8th March 2009. This was a beta release of the THF part of the TPTP, and contained higher-order problems in only the THF0 language. There were 1275 THF problem versions, stemming from 852 abstract problems, in nine domains:

– `ALG` - 50 problems. These are problems concerning higher-order abstract syntax, encoded in higher-order logic [19].

- GRA - 93 problems. These are problems about Ramsey numbers, some of which are open in the mathematics community.
- LCL - 56 problems. These are of modal logic problems that have been encoded in higher-order logic.
- NUM - 221 problems. These are mostly theorems from Jutting's AUTOMATH formalization [40] of the well known Landau book [24]. These are also some Church numeral problems.
- PUZ - 5 problems. These are "knights and knaves" problems.
- SET and SEU - 749 problems. Many of these are "standard" problems in set theory that have TPTP versions in first-order logic. This allows for an evaluation of the relative benefits of the different encodings with respect to ATP systems for the logics [14]. There is also a significant group of problems in dependently typed set theory [17], and a group of interesting problems about binary relations.
- SWV - 37 problems. The two main groups of problems are (i) security problems in access control logic, initially encoded in modal logic, and subsequently encoded in higher-order logic [9], and (ii) problems about security in an authorization logic that can be converted via modal logic to higher-order logic [20].
- SYN - 59 problems. These are simple problems designed to test properties of higher-order ATP systems [6].

1038 of the problems (81%) contain equality. 1172 of the problems (92%) are known or believed to be theorems, 28 (2%) are known or believed to be non-theorems, and the remaining 75 problems (6%) have unknown status. Table 1 provides some further detailed statistics about the problems.

Table 1. Statistics for THF problems

	Min	Max	Avg	Median
Number of formulae	1	749	118	16
% of unit formulae	0%	60%	24%	27%
Number of atoms	2	7624	1176	219
% of equality atoms	0%	33%	5%	6%
in problems with equality	1%	33%	7%	7%
% of variable atoms	0%	82%	33%	33%
Avg atoms per formula	1.8	998.0	50.0	8.0
Number of symbols	1	390	66	12
Number of variables	0	1189	182	31
^	0	175	22	5
!	0	1067	150	11
?	0	45	10	2
Number of connectives	0	4591	677	73
Number of type connectives	0	354	62	28
Maximal formula depth	2	351	67	14
Average formula depth	2	350	16	6

2.3 TPTP Infrastructure for THF Problems

The first-order TPTP provides a range of resources to support use of the problem library [34]. Many of these resources are immediately applicable to the higher-order setting, while some have required changes for the new features of the THF language.

From a TPTP user perspective, the TPTP2X utility distributed with the TPTP will initially be most useful for manipulating THF problems. TPTP2X has been extended to read, manipulate, and output (pretty print) data in the full THF language. Additionally, format modules for outputting problems in the TPS [4], Twelf [28], OmDoc [23], Isabelle [27], and S-expression formats have been implemented. The TPTP4X tool has also been extended to read, manipulate, and output data in the THF0 language, and will be extended to the full THF language.

The SystemOnTPTP utility for running ATP systems and tools on TPTP problems and solutions has been updated to deal with THF data, including use of the new higher-order formats output by TPTP2X. The online interface to SystemOnTPTP (`www.tptp.org/cgi-bin/SystemOnTPTP`) has also been updated to deal with THF data, and includes ATP systems and tools for THF data.

Internally, an important resource is the Twelf-based type checking of THF problems, implemented by exporting a problem in Twelf format, and submitting the result to the Twelf tool - see [13] for details.

The BNF based parsers for the TPTP [41] naturally parse the full THF language, and the `lex/yacc` files used to build these parsers are freely available.

3 Collecting Solutions to THF Problems, for the TSTP

The Thousands of Solutions from Theorem Provers (TSTP) solution library, the "flip side" of the TPTP, is a corpus of contemporary ATP systems' solutions to the TPTP problems. A major use of the TSTP is for ATP system developers to examine solutions to problems, and thus understand how they can be solved. The TSTP is built using a harness that calls the SystemOnTPTP utility, and thus leverages many aspects of the TPTP infrastructure for THF data.

Four higher-order ATP systems, LEO-II 0.99a, TPS 3.0, and two automated versions of Isabelle 2008 (one - IsabelleP - trying to prove theorems, the other - IsabelleM - trying to find (counter-)models), have been run over the 1275 THF problems in TPTP v3.7.0, and their results added to the TSTP. The systems are described in Section 4. Table 2 tabulates the numbers of problems solved. The "Any", "All", and "None" rows are with respect to to the three theorem provers. All the runs were done on 2.80GHz computers with 1GB memory and running the Linux operating system, with a 600s CPU limit.

The results show that the `GRA` Ramsey number problems are very difficult - this was expected. For the remaining domains the problems pose interesting challenges for the ATP systems, and the differences between the systems lead to different problems being solved, including some that are solved uniquely by each of the systems.

Table 2. Results for THF problems

	ALG	GRA	LCL	NUM	PUZ	SE?	SWV	SYN	Total	Unique
Problems	50	93	61	221	5	749	37	59	1275	
LEO-II 0.99a	34	0	48	181	3	401	19	42	725	127
IsabelleP 2008	0	0	0	197	5	361	1	30	594	74
TPS 3.0	10	0	40	150	3	285	9	35	532	6
Any	32	0	50	203	5	490	20	52	843	207
All	0	0	0	134	2	214	0	22	372	
None	18	93	12	18	0	259	17	15	432	
IsabelleM 2008	0	0	1	0	0	0	0	8	9	

4 Higher-Order ATP for the TPTP

Research and development of computer-supported reasoning for higher-order logic has been in progress for as long as that for first-order logic. It is clear that the computational issues in the higher-order setting are significantly harder than those in first-order. Problems such as the undecidability of higher-order unification, the handling of equality and extensionality reasoning, and the instantiation of set variables, have hampered the development of effective higher-order automated reasoning. Thus, while there are many *interactive* proof assistants based on some form of higher-order logic [43], there are few *automated* systems for higher-order logic. This section describes the three (fully automatic) higher-order ATP systems that we know of.

4.1 LEO-II

LEO-II [12] is a resolution based higher-order ATP system. It is the successor of LEO [8], which was implemented in LISP and hardwired to the OMEGA proof assistant [32]. LEO-II is implemented in Objective Caml, and is freely available from `http://www.ags.uni-sb.de/~leo/` under a BSD-like licence.

LEO-II is designed to cooperate with specialist systems for fragments of higher-order logic. The idea is to combine the strengths of the different systems: LEO-II predominantly addresses higher-order aspects in its reasoning process, with the aim of quickly removing higher-order clauses from the search space, and turning them into first-order clauses that can be refuted with a first-order ATP system. Currently, LEO-II is capable of cooperating with the first-order ATP systems E [31], SPASS [42], and Vampire [30].

In addition to a fully automatic mode, LEO-II provides an interactive mode [11]. This mode supports debugging and inspection of the search space, and also the tutoring of resolution based higher-order theorem proving to students. The interactive mode and the automatic mode can be interleaved.

LEO-II directly parses THF0 input. THF0 is the only input syntax supported by LEO-II. The THF problem collection has been a valuable testbed in this respect, providing examples that exposed intricacies of the LEO-II parser. Some problems revealed differing precedences for logical connectives in THF0 and LEO-II.

Instead of generating parsing errors these examples led to different semantic interpretations. An example is the tautologous axiom `! [X:$o]:(~(X) | X)`. Due to mistaken operator precedences, LEO-II used to (mis)read this axiom as `! [X:$o]:(~(X | X))`.

Communication between LEO-II and the cooperating first-order ATP system uses TPTP standards. LEO-II's clause set generally consists of higher-order clauses that are processed with LEO-II's calculus rules. Some of the clauses in LEO-II's search space additionally attain a special status: they are first-order clauses modulo the application of an appropriate transformation function. The default transformation is Hurd's fully typed translation [22]. LEO-II's extensional higher-order resolution approach enhances standard resolution proof search with specific extensionality rules that generate more and more essentially first-order clauses from higher-order ones. LEO-II is often too weak to find a refutation amongst the steadily growing set of essentially first-order clauses on its own. Therefore, LEO-II launches the cooperating first-order ATP system every n iterations of its (standard) resolution proof search loop (currently $n = 10$). The subproblem passed to the first-order ATP system is written in the TPTP language. If the first-order ATP system finds a refutation and communicates its success to LEO-II in the standard SZS format: `SZS status Unsatisfiable`. LEO-II analyzes this answer and recognizes the reference to unsatisfiablity in the SZS ontology. LEO-II stops the proof search and reports that the problem is a theorem, in the standard SZS format: `SZS status Theorem`.

Debugging of LEO-II benefits from the examples in the TPTP library. Several bugs in LEO-II, beyond the parsing bugs described above, have been detected through use of the TPTP library. These include problems in the translation from higher-order to first-order form, and accidentally omitted type checks in the higher-order unification algorithm. The library has thus provided an excellent basis for finding and curing various "Kinderkrankheiten" that a new ATP system inevitably experiences.

Future work includes further exploitation of TPTP infrastructure. An important step will be the integration of TPTP format proofs output by the cooperating first-order ATP system into LEO-II's higher-order resolution proofs. The goal is to produce a single, coherent proof in the TPTP language.

4.2 TPS

TPS [4] is a higher-order theorem proving system that has been developed under the supervision of Peter B. Andrews since the 1980s. Theorems can be proven in TPS either interactively or automatically. Some of the key ingredients of the automated search procedures of TPS are mating search [2] (which is similar to Bibel's connection method [15]), Miller's expansion trees [25], and Huet's higher-order pre-unification [21]. In essence, the goal of each search procedure is to find an appropriate set of connections (i.e., a complete mating), and to find appropriate instantiations for certain variables [3].

In TPS there are flags that can be set to affect the behavior of automated search. A collection of flag settings is called a *mode*. The mode determines which particular search procedure will be used, as well as how the exploration of the search space should be ordered. Over 500 modes are available in the TPS library. The two modes considered in this paper are `MS98-FO-MODE` and `BASIC-MS04-2-MODE`.

The mode `MS98-FO-MODE` uses a search procedure `MS98-1`, implemented by Matthew Bishop [16]. The procedure precomputes components (compatible sets of connections) and then attempts to combine the components to construct a complete mating. This approach enables TPS to solve a number of problems that were too hard for earlier search procedures. `MS98-1` and all earlier search procedures are based on Miller's expansion trees. Consequently, the procedures attempt to find proofs that do not use extensionality, i.e., these search procedures can prove theorems of only elementary type theory [1].

The mode `BASIC-MS04-2-MODE` uses a search procedure `MS04-2`, implemented by Chad Brown [18]. Unlike `MS98-1`, `MS04-2` can find proofs of theorems requiring extensionality. `MS04-2` is the only TPS search procedure that is complete relative to Henkin semantics [18]. The procedure is based on extensional expansion DAGs, a generalization of Miller's expansion trees. The trees become DAGs because connections can generate new nodes that are children of the two connected nodes. For theorems that do not require extensionality, this extra complication can expand the search space unnecessarily. Also, `MS04-2` relies on backtracking in a way `MS98-1` does not.

As the two TPS modes have quite different capabilities, and it is expected that any proofs found by either mode will be found quickly, running the two modes in competition parallel is a simple way of obtaining greater coverage. A simple `perl` script has ben used to do this, running two copies of TPS in parallel as separate UNIX processes, one for each of the modes. As soon as either process finds a proof, the script terminates the other. It was this competition parallel version of TPS that produced the 532 proofs noted in Table 2. Analysis of the system's outputs shows that the parallelism is effective, with the two modes each solving about half of the problems (first). This indicates that the TPTP has a good balance of problems with respect to these two TPS modes. A new strategy scheduling version of TPS is currently being developed, which will run many more modes, but in sequence to avoid memory contention.

4.3 IsabelleP and IsabelleM

Isabelle [27] is a well known proof assistant for higher-order logic. It is normally used interactively through the Proof General interface [5]. In this mode it is possible to apply various automated tactics that attempt to solve the current goal without further user interaction. Examples of these tactics are `blast`, `auto`, and `metis`. It is (a little known fact that it is) also possible to run Isabelle from the command line, passing in a theory file with a `lemma` to solve. Finally, Isabelle theory files can include ML code to be executed when the file is processed. These three features have been combined to implement a fully automatic Isabelle, using

the nine tactics `simp`, `blast`, `auto`, `metis`, `fast`, `fastsimp`, `best`, `force`, and `meson`. The TPTP2X Isabelle format module outputs a THF problem in Isabelle HOL syntax, augmented with ML code that (i) runs the nine tactics in sequence, each with a CPU time limit, until one succeeds or all fail, and (ii) reports the result and proof (if found) using the SZS standards. A `perl` script is used to insert the CPU time limit (equally divided over the nine tactics) into TPTP2X's Isabelle format output, and then run the command line `isabelle-process` on the resulting theory file. The complete system is named IsabelleP in Table 2.

While it was probably never intended to use Isabelle as a fully automatic system, this simple automation provides useful capability. It solves 74 problems that neither LEO-II nor Tps can solve. The strategy scheduling is effective, with eight of the modes contributing solutions. Over 400 of the 594 solutions are found by one of the first two tactics used - `simp` or `blast`, and more than another 100 by one of the next three tactics - `auto`, `metis`, or `fast`. Further research and development of this automated Isabelle will inevitably lead to improved performance.

The ability of Isabelle to find (counter-)models using the `refute` command has also been integrated into an automatic system, called IsabelleM in Table 2. This provides the TPTP with capability to confirm the satisfiability of axiom sets, and the countersatisfiability of non-theorems. It has been useful for exposing errors in some THF problem encodings. It is planned to extend IsabelleM to also use the (the newly developed) `nitpick` command for model finding.

5 Cunning Plans for the Future

CASC: The CADE ATP System Competition (CASC) [37] is held annually at each CADE (or IJCAR, of which CADE is a constituent) conference. CASC evaluates the performance of sound, fully automatic, ATP systems – it is the world championship for such systems. CASC has been a catalyst for impressive improvements in ATP, stimulating both theoretical and implementation advances [26]. The addition of a THF division to CASC is planned as a natural way to provide the same stimulation for the development of higher-order ATP systems. The first THF division of CASC will be part of CASC-22 at CADE-22. While the primary purpose of CASC is a public evaluation of the relative capabilities of ATP systems, it is important that the THF division should strongly focus on the other aims of CASC: to stimulate ATP research in general, to stimulate ATP research towards autonomous systems, to motivate implementation of robust ATP systems, to provide an inspiring environment for personal interaction between ATP researchers, and to expose ATP systems within and beyond the ATP community.

THF and THFX: Currently the TPTP contains problems in only the core THF0 fragment of the THF language. As ATP developers and users adopt the language, it is anticipated that demand for the richer features of the full THF language and the extended THFX language will quickly emerge. In preparation for this demand the THF and THFX languages have already been defined,

problems in these languages are being collected, and TPTP infrastructure for processing these problems is being developed. Thus the higher-order TPTP expects to be able to meet the expectations of the community, hence encouraging uptake of the THF language and use of the TPTP problems as a common basis for system evaluation.

6 Conclusion

This paper has described the significant practical progress that has been made towards developing the TPTP and associated infrastructure for automated reasoning in higher-order logic. An alpha-release of the TPTP (v3.6.0) with higher-order problems was made on 25th December 2008 (a Christmas present), a beta-release (v3.7.0) was made on 8th March 2009, and the first full release (v4.0.0) will be made in August 2009.

The core work of collecting THF problems is proceeding. Significant new contributions have come from the export of the TPS problem library [4] to THF0, and from a higher-order encoding [10] of problems from the Intuitionistic Logic Theorem Proving (ILTP) library [29]. TPTP v4.0.0 will have over 2500 THF problems.

Current work on the TPTP infrastructure is extending the Java parser for the TPTP language to read the THF language. This in turn will allow use of Java based tools, e.g., IDV [39], for manipulating THF data. The semantic derivation verifier GDV [33] is being updated to verify proofs that include formulae in the THF language.

A key goal of this work is to stimulate the development of ATP systems for higher-order logic - there are many potential applications for such systems. ATP systems that output proofs are particularly important, allowing proof verification. In the long term we hope to see burgeoning research and development of ATP for higher-order logic, with a richness similar to first-order ATP, with many ATP systems, common usage in applications, meta-systems, etc.

Acknowledgments. Thanks to Florian Rabe for help with the Twelf and OmDoc TPTP2X output formats, and to Lucas Dixon and Stefan Berghofer for help with the Isabelle format. Thanks to Florian Rabe for implementing the Twelf-based type checking. Thanks to Makarius Wenzel and Stefan Berghofer for writing the IsabelleP code. Thanks to Jasmin Blanchette for explaining how to implement IsabelleM. Thanks to Andrei Tchaltsev and Alexandre Riazanov for developing the first-order parts of the Java parser for the TPTP language.

References

1. Andrews, P.B.: Resolution in Type Theory. Journal of Symbolic Logic 36(3), 414–432 (1971)
2. Andrews, P.B.: Theorem Proving via General Matings. Journal of the ACM 28(2), 193–214 (1981)

3. Andrews, P.B.: On Connections and Higher-Order Logic. Journal of Automated Reasoning 5(3), 257–291 (1989)
4. Andrews, P.B., Brown, C.E.: TPS: A Hybrid Automatic-Interactive System for Developing Proofs. Journal of Applied Logic 4(4), 367–395 (2006)
5. Aspinall, D.: Proof General: A Generic Tool for Proof Development. In: Graf, S., Schwartzbach, M. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 38–42. Springer, Heidelberg (2000)
6. Benzmüller, C., Brown, C.E.: A Structured Set of Higher-Order Problems. In: Hurd, J., Melham, T. (eds.) TPHOLs 2005. LNCS (LNAI), vol. 3603, pp. 66–81. Springer, Heidelberg (2005)
7. Benzmüller, C., Brown, C.E., Kohlhase, M.: Higher-order Semantics and Extensionality. Journal of Symbolic Logic 69(4), 1027–1088 (2004)
8. Benzmüller, C., Kohlhase, M.: LEO - A Higher-Order Theorem Prover. In: Kirchner, C., Kirchner, H. (eds.) CADE 1998. LNCS (LNAI), vol. 1421, pp. 139–143. Springer, Heidelberg (1998)
9. Benzmüller, C., Paulson, L.: Exploring Properties of Normal Multimodal Logics in Simple Type Theory with LEO-II. In: Benzmüller, C., Brown, C.E., Siekmann, J., Statman, R. (eds.) Reasoning in Simple Type Theory: Festschrift in Honour of Peter B. Andrews on his 70th Birthday. Studies in Logic, Mathematical Logic and Foundations, vol. 17. College Publications (2009)
10. Benzmüller, C., Paulson, L.: Exploring Properties of Propositional Normal Multimodal Logics and Propositional Intuitionistic Logics in Simple Type Theory. Journal of Symbolic Logic (submitted)
11. Benzmüller, C., Paulson, L., Theiss, F., Fietzke, A.: Progress Report on LEO-II - An Automatic Theorem Prover for Higher-Order Logic. In: Schneider, K., Brandt, J. (eds.) Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics, pp. 33–48 (2007)
12. Benzmüller, C., Paulson, L., Theiss, F., Fietzke, A.: LEO-II - A Cooperative Automatic Theorem Prover for Higher-Order Logic. In: Baumgartner, P., Armando, A., Gilles, D. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 162–170. Springer, Heidelberg (2008)
13. Benzmüller, C., Rabe, F., Sutcliffe, G.: THF0 - The Core TPTP Language for Classical Higher-Order Logic. In: Baumgartner, P., Armando, A., Gilles, D. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 491–506. Springer, Heidelberg (2008)
14. Benzmüller, C., Sorge, V., Jamnik, M., Kerber, M.: Combined Reasoning by Automated Cooperation. Journal of Applied Logic 6(3) (2008) (to appear)
15. Bibel, W.: On Matrices with Connections. Journal of the ACM 28(4), 633–645 (1981)
16. Bishop, M.: A Breadth-First Strategy for Mating Search. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 359–373. Springer, Heidelberg (1999)
17. Brown, C.E.: Dependently Typed Set Theory. Technical Report SWP-2006-03, Saarland University (2006)
18. Brown, C.E.: Automated Reasoning in Higher-Order Logic: Set Comprehension and Extensionality in Church's Type Theory. Studies in Logic: Logic and Cognitive Systems, vol. 10. College Publications (2007)
19. Brown, C.E.: M-Set Models. In: Benzmüller, C., Brown, C.E., Siekmann, J., Statman, R. (eds.) Reasoning in Simple Type Theory: Festschrift in Honour of Peter B. Andrews on his 70th Birthday. Studies in Logic, Mathematical Logic and Foundations, vol. 17. College Publications (2009)

20. Garg, D.: Principal-Centric Reasoning in Constructive Authorization Logic. Technical Report CMU-CS-09-120, School of Computer Science, Carnegie Mellon University (2009)
21. Huet, G.: A Unification Algorithm for Typed Lambda-Calculus. Theoretical Computer Science 1(1), 27–57 (1975)
22. Hurd, J.: First-Order Proof Tactics in Higher-Order Logic Theorem Provers. In: Archer, M., Di Vito, B., Munoz, C. (eds.) Proceedings of the 1st International Workshop on Design and Application of Strategies/Tactics in Higher Order Logics. NASA Technical Reports, number NASA/CP-2003-212448, pp. 56–68 (2003)
23. Kohlhase, M.: OMDoc - An Open Markup Format for Mathematical Documents [version 1.2]. LNCS (LNAI), vol. 4180. Springer, Heidelberg (2006)
24. Landau, E.: Grundlagen der Analysis. Akademische Verlagsgesellschaft M.B.H. (1930)
25. Miller, D.: A Compact Representation of Proofs. Studia Logica 46(4), 347–370 (1987)
26. Nieuwenhuis, R.: The Impact of CASC in the Development of Automated Deduction Systems. AI Communications 15(2-3), 77–78 (2002)
27. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
28. Pfenning, F., Schürmann, C.: System Description: Twelf - A Meta-Logical Framework for Deductive Systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)
29. Raths, T., Otten, J., Kreitz, C.: The ILTP Problem Library for Intuitionistic Logic - Release v1.1. Journal of Automated Reasoning 38(1-2), 261–271 (2007)
30. Riazanov, A., Voronkov, A.: The Design and Implementation of Vampire. AI Communications 15(2-3), 91–110 (2002)
31. Schulz, S.: E: A Brainiac Theorem Prover. AI Communications 15(2-3), 111–126 (2002)
32. Siekman, J., Benzmüller, C., Autexier, S.: Computer Supported Mathematics with OMEGA. Journal of Applied Logic 4(4), 533–559 (2006)
33. Sutcliffe, G.: Semantic Derivation Verification. International Journal on Artificial Intelligence Tools 15(6), 1053–1070 (2006)
34. Sutcliffe, G.: TPTP, TSTP, CASC, etc. In: Diekert, V., Volkov, M., Voronkov, A. (eds.) CSR 2007. LNCS, vol. 4649, pp. 7–23. Springer, Heidelberg (2007)
35. Sutcliffe, G.: The SZS Ontologies for Automated Reasoning Software. In: Sutcliffe, G., Rudnicki, P., Schmidt, R., Konev, B., Schulz, S. (eds.) Proceedings of the LPAR Workshops: Knowledge Exchange: Automated Provers and Proof Assistants, and The 7th International Workshop on the Implementation of Logics. CEUR Workshop Proceedings, vol. 418, pp. 38–49 (2008)
36. Sutcliffe, G., Schulz, S., Claessen, K., Van Gelder, A.: Using the TPTP Language for Writing Derivations and Finite Interpretations. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 67–81. Springer, Heidelberg (2006)
37. Sutcliffe, G., Suttner, C.: The State of CASC. AI Communications 19(1), 35–48 (2006)
38. Sutcliffe, G., Suttner, C.B.: The TPTP Problem Library: CNF Release v1.2.1. Journal of Automated Reasoning 21(2), 177–203 (1998)
39. Trac, S., Puzis, Y., Sutcliffe, G.: An Interactive Derivation Viewer. In: Autexier, S., Benzmüller, C. (eds.) Proceedings of the 7th Workshop on User Interfaces for Theorem Provers, 3rd International Joint Conference on Automated Reasoning. Electronic Notes in Theoretical Computer Science, vol. 174, pp. 109–123 (2006)

40. van Benthem Jutting, L.S.: Checking Landau's "Grundlagen" in the AUTOMATH System. PhD thesis, Eindhoven University, Eindhoven, The Netherlands (1979)
41. Van Gelder, A., Sutcliffe, G.: Extending the TPTP Language to Higher-Order Logic with Automated Parser Generation. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 156–161. Springer, Heidelberg (2006)
42. Weidenbach, C., Schmidt, R., Hillenbrand, T., Rusev, R., Topic, D.: SPASS Version 3.0. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 514–520. Springer, Heidelberg (2007)
43. Wiedijk, F.: The Seventeen Provers of the World. LNCS, vol. 3600. Springer, Heidelberg (2006)

System Description: H-PILoT

Carsten Ihlemann and Viorica Sofronie-Stokkermans

Max-Planck-Institut für Informatik, Campus E 1.4, Saarbrücken, Germany

Abstract. This system description provides an overview of H-PILoT (Hierarchical Proving by Instantiation in Local Theory extensions), a program for hierarchical reasoning in extensions of logical theories with functions axiomatized by a set of clauses. H-PILoT reduces deduction problems in the theory extension to deduction problems in the base theory. Specialized provers and standard SMT solvers can be used for testing the satisfiability of the formulae obtained after the reduction. For local theory extensions this hierarchical reduction is sound and complete and – if the formulae obtained this way belong to a fragment decidable in the base theory – H-PILoT provides a decision procedure for testing satisfiability of ground formulae, and can also be used for model generation.

Keywords: local theory extensions, hierarchical reasoning.

1 Introduction

H-PILoT (Hierarchical Proving by Instantiation in Local Theory extensions) is an implementation of the method for hierarchical reasoning in local theory extensions presented in [6,10,12]: it reduces the task of checking the satisfiability of a (ground) formula over the extension of a theory with additional function symbols subject to certain axioms (a set of clauses) to the task of checking the satisfiability of a formula over the base theory. The idea is to replace the set of clauses which axiomatize the properties of the extension functions by a finite set of instances thereof. This reduction is polynomial in the size of the initial set of clauses and is always sound. It is complete in the case of so-called *local extensions* [10]; in this case, it provides a decision procedure for the universal theory of the theory extension if the clauses obtained by the hierarchical reduction belong to a fragment decidable in the base theory. The satisfiability of the reduced set of clauses is then checked with a specialized prover for the base theory.

State of the art SMT provers such as CVC3, Yices and Z3 [1,5,3] are very efficient for testing the satisfiability of *ground formulae* over standard theories, but use heuristics in the presence of *universally quantified* formulae, hence cannot detect *satisfiability* of such formulae. H-PILoT recognizes a class of local axiomatizations, performs the instantiation and hands in a ground problem to the SMT provers or other specialized provers, for which they are know to terminate with a yes/no answer, so it can be used as a tool for steering standard SMT provers, in order to provide decision procedures in the case of local theory extensions.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 131–139, 2009.

H-PILoT can also be used for generating models of satisfiable formulae; and even more, it can be coupled to programs with graphic facilities to provide graphical representations of these models. Being a decision procedure for many theories important in verification, H-PILoT is extremely helpful for deciding truth or satisfiability in a large variety of verification problems.

2 Description of the H-PILoT Implementation

H-PILoT is an implementation of the method for hierarchical reasoning in local theory extensions presented in [6,10,11,12]. H-PILoT is implemented in Ocaml. The system (with manual and examples) can be downloaded from `www.mpi-inf.mpg.de/~ihlemann/software/`. Its general structure is presented in Figure 1.

2.1 Theoretical Background

Let $\mathcal{T}_0$ be a Σ_0-theory. We consider extensions $\mathcal{T}_1 = \mathcal{T}_0 \cup \mathcal{K}$ of $\mathcal{T}_0$ with function symbols in a set Σ_1 (extension functions) whose properties are axiomatized by a set $\mathcal{K}$ of $\Sigma_0 \cup \Sigma_1$-clauses. Let Σ_c be an additional set of constants.

Task. Let G be a set of ground $\Sigma_0 \cup \Sigma_1 \cup \Sigma_c$-clauses. We want to check whether or not G is satisfiable w.r.t. $\mathcal{T}_0 \cup \mathcal{K}$.

Method. Let $\mathcal{K}[G]$ be the set of those instances of $\mathcal{K}$ in which every subterm starting with an extension function is a ground subterm already appearing in $\mathcal{K}$ or G. If G is unsatisfiable w.r.t. $\mathcal{T}_0 \cup \mathcal{K}[G]$ then it is also unsatisfiable w.r.t. $\mathcal{T}_0 \cup \mathcal{K}$. The converse is not necessarily true. We say that the extension $\mathcal{T}_0 \cup \mathcal{K}$ of $\mathcal{T}_0$ is *local* if for each set G of ground clauses, G is unsatisfiable w.r.t. $\mathcal{T}_0 \cup \mathcal{K}$ if and only if $\mathcal{K}[G] \cup G$ has no partial $\Sigma_0 \cup \Sigma_1$-model whose Σ_0-reduct is a total model of $\mathcal{T}_0$ and in which all Σ_1-terms of $\mathcal{K}$ and G are defined.

Theorem 1 ([10]). *Assume that the extension $\mathcal{T}_0 \subseteq \mathcal{T}_1 = \mathcal{T}_0 \cup \mathcal{K}$ is local and let G be a set of ground clauses. Let $\mathcal{K}^0 \cup G^0 \cup D$ be the purified form of $\mathcal{K} \cup G$ obtained by introducing fresh constants for the Σ_1-terms, adding their definitions $d \approx f(t)$ to D, and replacing $f(t)$ in G and $\mathcal{K}[G]$ by d. (Then Σ_1-functions occur only in D in unit clauses of the form $d \approx f(t)$.) The following are equivalent.*

1. *$\mathcal{T}_0 \cup \mathcal{K} \cup G$ has a (total) model.*
2. *$\mathcal{T}_0 \cup \mathcal{K}[G] \cup G$ has a partial model where all subterms of $\mathcal{K}$ and G and all Σ_0-functions are defined.*
3. *$\mathcal{T}_0 \cup \mathcal{K}^0 \cup G^0 \cup \mathsf{Con}^0$ has a total model, where*
   $\mathsf{Con}^0 := \{ \bigwedge_{i=1}^n c_i \approx d_i \rightarrow c \approx d \mid f(c_1, ..., c_n) \approx c, f(d_1, ..., d_n) \approx d \in D \}$.

A variant of this notion, namely Ψ-locality, was also studied, where the set of instances to be taken into account is $\mathcal{K}[\Psi(G)]$, where Ψ is a closure operator which may add a (finite) number of new terms to the subterms of G. We also analyzed a generalized version of locality, in which the clauses in $\mathcal{K}$ and the set G of ground clauses are allowed to contain first-order Σ_0-formulae.

Examples of Local Extensions. Among the theory extensions which we proved to be local or Ψ-local in previous work are:

- theories of pointers with stored scalar information in the nodes [8,7];
- theories of arrays with integer indices, and elements in a given theory [2,7];
- theories of functions (e.g. over an ordered domain, or over a numerical domain) satisfying e.g. monotonicity or boundedness conditions [10,14];
- various combinations of such extensions [12,7].

We can also consider successive extensions of theories: $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}_1 \subseteq \cdots \subseteq \mathcal{T}_0 \cup \mathcal{K}_1 \cup \cdots \cup \mathcal{K}_n$. If every variable in $\mathcal{K}_i$ occurs below a function symbol in Σ_i, this reduction process can be iterated [7].

2.2 Preprocessing

H-PILoT receives as input a many-sorted specification of the signature; a specification of the hierarchy of local extensions to be considered; an axiomatization $\mathcal{K}$ of the theory extension(s); a set G of ground clauses containing possibly additional constants. H-PILoT allows the following pre-processing functionality:

Translation to Clause Form. H-PILoT provides a translator to clause normal form (CNF) for ease of use. First-order formulas can be given as input; H-PILoT translates them into CNF.[1]

Flattening/Linearization. Methods for recognizing local theory extensions usually require that the clauses in the set $\mathcal{K}$ extending the base theory are *flat* and *linear*[2]. If the flags -linearize and/or -flatten are used then the input is flattened and/or linearized. H-PILoT allows the user to enter a more readable non-flattened version and will perform the flattening and linearization of $\mathcal{K}$.

Recognizing Syntactic Criteria which Imply Locality. Examples of local extensions include those mentioned in Section 2.1 (and also iterations and combinations thereof). In the pre-processing phase H-PILoT analyzes the input clauses to check whether they are in one of these fragments.

- If the flag - array is on: checks if the input is in the array property fragment[2];
- if the keyword "pointer" is detected: checks if the input is in the pointer fragment in [8] and possibly adds premises of the form "$t \neq \mathsf{null}$".

If the answer is "yes", we know that the extensions we consider are local, i.e. that H-PILoT can be used as a decision procedure. We are currently extending the procedure to recognize "free", "monotone" and "bounded" functions.

[1] In the present implementation, the CNF translator does not provide the full functionality of FLOTTER [9], but is powerful enough for most applications. (An optimized CNF translator is being implemented.)

[2] Flatness means that extension functions may not be nested; linearity means that variables may occur only in the same extension term (which may appear repeatedly).

2.3 Main Algorithm

The main algorithm hierarchically reduces a decision problem in a theory extension to a decision problem in the base theory.

Given a set of clauses $\mathcal{K}$ and a ground formula G, the algorithm we use carries out a hierarchical reduction of G to a set of formulae in the base theory, cf. Theorem 1. It then hands over the new problem to a dedicated prover such as Yices, CVC3 or Z3. H-PILoT is also coupled with Redlog (for handling non-linear real arithmetic) and with SPASS[3].

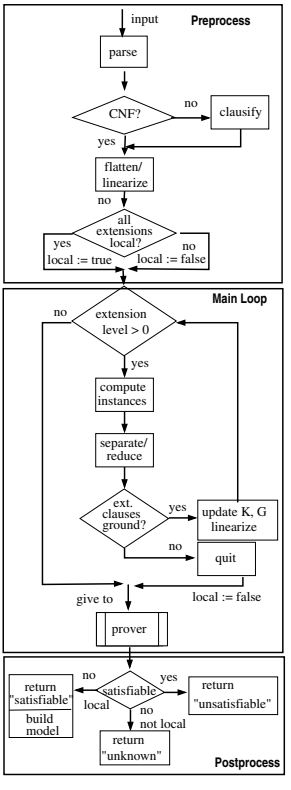


Fig. 1. H-PILoT Structure

Loop. For a chain of local extensions:

$$\mathcal{T}_0 \subseteq \mathcal{T}_1 = \mathcal{T}_0 \cup \mathcal{K}_1 \subseteq \mathcal{T}_2 = \mathcal{T}_0 \cup \mathcal{K}_1 \cup \mathcal{K}_2 \subseteq \ldots \subseteq \mathcal{T}_n = \mathcal{T}_0 \cup \mathcal{K}_1 \cup \ldots \cup \mathcal{K}_n.$$

a satisfiability check w.r.t. the last extension can be reduced (in n steps) to a satisfiability check w.r.t. $\mathcal{T}_0$. The only caveat is that at each step the reduced clauses $\mathcal{K}_i^0 \cup G^0 \cup \mathsf{Con}^0$ need to be ground. Groundness is assured if each variable in a clause appears at least once under an extension function. In that case, we know that at each reduction step the total clause size only grows polynomially in the size of $\Psi(G)$ [10]. H-PILoT allows the user to specify a chain of extensions by tagging the extension functions with their place in the chain (e.g., if f belongs to $\mathcal{K}_3$ but not to $\mathcal{K}_1 \cup \mathcal{K}_2$ it is declared as level 3). Let $i = n$. As long as the extension level $i > 0$, we compute $\mathcal{K}_i[G]$ ($\mathcal{K}_i[\Psi(G)]$ in case of arrays). If no separation of the extension symbols is required, we stop here (the result will be passed to an external prover). Otherwise, we perform the hierarchical reduction in Theorem 1 by purifying $\mathcal{K}_i$ and G (to $\mathcal{K}_i^0, G^0$ resp.) and by adding corresponding instances of the congruence axioms Con_i. To prepare for the next iteration, we transform the clauses into the form $\forall x.\Phi \vee \mathcal{K}_i$ (compute prenex form, skolemize). If $\mathcal{K}_i[G]/\mathcal{K}_i^0$ is not ground, we quit with a corresponding message. Otherwise we set: $G' := \mathcal{K}_i^0 \wedge G^0 \wedge \mathsf{Con}_i, \mathcal{T}_0' := \mathcal{T}_0 \setminus \{\mathcal{K}_{i-1}\}, \mathcal{K}' := \mathcal{K}_{i-1}$. We flatten and linearize $\mathcal{K}'$ and decrease i. If level $i = 0$ is reached, we exit the main loop and G' is handed to an external prover[4].

[3] H-PILoT only calls one of these solvers once.

[4] Completeness is guaranteed if all extensions are known to be local and if each reduction step produces a set of ground clauses for the next step.

2.4 Post-processing

Depending on the answer of the external provers to the satisfiability problem G_n, we can infer whether the initial set G of clauses was satisfiable or not.

- If G_n is unsatisfiable w.r.t. $\mathcal{T}_0$ then we know that G is unsatisfiable.
- If G_n is satisfiable, but H-PILoT failed to detect, and the user did not assert the locality of the sets of clauses used in the axiomatization, its answer is "unknown".
- If G_n is satisfiable and H-PILoT detected the locality of the axiomatization, then the answer is "satisfiable". In this case, H-PILoT takes advantage of the ability of SMT-solvers to provide counter-examples for the satisfiable set G_n of ground clauses and specifies a counter-example of G by translating the basic SMT-model of the reduced query to a model of the original query[5]. The counter-examples can be graphically displayed using Mathematica. This is currently done separately; an integration with Mathematica is planned for the future.

3 System Evaluation

We have used H-PILoT on a variety of local extensions and on chains of local extensions. The flags that we used are described in the manual (see `www.mpi-inf.mpg.de/~ihlemann/software/`). An overview of the tests we made is given in sections 3.1 and 3.2. In analyzing them, we distinguish between satisfiable and unsatisfiable problems.

Unsatisfiable Problems. For simple unsatisfiable problems, there hardly is any difference in run-time whether one uses H-PILoT or an SMT-solver directly.[6] When we consider chains of extensions the picture changes dramatically. On one test example (array insert), checking an invariant of an array insertion algorithm, which used a chain of two local extensions, Yices performed considerably slower than H-PILoT: The original problem took Yices 318.22s to solve. The hierarchical reduction yielded 113 clauses of the background theory (integers), proved unsatisfiable by Yices in 0.07s.

Satisfiable Problems. For satisfiable problems over local theory extensions, H-PILoT always provides the right answer. In local extensions, H-PILoT is a decision procedure whereas completeness of other SMT-solvers is not guaranteed. In the test runs, Yices either ran out of memory or took more than 6 hours when given any of the unreduced problems. This even was the case for small problems, e.g. problems over the reals with less than ten clauses. With H-PILoT as a front end, Yices solved all the satisfiable problems in less than a second.

[5] This improves readability greatly, especially when we have a chain of extensions.

[6] This is due to the fact that a good SMT-solver uses the heuristic of trying out all the occurring ground terms as instantiations of universal quantifiers. For local extensions this is always sufficient to derive a contradiction.

3.1 Test Runs for H-PILoT

We analyzed the following examples[7]:

array insert. Insertion of an element into a sorted integer array. Arrays are definitional extensions here.

array insert (∃). Insertion of an element into a sorted integer array. Arrays are definitional extensions here. Alternate version with (implicit) existential quantifier.

array insert (linear). Linear version of **array insert**.

array insert real. Like **array insert** but with an array of reals.

array insert real (linear). Linear version of **array insert real**.

update process priorities (∀∃). Updating of priorities of processes. This is an example of extended locality: We have a ∀∃-clause.

list1. Made up example of integer lists. Some arithmetic is required

chain1. Simple test for chains of extensions (plus transitivity).

chain2. Simple test for chains of extensions (plus transitivity and arithmetic).

double array insert. Problem of the array property fragment [2]. (A sorted array is updated twice.)

mono. Two monotone functions over integers/reals for SMT solver.

mono for distributive lattices.R. Two monotone functions over a distributive lattice. The axioms for a distributive lattice are stated together with the definition of a relation R: $R(x, y) :\Leftrightarrow x \wedge y = x$. Monotonicity of f (resp. of g) is given in terms of R: $R(x, y) \rightarrow R(f(x), f(y))$. Flag `-freeType` must be used.

mono for distributive lattices. Same as **mono for distributive lattices.R** except that no relation R is defined. Monotonicity of the two functions f, g is directly given: $x \wedge y = x \rightarrow f(x) \wedge f(y) = f(x)$. (Much harder than defining R.) Flag `-freeType` must be used.

mono for poset. Two monotone functions over a poset with poset axioms. Same as **mono**, except the order modeled by a relation R. Flag `-freeType` should be used.

mono for total order. Same as **mono** except linearity is an axiom. This makes no difference unless SPASS is used.

own. Simple test for monotone function.

mvLogic/mv1.sat. Example for MV-algebras. The Łukasiewicz connectives can be defined in terms of the (real) operations $+, -, \leq$. Linearity is deducible from axioms.

mvLogic/mv2. Example for MV-algebras. The Łukasiewicz connectives can be defined in terms of $+, -, \leq$. The BL axiom is deducible.

mvLogic/bl1. Example for MV-algebras with BL axiom (redundantly) included. The Łukasiewicz connectives can be defined in terms of $+, -, \leq$.

mvLogic/example_6.1. Example for MV-algebras with monotone and bounded function. The Łukasiewicz connectives can be defined in terms of $+, -, \leq$.

RBC_simple. Example with train controller.

RBC_variable2. Example with train controller.

[7] The satisfiable variant of a problem carries the suffix ".sat".

3.2 Test Results

Name	**status**	**#cl.**	**H-PILoT + yices**	**H-PILoT + yices** stop at $\mathcal{K}[G]$	**yices**
array insert (implicit ∃)	Unsat	310	0.29	0.06	0.36
array insert (implicit ∃).sat	Sat	196	0.13	0.04	time out
array insert	Unsat	113	0.07	0.03	318.22[1]
array insert (linear version)	Unsat	113	0.07	0.03	7970.53[2]
array insert.sat	Sat	111	0.07	0.03	time out
array insert real	Unsat	113	0.07	0.03	360.00[1]
array insert real (linear)	Unsat	113	0.07	0.03	7930.00[2]
array insert real.sat	Sat	111	0.07	0.03	time out
update process priorities	Unsat	45	0.02	0.02	0.03
update process priorities.sat	Sat	37	0.02	0.02	unkown
list1	Unsat	18	0.02	0.01	0.02
list1.sat	Sat	18	0.02	0.01	unknown
chain1	Unsat	22	0.01	0.01	0.02
chain2	Unsat	46	0.02	0.02	0.02
mono	Unsat	20	0.01	0.01	0.01
mono.sat	Sat	20	0.01	0.01	unknown
mono for distributive lattices.R	Unsat	27	0.22	0.06	0.03
mono for distributive lattices.R.sat	Sat	27	unknown*	unknown*	unknown
mono for distributive lattices	Unsat	17	0.01	0.01	0.02
mono for distributive lattices.sat	Sat	17	0.01	0.01	unknown
mono for poset	Unsat	20	0.02	0.02	0.02
mono for poset.sat	Sat	20	unknown*	unknown*	unknown
mono for total order	Unsat	20	0.02	0.02	0.02
own	Unsat	16	0.01	0.01	0.01
mvLogic/mv1	Unsat	10	0.01	0.01	0.02
mvLogic/mv1.sat	Sat	8	0.01	0.01	unknown
mvLogic/mv2	Unsat	8	0.01	0.01	0.06
mvLogic/bl1	Unsat	22	0.02	0.01	0.03
mvLogic/example_6.1	Unsat	10	0.01	0.01	0.03
mvLogic/example_6.1.sat	Sat	10	0.01	0.01	unknown
RBC_simple	Unsat	42	0.03	0.02	0.03
double array insert	Unsat	606	1.16	0.20	0.07
double array insert	Sat	605	1.10	0.20	unknown
RBC_simple.sat	Sat	40	0.03	0.02	out. mem.
RBC_variable2	Unsat	137	0.08	0.04	0.04
RBC_variable2.sat	Sat	136	0.08	0.04	out. mem.

User + sys times (in s). Run on an Intel Xeon 3 GHz, 512 K-byte cache. Median of 100 runs (entries marked with [1]: 10 runs; marked with [2]: 3 runs). The third column lists the number of clauses produced; "unknown" means Yices answer was "unknown", "out. mem." means out of memory and time out was set at 6h. For an explanation of "unknown*" see below.

(*) The answer "unknown" for the satisfiable example with monotone functions over distributive lattices/posets (H-Pilot followed by Yices) is due to the fact that Yices

cannot handle the universal axioms of distributive lattices/posets. A translation of such problems to SAT provides a decision procedure (cf. [10] and also [15]).

Acknowledgments. This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, `www.avacs.org`).

References

1. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
2. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2006)
3. de Moura, L., Bjorner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
4. Dolzmann, A., Sturm, T.: Redlog: Computer algebra meets computer logic. ACM SIGSAM Bulletin 31(2), 2–9 (1997)
5. Dutertre, B., de Moura, L.: Integrating Simplex with DPLL(T). CSL Technical Report, SRI-CSL-06-01 (2006)
6. Ganzinger, H.: Relating semantic and proof-theoretic concepts for polynomial time decidability of uniform word problems. In: 16th IEEE Symposion on Logic in Computer Science, pp. 81–92. IEEE Press, New York (2001)
7. Ihlemann, C., Jacobs, S., Sofronie-Stokkermans, V.: On local reasoning in verification. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 265–281. Springer, Heidelberg (2008)
8. McPeak, S., Necula, G.C.: Data structure specifications via local equality axioms. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 476–490. Springer, Heidelberg (2005)
9. Nonnengart, A., Weidenbach, C.: Computing Small Clause Normal Forms. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 335–367. Elsevier, Amsterdam (2001)
10. Sofronie-Stokkermans, V.: Hierarchic reasoning in local theory extensions. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 219–234. Springer, Heidelberg (2005)
11. Sofronie-Stokkermans, V.: Interpolation in local theory extensions. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 235–250. Springer, Heidelberg (2006)
12. Sofronie-Stokkermans, V.: Hierarchical and modular reasoning in complex theories: The case of local theory extensions. In: Konev, B., Wolter, F. (eds.) FroCos 2007. LNCS, vol. 4720, pp. 47–71. Springer, Heidelberg (2007)
13. Sofronie-Stokkermans, V.: Efficient Hierarchical Reasoning about Functions over Numerical Domains. In: Dengel, A.R., Berns, K., Breuel, T.M., Bomarius, F., Roth-Berghofer, T.R. (eds.) KI 2008. LNCS (LNAI), vol. 5243, pp. 135–143. Springer, Heidelberg (2008)

14. Sofronie-Stokkermans, V., Ihlemann, C.: Automated reasoning in some local extensions of ordered structures. J. Multiple-Valued Logics and Soft Computing 13(4-6), 397–414 (2007)
15. Sofronie-Stokkermans, V.: Resolution-based decision procedures for the universal theory of some classes of distributive lattices with operators. J. Symb. Comp. 36(6), 891–924 (2003)

SPASS Version 3.5

Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski

Max-Planck-Institut für Informatik, Campus E1 4
D-66123 Saarbrücken
spass@mpi-inf.mpg.de

Abstract. SPASS is an automated theorem prover for full first-order logic with equality and a number of non-classical logics. This system description provides an overview of our recent developments in SPASS 3.5 including subterm contextual rewriting, improved split backtracking, a significantly faster FLOTTER implementation with additional control flags, completely symmetric implementation of forward and backward redundancy criteria, faster parsing with improved support for big files, faster and extended sort module, and support for include commands in input files. Finally, SPASS 3.5 can now parse files in TPTP syntax, comes with a new converter tptp2dfg and is distributed under a BSD style license.

1 Introduction

SPASS is an automated theorem prover for full first-order logic with equality and a number of non-classical logics. This system description provides an overview of our recent developments in SPASS version 3.5 compared to version 3.0 [WSH+07].

One important change in the use of SPASS when moving from SPASS version 3.0 to SPASS version 3.5 is the change from a GPL to a BSD style license. We have been asked by several companies for this change and now eventually did it. Actually, it took some effort as it required the (re)implementation of several SPASS modules as we were so far relying on some code distributed under a GPL license, for example the command line parser up to version 3.0. Starting from SPASS version 3.5 it will be distributed under a "non-virulent" BSD style license.

The most important enhancements based on advances in theory are subterm contextual rewriting (Section 2) and improved split backtracking (Section 3).

Important enhancements based on further (re)implementations are a significantly faster FLOTTER procedure with more control on reduction during CNF translation (Section 4), faster parsing with support for big files, a faster and extended sort module, support for include commands in SPASS input files, support for the TPTP input problem syntax and a new tool tptp2dfg (Section 5).

In the subsequent sections we will explain our enhancements in more detail and also provide experimental data. All experiments were carried out on the TPTP version 3.2.0 [SS98] and performed on Opteron 2.4GHz computers running Linux with a 300 second time limit for each problem.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 140–145, 2009.

2 Subterm Contextual Rewriting

Contextual rewriting is a powerful reduction rule generalizing standard unit equational rewriting to equational rewriting under "conditions". For example, consider the two clauses

$$P(x) \to f(x) \approx x \qquad S(g(a)), a \approx b, P(b) \to R(f(a))$$

where we write clauses in implication form [Wei01]. Now in order to rewrite $R(f(a))$ in the second clause to $R(a)$ using the equation $f(x) \approx x$ of the first clause with matcher $\sigma = \{x \mapsto a\}$, we have to show that $P(x)\sigma$ is entailed by the context of the second clause $S(g(a)), a \approx b, P(b)$, i.e., $\models S(g(a)), a \approx b, P(b) \to P(x)\sigma$. This obviously holds, so we can replace $S(g(a)), a \approx b, P(b) \to R(f(a))$ by $S(g(a)), a \approx b, P(b) \to R(a)$ via a contextual rewriting application of $P(x) \to f(x) \approx x$. This step can not be performed by standard unit equational rewriting. Clauses of the form $\Gamma \to \Delta, s \approx t$ occur, e.g., often in problems from software verification where the execution of a function call represented by the term s may depend on some conditions represented in Γ, Δ. Then contextual rewriting simulates conditional execution. In case a clause set N is saturated, for all clauses of the above form $s \approx t$ is strictly maximal in the clause, s is strictly larger than t and s contains all variables occurring in Γ, Δ, and t, then contextual rewriting can be used to effectively decide validity of any ground clause with respect to the ordering based minimal model of N [GS92]. Translated to the above software verification scenario it means that in such a setting contextual rewriting effectively computes any function from N.

Contextual rewriting [BG94] was first implemented in the SATURATE system [GN94] but never matured. It turned out to be very useful for a bunch of examples, but the rule has to be turned off in general, because often the prover does not return in reasonable time from even a single contextual rewriting application test. Compared to this work, we have instantiated the contextual rewriting rule to subterm contextual rewriting and have implemented it in a much more sophisticated way. Our new rule is robust in the sense that invoking this rule in SPASS on the overall TPTP [SS98] results in an overall gain of solved problems.

The reduction rule *Subterm Contextual Rewriting* [WW08] is defined as follows. As already said, we write clauses in implication notation $\Gamma \to \Delta$ denoting that the conjunction of all atoms in Γ implies the disjunction of all atoms in Δ. As usual $\prec$ is a reduction ordering total on ground terms. Let N be a clause set, $C, D \in N$, σ be a substitution then the reductions

$$\mathcal{R}\frac{D = \Gamma_1 \to \Delta_1, s \approx t \qquad C = \Gamma_2, u[s\sigma] \approx v \to \Delta_2}{\begin{array}{c}\Gamma_1 \to \Delta_1, s \approx t \\ \Gamma_2, u[t\sigma] \approx v \to \Delta_2\end{array}}$$

$$\mathcal{R}\frac{D = \Gamma_1 \to \Delta_1, s \approx t \qquad C = \Gamma_2 \to \Delta_2, u[s\sigma] \approx v}{\begin{array}{c}\Gamma_1 \to \Delta_1, s \approx t \\ \Gamma_2 \to \Delta_2, u[t\sigma] \approx v\end{array}}$$

where the following conditions are satisfied (i) $s\sigma \succ t\sigma$, (ii) $C \succ D\sigma$, (iii) τ maps all variables from $C, D\sigma$ to fresh Skolem constants, (iv) $(\Gamma_2 \rightarrow A)\tau$ is ground subterm redundant in N for all A in $\Gamma_1\sigma$, (v) $(A \rightarrow \Delta_2)\tau$ is ground subterm redundant in N for all A in $\Delta_1\sigma$, are subterm contextual rewriting reductions.

A ground clause $\Gamma \rightarrow \Delta$ is *ground subterm redundant* in N if there are ground instances $C_1\sigma_1, \ldots, C_n\sigma_n$ of clauses from N such that all codomain terms of the σ_i are subterms of $\Gamma \rightarrow \Delta$, the clauses $(C_i\sigma_i) \prec (\Gamma \rightarrow \Delta)$ for all i, and $C_1\sigma_1, \ldots, C_n\sigma_n \models \Gamma \rightarrow \Delta$.

Subterm contextual rewriting is an instance of the general contextual rewriting rule. In particular, the recursive conditions (iv) and (v) are tested with respect to ground clauses of the form $\Gamma \rightarrow \Delta$ and only smaller ground clauses $C_i\sigma_i$. The smaller ground clauses $C_i\sigma_i$, considered for the test, are build by instantiating clauses from N only with ground terms from $\Gamma \rightarrow \Delta$. We actually implemented these conditions by a recursive call to the reduction machinery of SPASS including subterm contextual rewriting where we simply treat the variables in $(\Gamma_2 \rightarrow A)$, $(A \rightarrow \Delta_2)$ as fresh Skolem constants. This has the important advantage that the side condition clauses don't need to be explicitly instantiated (by τ). On the other hand we needed to extend our ordering computation algorithms to deal with particular variables as constants. Further details can be found in [WW08].

Even under these restrictions, our first implementation of the subterm contextual rewriting rule lost more examples on the overall TPTP than it won. A careful inspection of several runs showed that in particular a lot of time is wasted by testing the same failing terms for contextual rewriting again and again. Therefore, we decided to introduce *fault caching* as a form of (negative) dynamic programming. Whenever subterm contextual rewriting is tested for applicability on some term $s\sigma$ and the test fails, we store $s\sigma$ in an index. Then all subsequent tests first look into the index and only if the top term to be reduced is not contained, the conditions for the rule are evaluated. Of course, this is an approximation of the rule as some potential applications are lost. However, it turned out that at least with respect to the TPTP, SPASS with subterm contextual rewriting with fault caching solves more examples than SPASS without.

Subterm contextual rewriting can be controlled by the forward and backward rewriting flags `-RFRew` and `-RBRew` where starting from a value of `3` the rule is activated. Further details can be found in the SPASS man page.

Comparing SPASS version 3.0 with SPASS version 3.5 with activated subterm contextual rewriting (set flags `-RFRew=4 -RBRew=3 -RTaut=2`) yields an overall gain of 31 problems on the TPTP (actually 108 losses and 139 wins). The losses are partly due to the additional time needed for subterm contextual rewriting, partly due to a differently spanned search space. Eventually SPASS with subterm contextual rewriting solved 6 "open" TPTP problems (rating 1.0): SWC261+1, SWC308+1, SWC329+1, SWC335+1, SWC342+1, and SWC345+1. For this experiment we deactivated the advanced split backtracking introduced in the next section and used the split backtracking of SPASS version 3.0.

3 Improved Split Backtracking

The splitting rule implementation of SPASS until version 3.0 already contains a form of intelligent backtracking via branch condensing. For each clause we store in a bitfield the splits it depends on. If an empty clause is derived then all splits that did not contribute to the empty clause, easily checked from the empty clause's split bitfield, are removed by branch condensing.

We recently refined the splitting calculus [FW08] by storing with each split the bitfield of an empty clause after backtracking one of its branches. If now the second branch is also refuted then this information can be propagated upwards in the split tree by combining the bitfields of sibling branches or subtrees. Any splits that neither contributed to left nor to the right empty clause can be undone. It turns out that this refinement saves a remarkable number of splits on the TPTP problems.

On the average on all TPTP problems where SPASS performs splitting and which are solved both by version 3.0 and version 3.5, version 3.5 does 783 splits whereas version 3.0 performs 916 splits per problem. This means a saving of 14%. Furthermore, due to splitting SPASS 3.5 solves 28 more problems from the TPTP. Actually it loses 21 problems and wins 49 problems. For these experiments subterm contextual rewriting was not activated so that only the effect of the new split backtracking implementation shows up.

4 Improvements to FLOTTER

FLOTTER is the powerful CNF transformation procedure of SPASS. In particular, it contains sophisticated transformation rules such as optimized Skolemization [NW01] which actually require the computation of proofs during CNF transformation. For larger problems these techniques may actually consume so much time that FLOTTER does not terminate in an acceptable amount of time.

Therefore, we both improved the implementation of crucial FLOTTER parts with respect to large problems and added further flags that can be used to restrict sophisticated reductions during CNF transformation. The new flags `-CNFSub` and `-CNFCon` control the usage of subsumption and condensing during CNF transformation, respectively. The new flag `-CNFRedTimeLimit` can be used to set an overall time limit on all reductions performed in FLOTTER during CNF translation.

5 Further Enhancements

Faster Parsing: Until SPASS version 3.0 the overall input machinery was developed for "small" input files. Due to our own and our customers interest in "large" problems, e.g., expressing real-world finite domain theories, we have reimplemented the overall SPASS parsing technology. We now can parse a 60 MB file in less than 10 seconds and build the CNF for a 1 MB input file like SEU410+2 from TPTP version 3.5.0 with *full* FLOTTER CNF translation and reduction in about 30 seconds.

TPTP Input Syntax Support. Starting from SPASS version 3.5 we support TPTP input files via the new flag `-TPTP`. As TPTP input files may contain include commands, they are resolved by looking in the local directory or the value of the `TPTP` environment variable.

Include Commands. SPASS input files may now also contain include directives. They are resolved at parse time and include files are looked up in the local directory as well as in the directory bound to the `SPASSINPUTS` environment variable.

tptp2dfg. The new tool tptp2dfg translates input files from TPTP into SPASS syntax. Includes can either be expanded or translated into SPASS include directives controlled by the flag `-include`.

Sort Module. In SPASS sorts are used for soft typing and sort simplification reductions [Wei96, GMW97, Wei01]. For example, a clause $S(f(a)), \Gamma \rightarrow \Delta$ is reduced to $\Gamma \rightarrow \Delta$ in the presence of the clauses $\rightarrow S(a)$ and $S(x) \rightarrow S(f(x))$. We reimplemented the module such that it is now about 10-times faster and extended its scope. For example, the new module reduces a clause $S(x), T(x), \Gamma \rightarrow \Delta$ where x does not occur in Γ, Δ to $\Gamma \rightarrow \Delta$ in the presence of the three clauses $\rightarrow S(a)$, $S(x) \rightarrow S(f(x))$, and $\rightarrow T(f(a))$.

Symmetric Reduction: Until SPASS version 3.0 some of the more sophisticated rewrite reductions were only implemented in the forward direction [Wei01]. We now added also the backward direction for all reduction rules.

6 Future Work

There are five major directions for future improvements. Firstly, we will continue integrating results on the superposition theory into SPASS. We are currently working on integrating support for transitive relations and refinements thereof [BG94] and will shortly start implementing particular techniques for finite domain problems [HW07]. Secondly, we will continue our work on hierarchic combinations, in particular with the theory of linear arithmetic. Thirdly, we are planning a reimplementation of a reasonable portion of the SPASS basic modules in order to save memory consumption at run time, gaining further speed and simplifying structures that have been grown during the meanwhile 15 years of SPASS development. Fourthly, we are working on the theory and implementation of a specific SPASS version for reasoning in large ontologies. Fifthly, we will integrate support for model generation and minimal model reasoning into SPASS [HW09].

SPASS version 3.5 is available from its homepage

`http://www.spass-prover.org/`

Acknowledgments. We thank Geoff Sutcliffe for his persistent support for implementing parsing of the TPTP input syntax and our reviewers for their valuable comments.

References

[BG94] Bachmair, L., Ganzinger, H.: Rewrite-based equational theorem proving with selection and simplification. Journal of Logic and Computation 4(3), 217–247 (1994); Revised version of Max-Planck-Institut für Informatik technical report, MPI-I-91-208 (1991)

[FW08] Fietzke, A., Weidenbach, C.: Labelled splitting. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS, vol. 5195, pp. 459–474. Springer, Heidelberg (2008)

[GMW97] Ganzinger, H., Meyer, C., Weidenbach, C.: Soft typing for ordered resolution. In: McCune, W. (ed.) CADE 1997. LNCS (LNAI), vol. 1249, pp. 321–335. Springer, Heidelberg (1997)

[GN94] Ganzinger, H., Nieuwenhuis, R.: The saturate system 1994 (1994), `http://www.mpi-sb.mpg.de/SATURATE/Saturate.html`

[GS92] Ganzinger, H., Stuber, J.: Inductive theorem proving by consistency for first-order clauses. In: Rusinowitch, M., Remy, J.-L. (eds.) CTRS 1992. LNCS, vol. 656, pp. 226–241. Springer, Heidelberg (1992)

[HW07] Hillenbrand, T., Weidenbach, C.: Superposition for finite domains. Research Report MPI-I-2007-RG1-002, Max-Planck Institute for Informatics, Saarbruecken, Germany (April 2007)

[HW09] Horbach, M., Weidenbach, C.: Decidability results for saturation-based model building. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 404–420. Springer, Heidelberg (2009)

[NW01] Nonnengart, A., Weidenbach, C.: Computing small clause normal forms. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 6, vol. 1, pp. 335–367. Elsevier, Amsterdam (2001)

[SS98] Sutcliffe, G., Suttner, C.B.: The TPTP problem library – cnf release v1.2.1. Journal of Automated Reasoning 21(2), 177–203 (1998)

[Wei96] Weidenbach, C.: Unification in sort theories and its applications. Annals of Mathematics and Artificial Intelligence 18(2-4), 261–293 (1996)

[Wei01] Weidenbach, C.: Combining superposition, sorts and splitting. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 27, vol. 2, pp. 1965–2012. Elsevier, Amsterdam (2001)

[WSH+07] Weidenbach, C., Schmidt, R., Hillenbrand, T., Rusev, R., Topic, D.: Spass version 3.0. In: Pfenning, F. (ed.) CADE 2007. LNCS, vol. 4603, pp. 514–520. Springer, Heidelberg (2007)

[WW08] Weidenbach, C., Wischnewski, P.: Contextual rewriting in Spass. In: PAAR/ESHOL, Sydney, Australien. CEUR Workshop Proceedings, vol. 373, pp. 115–124 (2008)

DEI: A Theorem Prover for Terms with Integer Exponents

Hicham Bensaid[1], Ricardo Caferra[2], and Nicolas Peltier[2]

[1] INPT/LIG
Avenue Allal Al Fassi - Madinat Al Irfane - Rabat - Morocco
bensaid@inpt.ac.ma
[2] LIG, Grenoble INP/CNRS
Bâtiment IMAG C - 220, rue de la Chimie - 38400 Saint Martin d'Hères - France
Ricardo.Caferra@imag.fr, Nicolas.Peltier@imag.fr

Abstract. An extension of the superposition-based E-prover [8] is described. The extension allows terms with integer exponents [3] in the input language. Obviously, this possibility increases the capabilities of the E-prover particularly for preventing non-termination.

1 Introduction

Term schematisations allow one to denote infinite sequences of iterated terms, which frequently occur in many symbolic computation procedures (in particular in proof procedures). The number of iterations is part of the term syntax and may be a variable. In some cases, the capability to denote such sequences avoids non termination [6]. For instance the clause set $\{even(0), \forall x.even(x) \Rightarrow even(s(s(x)))\}$ can be replaced by the unit clause $\forall n.even(s^{2n}(0))$.

There exists a hierarchy of term schematisation languages, with different expressive powers. They mainly differ from each other by the class of inductive contexts that can be handled. The original formalism [2] allows only ground contexts with no nested iterations. [3] extends the language to any inductive context, provided that the inductive path is unique. [7] showed how to get rid of this last condition, allowing for instance sequences of the form $a, f(a,a), f(f(a,a), f(a,a)), \ldots$. Finally, the most powerful language of *primal grammars* [4] handles contexts depending on the iteration rank (as in the sequence $[], [0], [s(0), 0], [s(s(0)), s(0), 0], \ldots$). Unification is decidable for all these languages, thus they can be included in most symbolic computation procedures, in particular in first-order theorem provers. This significantly extends the expressive power of the input language.

In this paper we describe the first (to the best of our knowledge) system to perform inferences on clauses containing term schematisations. As the new prover is an extension of the *E*-prover [8], we have called it DEI (for **D**eduction with the **E**-prover and **I**-terms). The E-prover has been chosen as a starting point because it is well-known, widely distributed and efficient. Our system uses the language of *terms with integer exponents* [3] also called *I*-terms. This formalism is a good compromise between expressive power and simplicity: the ρ-terms [2] are easier

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 146–150, 2009.

to handle but lack expressivity and the primal grammars are very expressive but harder to use in practice (one has to define rewrite systems "outside" the iterated terms, moreover the unification algorithm is rather complex).

We hope that this work, by allowing practical experimentations, will promote the use of term schematisation languages in automated deduction and will allow further investigations (potential applications, better understanding of the formalisms etc.).

2 Terms with Integer Exponents (I-Terms)

We briefly recall the definition of terms with integer exponents (see [3] for details). We assume that three sets of symbols are given: function symbols Σ, standard variables $\mathcal{V}$ and arithmetic variables $\mathcal{V}_N$. Let $\diamond$ be a special symbol, called the "hole". This symbol denotes the changing part of a context.

The set of *terms with one hole* $T_\diamond$ and the set of *terms with integer exponents* T_I (or I*-terms*) are the smallest sets satisfying the following conditions: (i) $\diamond \in T_\diamond$ and $\mathcal{V} \subseteq T_I$. (ii) If $t_1, \ldots, t_n \in T_I$, $t'_i \in T_\diamond$ and $f \in \Sigma$ then $f(t_1, \ldots, t_n) \in T_I$ and $f(t_1, \ldots, t_{i-1}, t'_i, t_{i+1}, \ldots, t_n) \in T_\diamond$. (iii) If $t \in T_\diamond, t \neq \diamond, s \in T_I$ and $n \in \mathcal{V}_N \cup \mathbb{N}$ then $t^n.s \in T_I$. An I-term of the last form is called an N*-term*. I-terms can be naturally incorporated as arguments of literals. A clause built on I-terms is called an I*-clause*.

The semantics of I-terms are specified by the following rewrite system: $\{t^0.s \to s, t^{n+1}.s \to t\{\diamond \leftarrow t^n.s\}\}$. These rules obviously rewrite every *ground* I-term t to a (unique) standard term, denoted by $t\downarrow$. The value of t in an interpretation I is the same as the value of $t\downarrow$. This allows one to assign a value to every ground term, hence to evaluate every ground clause. The semantics is extended to the non ground case by interpreting a clause set as the set of its ground instances (variables in $\mathcal{V}_N$ are mapped to natural numbers).

3 The DEI System

The DEI system is freely available from the webpage: `http://capp.imag.fr/dei.html`. It is based on the $0.999 - 004$ *"Longview2"* release of E-Prover. The extension as for E-Prover is written in ANSI C, using the gcc compiler (the 4.2.4 version) and was successfully tested on a GNU Linux x86 platform (Ubuntu 8.04 Hardy Heron).

In order to preserve modularity and allow further extensions, we have tried to restrict as much as possible the modifications in the original code. Additional data structures have been designed to represent linear Diophantine expressions and the algorithm of [3] for unifying I-terms has been implemented. The Polylib library [5] is used for solving arithmetic constraints. About 4000 lines of code have been added to the E-prover (excluding the Polylib library). Modifications have been necessary in various parts of the code: term sharing handling, term definitions, input/output algorithms and inference machine. As unification of I-terms gives in general *several* maximal unifiers, many changes in the code were needed to take into account this important new feature.

3.1 Input Syntax

The syntax has been kept almost identical to the one of the E-prover. All options are supported. An N-term $t^n.s$ is denoted by `$iterm(t,s)^{n}`, where `$iterm` is a reserved keyword. `@` denotes the hole $\diamond$. Exponents can be any linear diophantine expression, which makes the language more flexible than using variables only (as it is done in [3] to simplify theoretical work). For instance the I-term $even(\diamond)^{2n}(0)$ of page 1 is encoded by `$iterm(even(@),0)^{ n+n }`. This flexibility also has the advantage that all the arithmetic constraints can be expressed in the terms themselves, which avoids having to use constrained clauses (for instance the clause $p(f^n(\diamond).0)$ with constraint $\exists m.n = m + m$ can be denoted by $p(f^{2\times m}(\diamond).0)$. Arithmetic expressions must only occur in an exponent (e.g. the term $g(n, f(\diamond)^n.a)$ is not allowed).

The arithmetic variables are interpreted by non-zero natural numbers. This assumption slightly simplifies some steps of the unification algorithm. For instance the occur check rule $(x = t) \rightarrow \mathit{false}$ can be applied as usual, whereas it would not be correct if t contains exponents that can be interpreted as 0, for instance $f(x, \diamond)^n.a = x$ has a solution, namely $\{n \mapsto 0, x \mapsto a\}$.

3.2 Inferences

All the inference rules of the E-prover (i.e. the **SP** calculus) are supported. Obviously, correctness is preserved. The calculus is complete for non equational I-clause sets (it is easy to see that one can "lift" the ground derivations to I-clauses). However, it is no more complete in general for *equational* I-clauses (this does not depend on the strategy). A simple counterexample proving this is the following: $\{p(f(a, \diamond)^n.b, \neg p(c), f(a, b) \approx d, f(a, d) \approx c\}$ is clearly unsatisfiable (it suffices to replace n by 2 in the first clause) but the superposition calculus generates only the clause $p(d)$. This is due to the fact that superposition can be applied at an arbitrary deep position in the standard terms denoted by an N-term – in particular at positions not occurring in the N-term. Allowing superposition at arbitrary deep positions in an N-term is possible, but this would make the inference rule infinitary (and inefficient). This has been avoided in Dei, for practical reasons. The calculus is still complete for some subclasses, e.g. if the superposition rule cannot be applied along iterated paths (of course, the formal definition of these classes is outside the scope of the present system description).

The usual term orderings (KBO or LPO) have been adapted in order to be applied on I-terms (it is obvious that they are not compatible with unfolding[1]). The simplification and redundancy rules are supported, but using an incomplete matching algorithm (thus Dei miss some simplifications). This is due to the fact that the matching is harder to perform on I-terms than in standard terms. In particular, the indexing techniques used by the E-prover (e.g. the perfect discrimination trees) cannot be directly applied.

[1] Of course the orderings can be defined as usual at the ground level i.e. on the standard terms obtained by instantiating arithmetic variables, but this does not help because there exist an infinite number of ground instances.

The term sharing is maintained except in the exponent part of the term (only the variable(s) are shared). For example if we consider the terms `$iterm(s(@),0)^{N1}`, `$iterm(s(@),0)^{N2}`, `$iterm(s(@),0)^{N1}` and `$iterm(s(@),0)^{2.N1}` the splay tree will be the following one:

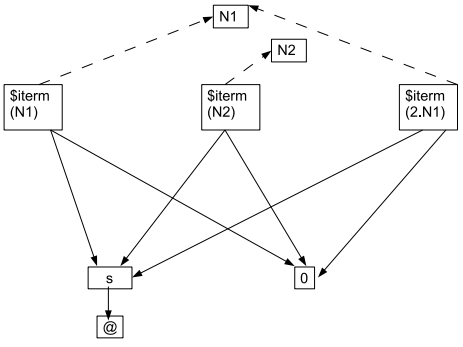


Fig. 1. The splay tree

4 A Short Example

We show DEI at work on the following set of clauses:

$$\{p(f(x_1, g(x_2, \diamond))^n, s(\diamond)^n) \vee$$
$$\neg p'(x_3), p'(a), q(g(x_1, f(x_2, \diamond))^n . g(x_3, x_2), s(\diamond)^m(0)) \vee$$
$$\neg q'(x_3), q'(b), r(g(y, x), z) \vee \neg p(x, z), \neg q(x, y) \vee \neg r(x, y)\}.$$

This little example has been constructed to illustrate the possibilities of the language (contexts containing variables, shared arithmetic variables etc.). The reader can check that it is unsatisfiable. DEI constructs the following refutation (due to space restriction we use the lowest level of verbosity):

```
#p1(a) <- .
#
#q1(b) <- .
#
#r(g(X1,X2),X3) <- p(X2,X3).
#
# <- r(X1,X2), q(X1,X2).
#
#p($iterm(f(X1,g(X2,@)),X3)^{0+1.X4},$iterm(s(@),0)^{0+1.X4}) <- p1(X3).
#
#p($iterm(f(X2,g(X3,@)),a)^{0+1.X1},$iterm(s(@),0)^{0+1.X1}) <- .
#
#r(g(X5,$iterm(f(X2,g(X3,@)),a)^{0+1.X1}),$iterm(s(@),0)^{0+1.X1}) <- .
#
#q($iterm(g(X1,f(X2,@)),g(X3,X2))^{0+1.X4},$iterm(s(@),0)^{0+1.X4})
<- q1(X3).
#
```

```
#q($iterm(g(X2,f(X3,@)),g(b,X3))^{0+1.X1},$iterm(s(@),0)^{0+1.X1}) <- .
#
#r(g(X6,$iterm(f(X2,g(X3,@)),X4)^{0+1.X1}),$iterm(s(@),0)^{0+1.X1})
<- p1(X4).
#
# <- r($iterm(g(X2,f(X3,@)),g(b,X3))^{0+1.X1},$iterm(s(@),0)^{0+1.X1}).
```

Of course, I-terms can be "encoded" into first-order logic (by adding the semantic axioms in Section 2 in the clause set). However, a system as Dei with built-in I-terms handling allows one to encompass some deductive steps in the unification algorithm, which reduces the length of the proofs, improves the readability and the termination behavior. I-terms are especially useful for satisfiability detection.

5 Future Work

Future work includes the extension of Dei to more expressive term schematisation languages (such as the primal grammars [4] or the terms with several holes [7]) and the adaptation to these languages of the reasoning techniques that are commonly used by successful deduction systems (in particular the indexing techniques). We are presently working on an extension of the discrimination trees handling I-terms. Designing a superposition calculus that is complete on I-clauses is a problem that also deserves to be investigated.

In order to fully benefit of the expressive power of I-terms, we also plan to implement additional inference rules to generate automatically I-terms from standard clauses (as in [7,6]). The use of inductive reasoning techniques (in connection with the system presented in [1]) will also be investigated.

References

1. Bensaid, H., Caferra, R., Peltier, N.: Towards systematic analysis of theorem provers search spaces: First steps. In: Leivant, D., de Queiroz, R. (eds.) WoLLIC 2007. LNCS, vol. 4576, pp. 38–52. Springer, Heidelberg (2007)
2. Chen, H., Hsiang, J., Kong, H.: On finite representations of infinite sequences of terms. In: Okada, M., Kaplan, S. (eds.) CTRS 1990. LNCS, vol. 516, pp. 100–114. Springer, Heidelberg (1991)
3. Comon, H.: On unification of terms with integer exponents. Mathematical System Theory 28, 67–88 (1995)
4. Hermann, M., Galbavý, R.: Unification of Infinite Sets of Terms schematized by Primal Grammars. Theoretical Computer Science 176(1-2), 111–158 (1997)
5. Loechner, V.: Polylib: A library for manipulating parametrized polyhedra. Tech. rep., ICPS, Universite Louis Pasteur de Strasbourg (1999)
6. Peltier, N.: A General Method for Using Terms Schematizations in Automated Deduction. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS, vol. 2083, pp. 578–593. Springer, Heidelberg (2001)
7. Salzer, G.: The unification of infinite sets of terms and its applications. In: Voronkov, A. (ed.) LPAR 1992. LNCS (LNAI), vol. 624, pp. 409–429. Springer, Heidelberg (1992)
8. Schulz, S.: System Description: E 0.81. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 223–228. Springer, Heidelberg (2004)

veriT: An Open, Trustable and Efficient SMT-Solver

Thomas Bouton[2], Diego Caminha B. de Oliveira[2],
David Déharbe[1], and Pascal Fontaine[2]

[1] Universidade Federal do Rio Grande do Norte, Natal, RN, Brazil
david@dimap.ufrn.br
[2] LORIA–INRIA, Nancy, France
{Thomas.Bouton,Diego.Caminha,Pascal.Fontaine}@loria.fr

Abstract. This article describes the first public version of the satisfiability modulo theory (SMT) solver veriT. It is open-source, proof-producing, and complete for quantifier-free formulas with uninterpreted functions and difference logic on real numbers and integers.

1 Introduction

We present the satisfiability modulo theory (SMT) solver veriT, a joint work of University of Nancy, INRIA (Nancy, France) and Federal University of Rio Grande do Norte (Natal, Brazil). veriT provides an open, trustable and reasonably efficient decision procedure for the logic of unquantified formulas over uninterpreted symbols, difference logic over integer and real numbers, and the combination thereof. This corresponds to the logics identified as QF_IDL, QF_RDL, QF_UF and QF_UFIDL in the SMT-LIB benchmarks [15,3]. veriT also includes quantifier reasoning capabilities through the integration of a first-order prover and quantifier instantiation heuristics. Finally, veriT has proof-production capabilities; it outputs proofs that may be used or checked by external tools.

veriT is incremental, i.e. after each satisfiability check, new formulas can be added conjunctively to the already checked set of formulas. The input format is the SMT-LIB language [15], but veriT can also be used as a library with an API following the guidelines of [12]. The tool is open-source and distributed under the BSD licence at `http://www.verit-solver.org`. Internally, the solver is organized to be easily extended by plugging new decision procedures in a Nelson-Oppen like combination schema. Although not (yet) as fast as the solvers performing best in the SMT competition [3], veriT has a decent efficiency. We thus claim that it can already be useful in verification platforms where an open-source license, extensibility, and proof certification are important.

Selected features of the veriT solver and an experimental evaluation of its efficiency are presented in Section 2 and 3, respectively. Future developments are described in Section 4.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 151–156, 2009.

2 System Description

The reasoning core of veriT uses a SAT solver [9] to produce models of the Boolean abstraction of the input formula. Such propositional assignments are given to a so-called *theory reasoner*, responsible for verifying if they are models in the background theory. This theory reasoner is a fully incremental combination of decision procedures *à la* Nelson and Oppen, where non-convexity of theories is handled using the model-equality propagation technique [7] which integrates model-based guessing [5] in a classical Nelson-Oppen equality exchange. Equality propagation is controlled by the congruence closure algorithm.

The remainder of this section describes some special features of veriT: integration of a third-party first-order prover, extension of the input language with macro definitions, and production of proofs certifying the produced results.

2.1 Integrating a First-Order Prover

As a particular feature inherited from its predecessor haRVey [8] and, to complement very simple instantiation heuristics, the veriT solver includes a first-order logic (FOL) superposition prover. However, veriT greatly improves the integration of the FOL prover with the other decision procedures, notably with congruence closure. Indeed, the first-order prover is seen within the combination *à la* Nelson-Oppen as a "decision procedure" that takes an arbitrary FOL theory as a parameter. However, due to the cost of running the FOL prover and to its non-incremental nature (when used as a black box), this procedure is called in last resort. A FOL theory is computed from the quantified sub-formulas in the assignment, abstracting ground sub-terms in order to minimize the number of relevant symbols in the theory. In addition, information from congruence closure is used to abstract all subterms in the assignment that do not contain such relevant symbols.

The prover may deduce that the given set of formulas is unsatisfiable. In that case, the deduction tree is parsed to obtain the relevant unsatisfiable subset of the input. A conflict clause is then built using this set and, again, information from the congruence closure data structures. Since the prover is given an upper limit of resources, it always terminates. If the prover terminates without proving the unsatisfaibility of the given set of formulas, ground equalities and deduced ground clauses are identified and propagated back to veriT.

In many cases where the superposition calculus is a decision procedure [2] for the theory represented by the quantified formulas, our technique simulates a Nelson-Oppen combination with on-the-fly purification. It has been shown that first-order generic provers may perform quite well even compared to dedicated decision procedures (see for instance [1]). Currently, the E-prover [16] is used as the first-order prover, and we plan to include Spass [18], which provides better sort handling. Fine-tuning the interplay between the instantiation heuristics and the e-prover is essential for efficiency and remains to be done.

2.2 Macros

The input format for veriT is the SMT-LIB language extended with macro definitions. This syntactic sugar is particularly useful for instance to write formulas containing simple sets constructions (see Figure 1). After β-reduction, and after rewriting equalities between predicates and functions (for instance, if p and q are unary predicates, $p = q$ is rewritten as $\forall x \,.\, p(x) \equiv q(x)$), the obtained formula is first-order. Such formulas may contain quantifiers but, if no function is used, they belong to the Bernays-Schönfinkel-Ramsey fragment (the Bernays-Schönfinkel class with equality) which is decidable. veriT can use the embedded FOL prover as a decision procedure for this fragment. In many more intricate cases [10], the resulting formula still belongs to a decidable fragment, but the decision procedure may become very expensive. To be practical, such cases require heuristics that have not yet been implemented in veriT.

This macro feature is indeed used in tools (e.g. CRefine [14]) that generate verification conditions for formal developments in set-based modelling languages, such as Circus [4], and that integrate veriT as a verification engine to discharge these proof obligations.

```
(benchmark SET008_3p
  :logic UNKNOWN
  :extrasorts (ELMT)
  :extrapreds ((B ELMT) (C ELMT))
  :extramacros
   ((emptyset (lambda (?v ELMT) . false))
    (intersection (lambda (?p (ELMT boolean)) (?q (ELMT boolean)) .
                     (lambda (?x ELMT) . (and (?p ?x) (?q ?x)))))
    (difference (lambda (?p (ELMT boolean)) (?q (ELMT boolean)) .
                   (lambda (?x ELMT) . (and (?p ?x) (not (?q ?x)))))))
  :formula (not (= (intersection (difference B C) C) emptyset)))
```

Fig. 1. A simple example with the macro capability

2.3 Proofs

Proof production has two goals. First, this feature increases the confidence in the tool, the proofs being checked by an independent module inside veriT. Second, skeptical proof assistants can use such traces to reconstruct proofs of formulas discharged by veriT (see [11]).

In Figure 2, we give an example of a very simple formula, and the proof output by veriT. Each line states a fact that can be assumed to hold. It is identified by a number, followed by a list starting with a label identifying the rule used to deduce the fact, followed by a clause, and optionally ended by numerical parameters. In our context, a clause is a disjunctive list of formulas (not literals), maybe containing a sole formula. The numerical parameters depend on the rule, and may be either identifiers of previous clauses (e.g. in the resolution rule), or other

```
(benchmark example
  :logic QF_UF
  :extrafuns ((a U) (b U) (c U) (f U U))
  :extrapreds ((p U))
  :formula (and (= a c) (= b c)
                (or (not (= (f a) (f b)))
                    (and (p a) (not (p b))))))

1:(input ((and (= a c) (= b c)
               (or (not (= (f a) (f b))) (and (p a) (not (p b)))))))
2:(and ((= a c)) 1 0)
3:(and ((= b c)) 1 1)
4:(and ((or (not (= (f a) (f b))) (and (p a) (not (p b))))) 1 2)
5:(and_pos ((not (and (p a) (not (p b)))) (p a)) 0)
6:(and_pos ((not (and (p a) (not (p b)))) (not (p b))) 1)
7:(or ((not (= (f a) (f b))) (and (p a) (not (p b)))) 4)
8:(eq_congruent ((not (= b a)) (= (f a) (f b))))
9:(eq_transitive ((not (= b c)) (not (= a c)) (= b a)))
10:(resolution ((= (f a) (f b)) (not (= b c)) (not (= a c))) 8 9)
11:(resolution ((= (f a) (f b))) 10 2 3)
12:(resolution ((and (p a) (not (p b)))) 7 11)
13:(resolution ((p a)) 5 12)
14:(resolution ((not (p b))) 6 12)
15:(eq_congruent_pred ((not (= b a)) (p b) (not (p a))))
16:(eq_transitive ((not (= b c)) (not (= a c)) (= b a)))
17:(resolution ((p b) (not (p a)) (not (= b c)) (not (= a c))) 15 16)
18:(resolution () 17 2 3 13 14)
```

Fig. 2. A simple example with its proof

place information. As an example, the *and* rule (for instance the second line in Figure 2: `(and ((= a c)) 1 0)`) takes two numerical parameters. The first numerical parameter refers to the clause C in a previous numbered rule (i.e. `1` refers to the clause in the `input` rule, at line 1). This clause C is unit and is hence represented as a list of one formula (the whole input formula in our example), and this formula is a conjunction $a_0 \wedge \cdots \wedge a_n$. Obviously, each subformula $a_0, \ldots a_n$ is a consequence of C, and the second parameter just gives the identifier of the formula in the new clause, i.e. the second numerical parameter in rule at line 2 indicates the formula at position 0 in the input.

veriT already provides proof production for formulas with arbitrary Boolean structure and uninterpreted functions, and is being extended to linear arithmetics. The first line is the input. Every other fact is either a consequence of previous ones, or is a tautology. The input formula being unsatisfiable, the last deduced fact is the empty clause. In the example, lines 2 to 7 account for the conjunctive normal form transformation. Lines 8, 9, 15, and 16 are tautologies related to the theory of equality. The remaining facts are deduced by resolution from the other ones.

Since every proof-related information is handled through a unique module inside veriT, any proof format for SMT (for instance [17,6]) can be adopted as soon as it becomes a standard. Although our previous experiments [11] showed that the proof size was not the bottleneck for the cooperation with skeptical proof assistants, the implementation of techniques to greatly reduce the size of our proof traces is planned.

3 Experimental Evaluation

We evaluated veriT, CVC3 and Z3 (both using the latest available version in February 2009) against the SMT-LIB benchmarks for QF_IDL, QF_RDL, QF_UF and QF_UFIDL (June 2008 version) using an Intel(R) Pentium(R) 4 CPU at 3.00 GHz with 1 GiB of RAM and a timeout of 120 seconds. The following table presents, for each solver, the number of completed benchmarks.

Solver	QF_UF (6656)	QF_UFIDL (432)	QF_IDL (1673)	QF_RDL (204)	all (8965)
veriT	6323	332	918	100	7673
CVC3	6378	278	802	45	7503
Z3	6608	419	1511	158	8696

This clearly shows that, although veriT is not yet as efficient as competition winning tools [3], its efficiency is decent. The proof production capability of veriT does not come at a cost on efficiency.

4 Future Work

veriT is a new SMT-solver that provides an open framework to generate certifiable proofs without sacrificing too much efficiency. The future developments will notably include features related to efficiency and expressiveness. Considering efficiency aspects, the tool does not yet implement theory propagation [13], a technique that is known to greatly improve the efficiency of SMT solvers. Concerning expressiveness, the linear arithmetic decision procedure currently only handles difference logic; we are now developing a reasoning engine for linear arithmetic based on the Simplex method, which will extend completeness to full linear arithmetic. Quantifier reasoning will be improved by including new instantiation heuristics, as well as adding support for patterns guiding quantifier instantiations. We also plan to integrate the proof production capabilities of the embedded FOL prover with that of veriT.

Finally, we are working on the application of veriT to formal development efforts, mainly of concurrent systems. In that context, the ability to handle sets is very helpful; a major objective is to improve the support for such constructions.

Acknowledgments. We would like to thank Stephan Merz for his comments and guidance. The ancestor of the veriT solver, haRVey, was initiated by the third author and Silvio Ranise, to whom we are indebted of several ideas used in veriT. We are also grateful to the anonymous reviewers for their remarks.

References

1. Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: New results on rewrite-based satisfiability procedures. ACM Trans. Comput. Log. 10(1) (2009)
2. Armando, A., Ranise, S., Rusinowitch, M.: A rewriting approach to satisfiability procedures. Information and Computation 183(2), 140–164 (2003)
3. Barrett, C., de Moura, L., Stump, A.: SMT-COMP: Satisfiability Modulo Theories Competition. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 20–23. Springer, Heidelberg (2005)
4. Cavalcanti, A.L.C., Sampaio, A.C.A., Woodcock, J.C.P.: A Refinement Strategy for *Circus*. Formal Aspects of Computing 15(2-3), 146–181 (2003)
5. de Moura, L., Bjørner, N.: Model-based theory combination. Electronic Notes in Theoretical Computer Science 198(2), 37–49 (2008)
6. de Moura, L.M., Björner, N.: Proofs and refutations, and Z3. In: LPAR Workshops. CEUR Workshop Proceedings, vol. 418 (2008)
7. Déharbe, D., de Oliveira, D., Fontaine, P.: Combining decision procedures by (model-)equality propagation. In: Brazil. Symp. Formal Methods, pp. 51–66 (2008)
8. Déharbe, D., Ranise, S.: Bdd-driven first-order satisfiability procedures (extended version). Research report 4630, LORIA (2002)
9. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
10. Fontaine, P.: Combinations of theories and the Bernays-Schönfinkel-Ramsey class. In: 4th Int'l Verification Workshop (VERIFY) (2007)
11. Fontaine, P., Marion, J.-Y., Merz, S., Nieto, L.P., Tiu, A.: Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 167–181. Springer, Heidelberg (2006)
12. Grundy, J., Melham, T., Krstić, S., McLaughlin, S.: Tool building requirements for an API to first-order solvers. Electronic Notes in Theoretical Computer Science 144, 15–26 (2005)
13. Nieuwenhuis, R., Oliveras, A.: DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 321–334. Springer, Heidelberg (2005)
14. Oliveira, M., Gurgel, C., de Castro, A.: Crefine: Support for the Circus refinement calculus. In: IEEE Intl. Conf. Software Engineering and Formal Methods, pp. 281–290. IEEE Comp. Soc. Press, Los Alamitos (2008)
15. Ranise, S., Tinelli, C.: The SMT-LIB standard: Version 1.2 (August 2006)
16. Schulz, S.: System Description: E 0.81. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 223–228. Springer, Heidelberg (2004)
17. Stump, A.: Proof Checking Technology for Satisfiability Modulo Theories. In: Logical Frameworks and Meta-Languages: Theory and Practice (2008)
18. Weidenbach, C., Schmidt, R., Hillenbrand, T., Rusev, R., Topic, D.: System description: Spass version 3.0. In: Pfenning, F. (ed.) CADE 2007. LNCS, vol. 4603, pp. 514–520. Springer, Heidelberg (2007)

Divvy: An ATP Meta-system Based on Axiom Relevance Ordering

Alex Roederer, Yury Puzis, and Geoff Sutcliffe*

University of Miami, USA

Abstract. This paper describes two syntactic relevance orderings on the axioms available for proving a given conjecture, and an ATP meta-system that uses the orderings to select axioms to use in proof attempts. The system has been evaluated, and the results show that it is effective.

1 Motivation, History, and Overview

In recent years there has been growing demand for Automated Theorem Proving (ATP) in large theories. A *large theory* is one that has many functors and predicates, has many axioms of which typically only a few are required for the proof of a theorem, and many theorems to be proved using the axioms. Examples of large theories that are in a form, or have been translated to a form, suitable for ATP include the SUMO ontology, the Cyc knowledge base, the Mizar mathematical library, the YAGO knowledge base, WordNet, and the MeSH vocabulary thesaurus.

Large theories pose challenges for ATP systems, which are different from the challenges of small theories. These include parsing and building data structures for the large numbers of formulae, loading and preprocessing the axioms only once, selecting axioms that are likely to be used for proving a given conjecture, and extracting heuristics and lemmas from proofs to improve subsequent performance. The work described in this paper addresses the issue of selecting axioms from a large theory, to obtain a proof of a conjecture. The aim is to select as few axioms as possible, but enough to obtain a proof. There have been previous efforts in this direction, including abstraction-based techniques [1], analysis of possible inference chains from the conjecture [6], axiom selection based on symbol count [10], axiom selection based on symbol overlap [5], axiom selection based on models of the formulae [7], and axiom selection based on machine learning from previous proofs [11]. In all cases, the general approach has been to order the axioms according to their *relevance* to the conjecture, and then use or select axioms with respect to the ordering. A system that selects axioms, i.e., the non-selected axioms are made unavailable, typically iterates a process of selecting axioms, executing an ATP system to try find a proof, and if the ATP system is unsuccessful (within the resource limits imposed) looping to make a new selection. A common feature of previous efforts is that they start with the

* Currently at the Automation of Logic group, Max Planck Institut für Informatik.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 157–162, 2009.

axioms at the top of the relevance ordering, and work their way down. This is somewhat fragile, because the system performs poorly if just one necessary axiom is placed low in the ordering.

This paper describes two syntactic relevance orderings on the axioms available for proving a given conjecture, and an ATP meta-system called Divvy that selects axioms with respect to an ordering. Divvy uses a dividing approach that "starts in the middle", thus overcoming some of the fragility experienced by approaches that "start at the top". The combined system has been evaluated, and the results show that it is effective.

2 Two Syntactic Relevance Orderings

2.1 Ordering Based on Symbol Overlap

This relevance measure is based on the intuitive notion of whether or not formulae are "talking about the same things". This is measured in terms of the extent to which the formulae use the same predicate and function symbols.

First, the *contextual direct relevance* between all formulae pairs $F_1, F_2 \in S = Axioms \cup \{Conjecture\}$ is measured by

$$\frac{\sum_{s \in (sym(F_1) \cap sym(F_2))} \left(1 - \frac{|\{f : f \in S, s \in sym(f)\}|}{|S|}\right)}{|sym(F_1) \cup sym(F_2)|}$$

where $sym(F)$ is the set of function and predicate symbols occurring in F. The numerator sums the symbol weights for the symbols that occur in both F_1 and F_2, where the weight is 0 for symbols that occur in all formulae in S, and higher (approaching 1) for symbols that occur in fewer formulae. This sum is scaled by the number of unique symbols in F_1 and F_2, to produce a value in the range 0 to 1. (This a variant of the Jaccard similarity coefficient [4].) Next, the *contextual path relevance* of every path $F_a = F_1 \cdot F_2 \cdot \ldots \cdot F_n = F_c$ from an axiom F_a to the conjecture F_c is calculated as the smallest contextual direct relevance in the path, divided by the length of the path. This captures the intuition that a path is only as strong as its weakest link, and that relevance decreases with distance. Finally, the *contextual indirect relevance* between F_a and F_c is taken as the maximal contextual path relevance over all paths connecting F_a to F_c. (The latter two steps are implemented together using Dijkstra's algorithm.)

There are several adaptations of this basic measure, including different ways of weighting symbols, taking variables into account, and different options for treating predicate and function symbols separately. The basic and adapted measures have been implemented in C++, as the Prophet tool. It is available for use in the SystemB4TPTP interface, at `http://www.tptp.org/cgi-bin/SystemB4TPTP`.

2.2 Ordering Based on Latent Semantic Analysis

Latent Semantic Analysis (LSA) is a technique for analysing of relationships between documents, using the terms they contain [3]. LSA has been used to

compute the relevance of axioms to a conjecture by treating the formulae as documents, and the predicate and function symbols as the terms they contain.

The computation of axiom relevance using LSA is a three step process. First, a relationship strength between every pair of symbols is computed. An initial relationship strength is computed based on the co-occurrences of the symbols in the formulae, and the total number of formulae containing the symbols (like the numerator of Prophet's contextual direct relevance). The final relationship strength is computed by repeatedly combining the existing relationship strength with the relationship strengths between each of the two symbols and each other symbol, i.e., taking into account transitive relationships between symbols. Second, a relationship strength vector is computed for each formula. The vector has an entry for each symbol. A symbol's entry is the sum, across all other symbols, of the product of the relationship strength between the two symbols, and the number of occurrences of the other symbol in the formula (so that other symbols that do not occur in the formula make no contribution to the vector entry). Finally, the relevance of each axiom to the conjecture is computed as the dot product of their symbol relationship strength vectors.

The LSA approach to computing axiom relevance has been implemented in C, as the Automated Prophesier of Relevance Incorporating Latent Semantics (APRILS) tool. It is available for use through the SystemB4TPTP interface.

APRILS has been evaluated against Prophet by comparing their relevance orderings of the axioms for 1337 TPTP problems for which EP has found a proof. Their highest ranks of an axiom used in the proof of each problem were compared - a higher rank is worse. APRILS did better than Prophet for 654 (49%) of the problems, and tied for 301 (23%) of the problems. Most of the ties were on problems containing few axioms. APRILS' methods are better suited for problems with large numbers of axioms, which contain more semantic information.

3 The Divvy ATP Meta-system

The Divvy ATP meta-system uses a relevance ordering to select subsets of the axioms available for proving a given conjecture, and attempts to prove the conjecture from the selected axioms. The basic idea is very simple - start by selecting the top half of the axioms (in the ordering), and try prove the conjecture. If the conjecture is proved then stop. If the conjecture is countersatisfiable[1] with respect to the selected axioms, then more axioms are needed. If nothing is established about the conjecture, e.g., because the proof attempt reached a resource limit, then fewer or more axioms are needed. The top quarter, and then the top three quarters, of the axioms are then considered. If neither of those produce a proof, the top eighth, three eighths, five eighths, and seven eighths, are selected. This continues, selecting an odd number of sixteenths, thirtysecondths, etc., until a granularity limit (halves, quarters, eighths, etc.), or global resource limit,

[1] *Countersatisfiable* is an SZS ontology status value [9], meaning that the conjecture is not provable from the axioms.

is reached. In all cases, selection of fewer axioms than the maximal number that has led to a countersatisfiable result is rejected.

The basic process is improved by several optimizations. First, a proof attempt using all the axioms can be made before the dividing and selecting starts. Second, the user can specify a maximal number of axioms to ever be selected, and only that number of axioms is considered from the relevance ordered list. Third, before each proof attempt, model finders can be run to try show that the conjecture is countersatisfiable with respect to the selected axioms. This aims to avoid proof attempts that fail, and to raise the maximal number that has led to a countersatisfiable result. Fourth, the proof attempts can use an ATP system that only establishes an assurance of the existence of a proof, and ultimately an ATP system that outputs a full proof is run on the conjecture and selected axioms. Fifth, the user can specify the ATP systems to be used for proof assurance, proof finding, and model finding. Sixth, and most relevant, the user can specify the tool to be used for computing the relevance ordering for the axioms.

CPU limits are imposed on the initial ATP system run using all axioms, the model finding runs, the individual ATP systems runs using selected axioms, and the overall process. The CPU limit on the individual ATP systems runs is dynamically adjusted to take into account the minimal number of axioms that must be selected (based on countersatisfiable results).

Divvy is implemented in C, and relies heavily on the TPTP world infrastructure for manipulating the formulae, running the relevance measuring tool, and running the ATP systems. It is available for use through the SystemOnTPTP interface at `http://www.tptp.org/cgi-bin/SystemOnTPTP`.

4 Evaluation

Divvy, using Prophet and APRILS, has been evaluated on the MPTP challenge problems. These are two sets of 252 problems extracted from the Mizar mathematical library by the MPTP process. The problems in the "Bushy" set have the axioms that the MPTP process determines might be necessary for a proof. While the MPTP process tries to minimize this set, each problem contains many unnecessary axioms. The problems in the "Chainy" set have all preceding Mizar knowledge as axioms, i.e., there are very many unnecessary axioms. The MPTP challenge problems, and results for some well known ATP systems, are available at `http://www.tptp.org/Challenges/MPTPChallenge/`.

Divvy was configured to use the E 1.0 ATP system [8] to establish the existence of proofs, Paradox 3.0 [2] to establish countersatisfiability, and EP 1.0 to ultimately produce the proofs. Some testing was also done using Fampire instead of E. Fampire 1.3 is the plain ATP system with the best results on the MPTP challenge. While the results using Fampire are of interest, the full evaluation was done using E because of its availability and stable high performance as a monolithic ATP system. (Fampire, in contrast, is a little known and undocumented system.) An overall CPU limit of 300s was imposed, and an initial ATYP system run using all axioms was done with a 60s CPU limit, i.e., leaving at least 240s

Table 1. Divvy results for the MPTP challenge problems, for various axiom orderings

System	Bushy			Chainy		
	Total	<60s	>60s	Total	<60s	>60s
E	141	139	2	91	80	11
Divvy(E)+Original	163	139	24	80	80	0
Divvy(E)+Reverse	151	141	10	92	82	10
Divvy(E)+Prophet-indirect	170	137	33	113	81	32
Divvy(E)+Prophet-direct	175	137	38	118	81	37
Divvy(E)+APRILS	180	140	40	117	80	37
Fampire	191	165	26	126	78	48
Divvy(F)+APRILS	186	165	21	132	68	64

for ATP system runs using selected axioms. All axioms were always available for selection, and the dividing was done down to a granularity of eighths. The testing was done on a 2.8GHz Intel Xeon computer with 1GB memory, and running Linux 2.6.

Table 1 tabulates the results. The "E" and "Fampire" rows give the results for E and Fampire alone. The "Divvy(*System*)+*Ordering*" rows give the results for Divvy using either the E or Fampire system to establish the existence of a proof, and the axioms ordered either as they are in the original problem, the reverse of the original problem, or according to the relevance values computed by Prophet or APRILS. The "<60s" and ">60s" columns give the number of problems solved in less than and more than 60s. For the "Divvy" rows this separates the problems solved in the initial ATP system run using all axioms from those that benefited from the axiom selection. Note that the number solved in less than 60s is not constant across the "Divvy(E)" rows, due to the reordering of the axioms.

The results show that the basic Divvy idea, despite its simplicity, takes effective advantage of axiom ordering. The fact that Divvy improves on E without any axiom ordering indicates that the original order of the axioms in the MPTP challenge problems is better than random - more relevant axioms already occur earlier in the problems, thanks to the nature of the MPTP process. This is confirmed by the reduced performance using the axioms in reverse order. The successive improvements obtained by adding explicit axiom orderings shows that Prophet and APRILS do compute meaningful axiom relevance. The improvement from the original ordering to APRILS' ordering is 10% - a non-trivial improvement for these challenging problems. When Fampire is used as the underlying ATP system, the reduced performance of Divvy on the Bushy problems is believed to be an artifact of Fampire's strategy scheduling, which works effectively only with a reasonably large CPU limit (e.g., around 300s). When Fampire is used in Divvy, with Divvy being given an overall CPU limit of 300s, the individual ATP system runs receive relatively small CPU limits, typically less than 50s. The significantly larger number of unnecessary axioms in the Chainy problems makes the relevance ordering and axiom selection more valuable for Fampire.

5 Conclusion

This work and implemented system have shown that axiom ordering in conjunction with axiom selection is an effective technique for proving theorems in large theories. The syntactic orderings presented in this paper have been shown to be meaningful, and are available for use in other contexts. The Divvy ATP meta-system has shown that "starting in the middle" is a useful way of selecting axioms, and can be used with any meaningful axiom ordering. Current work is focussing on optimizing the accuracy and runtime performance of APRILS.

References

1. Barker-Plummer, D.: Gazing: An Approach to the Problem of Definition and Lemma Use. Journal of Automated Reasoning 8(3), 311–344 (1992)
2. Claessen, K., Sorensson, N.: New Techniques that Improve MACE-style Finite Model Finding. In: Baumgartner, P., Fermueller, C. (eds.) Proceedings of the Workshop on Model Computation - Principles, Algorithms, Applications (2003)
3. Deerwester, S., Dumais, S., Furnas, G., Landauer, K., Harschman, R.: Indexing by Latent Semantic Analysis. Journal of the American Society for Information Science 41(6), 391–407 (1990)
4. Jaccard, P.: Étude Comparative de la Distribution Florale Dans une Portion des Alpes et des Jura. Bulletin del la Société Vaudoise des Sciences Naturelles 37, 547–579 (1901)
5. Meng, J., Paulson, L.: Lightweight Relevance Filtering for Machine-Generated Resolution Problems. In: Sutcliffe, G., Schmidt, R., Schulz, S. (eds.) Proceedings of the FLoC 2006 Workshop on Empirically Successful Computerized Reasoning. CEUR Workshop Proceedings, vol. 192, pp. 53–69 (2006)
6. Plaisted, D.A., Yahya, A.: A Relevance Restriction Strategy for Automated Deduction. Artificial Intelligence 144(1-2), 59–93 (2003)
7. Pudlak, P.: Semantic Selection of Premisses for Automated Theorem Proving. In: Urban, J., Sutcliffe, G., Schulz, S. (eds.) Proceedings of the CADE-21 Workshop on Empirically Successful Automated Reasoning in Large Theories. CEUR Workshop Proceedings, vol. 257, pp. 27–44 (2007)
8. Schulz, S.: E: A Brainiac Theorem Prover. AI Communications 15(2-3), 111–126 (2002)
9. Sutcliffe, G.: The SZS Ontologies for Automated Reasoning Software. In: Sutcliffe, G., Rudnicki, P., Schmidt, R., Konev, B., Schulz, S. (eds.) Proceedings of the Workshop or Knowledge Exchange: Automated Provers and Proof Assistants. CEUR Workshop Proceedings, vol. 418, pp. 38–49 (2008)
10. Sutcliffe, G., Dvorsky, A.: Proving Harder Theorems by Axiom Reduction. In: Russell, I., Haller, S. (eds.) Proceedings of the 16th International FLAIRS Conference, pp. 108–112. AAAI Press, Menlo Park (2003)
11. Urban, J., Sutcliffe, G., Pudlak, P., Vyskocil, J.: MaLARea SG1: Machine Learner for Automated Reasoning with Semantic Guidance. In: Baumgartner, P., Armando, A., Gilles, D. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 441–456. Springer, Heidelberg (2008)

Instantiation-Based Automated Reasoning: From Theory to Practice

Konstantin Korovin*

University of Manchester, UK
korovin@cs.man.ac.uk

Instantiation-based automated reasoning aims at combining the efficiency of propositional SAT and SMT technologies with the expressiveness of first-order logic. Propositional SAT and SMT solvers are probably the most successful reasoners applied to real-world problems, due to extremely efficient propositional methods and optimized implementations. However, the expressiveness of first-order logic is essential in many applications ranging from formal verification of software and hardware to knowledge representation and querying. Therefore, there is a growing demand to integrate efficient propositional and more generally ground reasoning modulo theories into first-order reasoning.

The basic idea behind instantiation-based reasoning is to interleave smart generation of instances of first-order formulae with propositional type reasoning. Instantiation-based methods can be divided into two major categories: (i) fine-grained interleaving of instantiation with efficient propositional inference rules, and (ii) modular combination of instantiation and propositional reasoning. Examples from the first category include the disconnection calculus (DCTP) [8,24], which combines instance generation with an efficient tableau data structure, and the model evolution calculus (ME) [6], which interleaves instance generation with DPLL style reasoning. Both DCTP and ME methods have advanced implementations DCTP [33] and Darwin [3], respectively.

Our approach to instantiation-based reasoning [15,21] falls into the second category, where propositional reasoning is integrated in a modular fashion and was inspired by work on hyper-linking and its extensions (see [23,31,18]). The main advantage of the modular combination is that it allows one to use off-the-shelf SAT/SMT solvers in the context of first-order reasoning. One of our main goals was to develop a flexible theoretical framework, called Inst-Gen, for modular combination of instantiation with propositional reasoning and more generally with ground reasoning modulo theories. This framework provides methods for proving completeness of instantiation calculi, powerful redundancy elimination criteria and flexible saturation strategies. All these ingredients are crucial for developing reasoning systems which can be used in practical applications. We also show that most of the powerful machinery developed in the resolution-based framework can be suitably adapted for the Inst-Gen method.

Based on these theoretical results we have developed and implemented an automated reasoning system, called iProver [22]. iProver features state-of-the-art implementation techniques such as unification and simplification indexes;

* Supported by The Royal Society.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 163–166, 2009.

semantically-guided inferences based on propositional models; redundancy elimination based on dismatching constraints, blocking of non-proper instantiations and global subsumption. For propositional reasoning iProver uses an optimised SAT solver MiniSat [12]. For efficient equational and theory reasoning, we are currently integrating (joint work with C. Sticksel) state-of-the-art SMT solvers CVC3 [1] and Z3 [11] into iProver.

One of the major success stories of instantiation-based methods is in reasoning with the effectively propositional (EPR) fragment of first-order logic, also called the Bernays-Schönfinkel class. All known instantiation-based methods are decision procedures for the EPR fragment. Recently it was shown that the EPR fragment has a number of applications in areas such as bounded model checking, planning, logic programming and knowledge representation [28,30,19,13]. As witnessed by the CASC competition [34] instantiation-based methods considerably outperform other methods in the EPR division. The importance of the EPR fragment triggered the development of a number of dedicated methods [10,29,5], but they have not yet been extensively evaluated and compared with general-purpose instantiation-based methods.

There are many challenges remaining in the area of instantiation-based reasoning. Let me just mention some of them. The first challenge is the integration of theory reasoning and, in particular, reasoning with real and integer arithmetic. There are results on the integration of equational reasoning [25,16,7] and some initial results on the integration of theory reasoning [17,4], but these should be considerably extended to cover more problems coming from applications.

The second challenge is combining instantiation-based methods with other reasoning methods such as resolution. Refinements of resolution are decision procedures for many important fragments of first-order logic including the guarded fragment and fragments corresponding to translations of various modal and description logics (see e.g., [14,32,20]). It is a natural progression to combine instantiation-based methods with resolution in order to obtain efficient reasoning methods for combinations of the EPR fragment and fragments decidable by resolution (note that in general, the resulting fragments can be undecidable).

The third challenge is in applying instantiation-based methods in reasoning with large theories. There is growing interest using first-order reasoning systems in problems involving large theories and, in particular, large knowledge-bases [26]. Initial experiments show that the performance of instantiation-based methods on such problems is promising but more research is needed in this area.

The fourth challenge is in applying instantiation-based methods to model finding. Instantiation-based methods are designed mainly to prove validity of first-order formulae. In many applications the dual problem of proving satisfiability of first-order formulae, or model finding, is equally important. Recently it was shown that the problem of finite model finding for first-order logic can be reduced to the satisfiability problem in the EPR fragment [2,27]. Therefore, instantiation-based methods can be naturally used for finite model finding and such capabilities are incorporated into Darwin and iProver. Already finding models with small domain sizes is a challenging problem due to enormous search

spaces. Symmetry reduction is one of the main methods used to reduce redundant computations in model finders (see e.g., [9,2]). More research is required to develop powerful symmetry reductions in the context of instantiation-based methods. Finally, little is known about model finding in the case of very large models or infinite models.

To conclude, instantiation-based reasoning is a rapidly developing area with high potential and exciting research challenges.

References

1. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
2. Baumgartner, P., Fuchs, A., de Nivelle, H., Tinelli, C.: Computing finite models by reduction to function-free clause logic. In: Third Workshop on Disproving - Non-Theorems, Non-Validity, Non-Provability (DISPROVING 2006) (July 2006)
3. Baumgartner, P., Fuchs, A., Tinelli, C.: Implementing the model evolution calculus. International Journal on Artificial Intelligence Tools 15(1), 21–52 (2006)
4. Baumgartner, P., Fuchs, A., Tinelli, C.: ME(LIA) – Model evolution with linear integer arithmetic constraints. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS, vol. 5330, pp. 258–273. Springer, Heidelberg (2008)
5. Baumgartner, P., Schmidt, R.A.: Blocking and other enhancements for bottom-up model generation methods. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS, vol. 4130, pp. 125–139. Springer, Heidelberg (2006)
6. Baumgartner, P., Tinelli, C.: The model evolution calculus. In: Baader, F. (ed.) CADE 2003. LNCS, vol. 2741, pp. 350–364. Springer, Heidelberg (2003)
7. Baumgartner, P., Tinelli, C.: The model evolution calculus with equality. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS, vol. 3632, pp. 392–408. Springer, Heidelberg (2005)
8. Billon, J.-P.: The disconnection method: a confluent integration of unification in the analytic framework. In: Miglioli, P., Moscato, U., Ornaghi, M., Mundici, D. (eds.) TABLEAUX 1996. LNCS (LNAI), vol. 1071, pp. 110–126. Springer, Heidelberg (1996)
9. Claessen, K., Sörensson, N.: New techniques that improve MACE-style model finding. In: Proc. of Workshop on Model Computation (MODEL) (2003)
10. de Moura, L.M., Bjørner, N.: Deciding effectively propositional logic using DPLL and substitution sets. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS, vol. 5195, pp. 410–425. Springer, Heidelberg (2008)
11. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
12. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
13. Eiter, T., Faber, W., Traxler, P.: Testing strong equivalence of datalog programs - implementation and examples. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS, vol. 3662, pp. 437–441. Springer, Heidelberg (2005)
14. Ganzinger, H., de Nivelle, H.: A superposition decision procedure for the guarded fragment with equality. In: Proc. of LICS 1999, pp. 295–304 (1999)
15. Ganzinger, H., Korovin, K.: New directions in instantiation-based theorem proving. In: Proc. 18th IEEE Symposium on LICS, pp. 55–64. IEEE, Los Alamitos (2003)

16. Ganzinger, H., Korovin, K.: Integrating equational reasoning into instantiation-based theorem proving. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 71–84. Springer, Heidelberg (2004)
17. Ganzinger, H., Korovin, K.: Theory Instantiation. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS, vol. 4246, pp. 497–511. Springer, Heidelberg (2006)
18. Hooker, J.N., Rago, G., Chandru, V., Shrivastava, A.: Partial instantiation methods for inference in first order logic. J. Autom. Reasoning 28, 371–396 (2002)
19. Hustadt, U., Motik, B., Sattler, U.: Reducing SHIQ-description logic to disjunctive datalog programs. In: The Ninth International Conference on Principles of Knowledge Representation and Reasoning, pp. 152–162. AAAI Press, Menlo Park (2004)
20. Kazakov, Y., Motik, B.: A resolution-based decision procedure for SHOIQ. J. Autom. Reasoning 40(2-3), 89–116 (2008)
21. Korovin, K.: An invitation to instantiation-based reasoning: a modular approach. In: Podelski, A., Voronkov, A., Wilhelm, R. (eds.) Volume in memoriam of Harald Ganzinger. Springer, Heidelberg (2006) (invited paper) (to appear)
22. Korovin, K.: iProver - an instantiation-based theorem prover for first-order logic (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS, vol. 5195, pp. 292–298. Springer, Heidelberg (2008)
23. Lee, S.-J., Plaisted, D.: Eliminating duplication with the Hyper-linking strategy. J. Autom. Reasoning 9, 25–42 (1992)
24. Letz, R., Stenz, G.: Proof and model generation with disconnection tableaux. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS (LNAI), vol. 2250, pp. 142–156. Springer, Heidelberg (2001)
25. Letz, R., Stenz, G.: Integration of equality reasoning into the disconnection calculus. In: Egly, U., Fermüller, C. (eds.) TABLEAUX 2002. LNCS, vol. 2381, pp. 176–190. Springer, Heidelberg (2002)
26. Pease, A., Sutcliffe, G., Siegel, N., Trac, S.: The annual SUMO reasoning prizes at CASC. In: The First International Workshop on Practical Aspects of Automated Reasoning. CEUR Workshop Proceedings, vol. 373 (2008)
27. Pérez, J.A.N.: Encoding and Solving Problems in Effectively Propositional Logic. PhD thesis, University of Manchester (2007)
28. Pérez, J.A.N., Voronkov, A.: Encodings of bounded LTL model checking in effectively propositional logic. In: Pfenning, F. (ed.) CADE 2007. LNCS, vol. 4603, pp. 346–361. Springer, Heidelberg (2007)
29. Pérez, J.A.N., Voronkov, A.: Proof systems for effectively propositional logic. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS, vol. 5195, pp. 426–440. Springer, Heidelberg (2008)
30. Pérez, J.A.N., Voronkov, A.: Planning with effectively propositional logic. In: Podelski, A., Voronkov, A., Wilhelm, R. (eds.) Volume in memoriam of Harald Ganzinger. Springer, Heidelberg (to appear) (invited paper)
31. Plaisted, D., Zhu, Y.: Ordered semantic hyper-linking. J. Autom. Reasoning 25(3), 167–217 (2000)
32. Schmidt, R.A., Hustadt, U.: First-order resolution methods for modal logics. In: Podelski, A., Voronkov, A., Wilhelm, R. (eds.) Volume in memoriam of Harald Ganzinger. LNCS. Springer, Heidelberg (2006) (invited overview paper) (to appear)
33. Stenz, G.: DCTP 1.2 - system abstract. In: Egly, U., Fermüller, C. (eds.) TABLEAUX 2002. LNCS, vol. 2381, pp. 335–340. Springer, Heidelberg (2002)
34. Sutcliffe, G.: The 4th IJCAR automated theorem proving system competition - CASC-J4. AI Communications 22(1), 59–72 (2009)

Interpolant Generation for UTVPI*

Alessandro Cimatti[1], Alberto Griggio[2], and Roberto Sebastiani[2]

[1] FBK-Irst, Trento, Italy
cimatti@fbk.eu
[2] DISI, Università di Trento, Italy
{griggio,rseba}@disi.unitn.it

Abstract. The problem of computing Craig interpolants in SMT has recently received a lot of interest, mainly for its applications in formal verification. Efficient algorithms for interpolant generation have been presented for some theories of interest –including that of equality and uninterpreted functions ($\mathcal{EUF}$),linear arithmetic over the rationals ($\mathcal{LA}(\mathbb{Q})$), and some fragments of linear arithmetic over the integers ($\mathcal{LA}(\mathbb{Z})$)– and they are successfully used within model checking tools.

In this paper we address the problem of computing interpolants in the theory of Unit-Two-Variable-Per-Inequality ($\mathcal{UTVPI}$). This theory is a very useful fragment of $\mathcal{LA}(\mathbb{Z})$, since it is expressive enough to encode many hardware and software verification queries while still admitting a polynomial time decision procedure.

We present an efficient graph-based algorithm for interpolant generation in $\mathcal{UTVPI}$, which exploits the power of modern SMT techniques. We have implemented our new algorithm within the MathSAT SMT solver. Our experimental evaluation demonstrates both the efficiency and the usefulness of the new algorithm.

1 Motivations and Goals

Given two formulas A and B such that $A \wedge B$ is inconsistent, a *Craig interpolant* (simply "interpolant" hereafter) for (A, B) is a formula I s.t. A entails I, $I \wedge B$ is inconsistent, and all uninterpreted symbols of I occur in both A and B. Since the seminal work of McMillan [17], interpolation has been recognized to be a substantial tool for formal verification. For instance, in the context of software model checking based on counter-example-guided-abstraction-refinement (CEGAR), interpolants of quantifier-free formulas in suitable theories are computed for automatically refining abstractions in order to rule out spurious counterexamples (see, e.g. [8,11,19]). This technique is used by state-of-the-art software model checkers like, e.g., Blast [2] and Impact [19]. Consequently, the problem of computing interpolants in SMT has received a lot of interest in the last years (e.g., [18,26,12,22,13,5,10,3,14]).

* The first author is partly supported by the European Commission under project FP7-2007-IST-1-217069 COCONUT. The second and third authors are partly supported by SRC under GRC Custom Research Project 2009-TJ-1880 WOLFLING, and by MIUR under PRIN project 20079E5KM8_002.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 167–182, 2009.

In the recent years, efficient algorithms and tools for interpolant generation for quantifier-free formulas in SMT have been presented for some theories of interest, including that of equality and uninterpreted functions ($\mathcal{EUF}$) [18,14], linear arithmetic over the rationals ($\mathcal{LA}(\mathbb{Q})$) [18,22,5], and for their combination [26,25,5,3]. In many applications, however, the domain of rational numbers is often inadequate for representing variables, which could be represented much more precisely in the integer domain (see, e.g., [7,15]). Unfortunately, the computation of interpolants in the theory of linear arithmetic over the integers ($\mathcal{LA}(\mathbb{Z})$) raises many more problems than $\mathcal{LA}(\mathbb{Q})$, and in fact there is no known and efficient algorithm for computing interpolants in $\mathcal{LA}(\mathbb{Z})$. The only known algorithm is based on quantifier elimination, which is typically prohibitively expensive, and also requires the introduction of divisibility predicates. Therefore, it is useful to investigate interpolation for *fragments* of $\mathcal{LA}(\mathbb{Z})$, simple enough to be treated efficiently, although general enough to allow for encoding a significant amount of verification problems. To this extent, Jain, Clarke and Grunberg [10] proposed efficient algorithms for computing interpolants for conjunctions of linear Diophantine equations and disequations and for conjunctions of linear modular equations, and showed that these algorithms enabled the verification of simple programs which could not be previously checked by CEGAR-based model checkers.

In this paper, we move along the same track of Jain et al. [10], and we tackle the problem of computing interpolants in another important fragment of $\mathcal{LA}(\mathbb{Z})$, the theory of Unit-Two-Variable-Per-Inequality ($\mathcal{UTVPI}$). In $\mathcal{UTVPI}$ a formula is a Boolean combination of atoms in the form $(0 \leq ax_1 + bx_2 + k)$, where x_i are variables over $\mathbb{Z}$, k is an integer constant, and $a, b \in \{-1, 0, 1\}$. $\mathcal{UTVPI}$ is a very interesting theory: it generalizes the well-known Difference Logic ($\mathcal{DL}(\mathbb{Z})$) (where the a and b coefficient are forced to the 1 and -1 values, respectively), and it is one of the most expressive fragments of $\mathcal{LA}(\mathbb{Z})$ with a polynomial decision procedure [9]. (In fact, it is sufficient to extend the fragment to contain three unit variables, or to add non-unit coefficients, to make the decision problems NP-complete.) $\mathcal{UTVPI}$ is also a very useful fragment of $\mathcal{LA}(\mathbb{Z})$, since it allows to naturally express the queries occurring in many hardware and software verification problems [1,24].

The problem of satisfiability modulo $\mathcal{UTVPI}$ can be tackled following the approach proposed in [20,16], where the consistency check of a conjunction of $\mathcal{UTVPI}$ constraints is based on an encoding into $\mathcal{DL}$. This allows for the use of very efficient graph-based decision procedures for $\mathcal{DL}$ [21,6]: these algorithms have a $O(n \cdot m)$ time complexity for problems with n variables and m constraints, and are extremely fast in practice. In addition, they have all the features required for a tight integration within a modern SMT solver: incrementalitly and backtrackability, construction of minimal conflict sets, and deduction of unassigned literals (see [23] for a survey).

The contribution of this paper is the first interpolation algorithm for $\mathcal{UTVPI}$. The algorithm follows the decision procedure for $\mathcal{UTVPI}$, working in two phases. In the first phase, it checks whether the conjunction of $\mathcal{UTVPI}$ constraints

is inconsistent in the rational domain ($\mathcal{UTVPI}(\mathbb{Q})$). If so, an interpolant is obtained with a generalization of the graph-based algorithm for $\mathcal{DL}$ [5]. The second phase is entered if the problem is consistent in the rational domain, but inconsistent in the integer domain. The second phase is also based on the analysis of the graph resulting from the encoding into $\mathcal{DL}$. However, unlike with $\mathcal{DL}(\mathbb{Q})$ and $\mathcal{DL}(\mathbb{Z})$, the problem of interpolant generation on $\mathcal{UTVPI}(\mathbb{Z})$ is by no means a straightforward variant of that in $\mathcal{UTVPI}(\mathbb{Q})$, and several cases must be covered.

The proposed algorithm has the following merits. First, it generates interpolants that are within $\mathcal{UTVPI}$. This is important for applications where the computed interpolants are iteratively combined with the original problem, such as in interpolation-based bounded model checking [17]. Second, the approach can be easily implemented on top of a modern SMT procedure for $\mathcal{UTVPI}$, and runs with very limited overhead.

We have implemented our new algorithm within the MathSAT SMT solver [4], and performed experiments in order to evaluate both its efficiency and its usefulness. Our results demonstrate not only that our specialized $\mathcal{UTVPI}(\mathbb{Q})$ algorithm is faster and generates smaller interpolants than general $\mathcal{LA}(\mathbb{Q})$ interpolation procedures (and thus is interesting in itself), but also that our $\mathcal{UTVPI}(\mathbb{Z})$ algorithm can be useful for the verification of software model checking problems which require reasoning on the integers, and which could not be proved before by the Blast software model checker [2], due to the approximation resulting from its use of $\mathcal{LA}(\mathbb{Q})$ interpolation procedures.

Content of the Paper. In §2 we provide the necessary background knowledge on SMT and interpolant generation in SMT. In §3 we present our novel graph-based interpolant technique for $\mathcal{UTVPI}(\mathbb{Q})$, whilst in §4 we show how to extend it to the case of $\mathcal{UTVPI}$ over $\mathbb{Z}$. In §5 we report some empirical results. In §6 we draw some conclusions, and outline directions for future research.

2 Background

Satisfiability Modulo Theory – SMT. Our setting is standard first order logic. We use the standard notions of theory, satisfiability, validity, logical consequence. We call *Satisfiability Modulo (the) Theory* $\mathcal{T}$, $SMT(\mathcal{T})$, the problem of deciding the satisfiability of quantifier-free formulas wrt. a background theory $\mathcal{T}$. [1] Given a theory $\mathcal{T}$, we write $\phi \models_{\mathcal{T}} \psi$ (or simply $\phi \models \psi$) to denote that the formula ψ is a logical consequence of ϕ in the theory $\mathcal{T}$. With $\phi \preceq \psi$ we denote that all uninterpreted (in $\mathcal{T}$) symbols of ϕ appear in ψ. Without loss of generality, we also assume that the formulas are in Conjunctive Normal Form (CNF). If C is a clause, $C \downarrow B$ is the clause obtained from C by removing all the literals whose atoms do not occur in B, and $C \setminus B$ that obtained by removing all the literals whose atoms do occur in B. With a little abuse of notation, we might

[1] The general definition of SMT deals also with quantified formulas. Nevertheless, in this paper we restrict our interest to quantifier-free formulas.

sometimes denote conjunctions of literals $l_1 \wedge \ldots \wedge l_n$ as sets $\{l_1, \ldots, l_n\}$ and vice versa. If η is a the set $\{l_1, \ldots, l_n\}$, we might write $\neg\eta$ to mean $\neg l_1 \vee \ldots \vee \neg l_n$.

We call $\mathcal{T}$-solver a procedure that decides the consistency of a conjunction of literals in $\mathcal{T}$. If S is a set of literals in $\mathcal{T}$, we call *$\mathcal{T}$-conflict set w.r.t. S* any subset η of S which is inconsistent in $\mathcal{T}$. We call $\neg\eta$ a *$\mathcal{T}$-lemma* (notice that $\neg\eta$ is a $\mathcal{T}$-valid clause). Given a set of clauses $S \stackrel{\text{def}}{=} \{C_1, \ldots, C_n\}$ and a clause C, we call a *resolution proof* that $\bigwedge_i C_i \models_{\mathcal{T}} C$ a DAG $\mathcal{P}$ such that:

1. C is the root of $\mathcal{P}$;
2. the leaves of $\mathcal{P}$ are either elements of S or $\mathcal{T}$-lemmas;
3. each non-leaf node C' has two parents C_{p_1} and C_{p_2} such that C_{p_1} is in the form $p \vee \phi_1$, C_{p_2} is in the form $\neg p \vee \phi_2$, and C' is $\phi_1 \vee \phi_2$. The atom p is called the *pivot* of C_{p_1} and C_{p_2}.

If C is the empty clause (denoted with $\bot$), then $\mathcal{P}$ is a *resolution proof of unsatisfiability* (or *resolution refutation*) for $\bigwedge_i C_i$.

A standard technique for solving the SMT($\mathcal{T}$) problem is to integrate a DPLL-based SAT solver and a $\mathcal{T}$-solver in a *lazy* manner (see, e.g., [23] for a detailed description). DPLL is used as an enumerator of truth assignments for the propositional abstraction of the input formula. At each step, the set of $\mathcal{T}$-literals in the current assignment is sent to the $\mathcal{T}$-solver to be checked for consistency in $\mathcal{T}$. If S is inconsistent, the $\mathcal{T}$-solver returns a conflict set η, and the corresponding $\mathcal{T}$-lemma $\neg\eta$ is added as a blocking clause in DPLL, and used to drive the back-jump mechanism. With a small modification of the embedded DPLL engine, a lazy SMT solver can also be used to generate a resolution proof of unsatisfiability, where the leaf $\mathcal{T}$-lemmas are (some of) those returned by the $\mathcal{T}$-solver.

Interpolation in SMT. We consider the $SMT(\mathcal{T})$ problem for some background theory $\mathcal{T}$. Given an ordered pair (A, B) of formulas such that $A \wedge B \models_{\mathcal{T}} \bot$, a *Craig interpolant* (simply "interpolant" hereafter) is a formula I s.t. (i) $A \models_{\mathcal{T}} I$, (ii) $I \wedge B$ is $\mathcal{T}$-inconsistent, and (iii) $I \preceq A$ and $I \preceq B$.

Following [18], an interpolant for (A, B) is constructed from a resolution proof of unsatisfiability of $A \wedge B$, generated as outlined above. The algorithm works by computing a formula I_C for each clause in the resolution refutation, such that the formula $I_\bot$ associated to the empty root clause is the computed interpolant. The algorithm can be described as follows:

Algorithm 1. Interpolant generation for $SMT(\mathcal{T})$

1. Generate a proof of unsatisfiability $\mathcal{P}$ for $A \wedge B$.
2. For every $\mathcal{T}$-lemma $\neg\eta$ occurring in $\mathcal{P}$, generate an interpolant $I_{\neg\eta}$ for $(\eta \setminus B, \eta \downarrow B)$.
3. For every input clause C in $\mathcal{P}$, set I_C to $C \downarrow B$ if $C \in A$, and to $\top$ if $C \in B$.
4. For every inner node C of $\mathcal{P}$ obtained by resolution from $C_1 \stackrel{\text{def}}{=} p \vee \phi_1$ and $C_2 \stackrel{\text{def}}{=} \neg p \vee \phi_2$, set I_C to $I_{C_1} \vee I_{C_2}$ if p does not occur in B, and to $I_{C_1} \wedge I_{C_2}$ otherwise.
5. Output $I_\bot$ as an interpolant for (A, B).

Notice that Step 2. of the algorithm is the only part which depends on the theory $\mathcal{T}$, so that the problem of interpolant generation in $SMT(\mathcal{T})$ reduces to that of finding interpolants for negations of $\mathcal{T}$-lemmas, that is, for conjunctions of $\mathcal{T}$-literals in the given theory $\mathcal{T}$. Therefore, in the rest of the paper, we shall present algorithms for conjunctions/sets of literals only, which can be extended to general formulas by simply "plugging" them into the above algorithm. Moreover, we shall assume that when computing an interpolant for an inconsistent conjunction $A \wedge B$, neither A nor B is inconsistent in itself. In such cases, in fact, the interpolant would simply be $\bot$ and $\top$ respectively.

Graph-Based Interpolation for Difference Logic. The theory of Difference Logic ($\mathcal{DL}$) is a fragment of the theory of linear arithmetic in which all atoms are inequalities of the form $(0 \leq y - x + c)$, where x and y are variables and c is an integer constant. [2] Many SMT solvers use dedicated, graph-based algorithms for checking the consistency of a set of $\mathcal{DL}$ atoms [6,21]. Intuitively, a set ϕ of $\mathcal{DL}$ atoms induces a graph $G(\phi)$ whose vertices are the variables of the atoms, and there exists an edge $x \xrightarrow{c} y$ for every $(0 \leq y - x + c) \in \phi$. ϕ is inconsistent if and only if $G(\phi)$ has a cycle of negative weight.

In [5] we extended the graph-based approach to generate interpolants. Consider the interpolation problem (A, B) where A and B are sets of inequalities as above, and let C be (the set of atoms in) a negative cycle in the graph corresponding to $A \cup B$. Since we are assuming that neither A nor B is inconsistent, the edges in the cycle can be partitioned in subsets of A and B. We call a maximal A-path of C a path $x_1 \xrightarrow{c_1} \ldots \xrightarrow{c_{n-1}} x_n$ such that (I) $x_i \xrightarrow{c_i} x_{i+1} \in A$, and (II) C contains $x' \xrightarrow{c'} x_1$ and $x_n \xrightarrow{c''} x''$ that are in B. The variables x_1 and x_n of a maximal A-path $x_1 \rightsquigarrow x_n$ are called end-point variables.

Let the *summary constraint* of a maximal A-path $x_1 \xrightarrow{c_1} \ldots \xrightarrow{c_{n-1}} x_n$ be the inequality $0 \leq x_n - x_1 + \sum_{i=1}^{n-1} c_i$. The conjunction of summary constraints of the maximal A-paths of C is an interpolant for (A, B) [5].

3 Graph-Based Interpolation for $\mathcal{UTVPI}(\mathbb{Q})$

We analyze first the simpler case of $\mathcal{UTVPI}(\mathbb{Q})$. Miné [20] showed that it is possible to encode a set of $\mathcal{UTVPI}(\mathbb{Q})$ constraints into a $\mathcal{DL}(\mathbb{Q})$ one in a satisfiability-preserving way. The encoding works as follows. We use x_i to denote variables in the $\mathcal{UTVPI}(\mathbb{Q})$ domain and u, v for variables in the $\mathcal{DL}(\mathbb{Q})$ domain. For every variable x_i, we introduce two distinct variables x_i^+ and x_i^-. We introduce a mapping Υ from $\mathcal{DL}(\mathbb{Q})$ variables to $\mathcal{UTVPI}(\mathbb{Q})$ signed variables, such that $\Upsilon(x_i^+) = x_i$ and $\Upsilon(x_i^-) = -x_i$. Υ extends to (sets of) constraints

[2] Notice that a conjunction of non-strict difference inequalities has a solution in the integer domain if and only if it has a solution in the rational domain, so that in this context we make no distinction between $\mathcal{DL}(\mathbb{Q})$ and $\mathcal{DL}(\mathbb{Z})$.

$\mathcal{UTVPI}(\mathbb{Q})$ constraints	$\mathcal{DL}(\mathbb{Q})$ constraints
$(0 \leq x_1 - x_2 + k)$	$(0 \leq x_1^+ - x_2^+ + k), (0 \leq x_2^- - x_1^- + k)$
$(0 \leq -x_1 - x_2 + k)$	$(0 \leq x_1^- - x_2^+ + k), (0 \leq x_2^- - x_1^+ + k)$
$(0 \leq x_1 + x_2 + k)$	$(0 \leq x_1^+ - x_2^- + k), (0 \leq x_2^+ - x_1^- + k)$
$(0 \leq -x_1 + k)$	$(0 \leq x_1^- - x_1^+ + 2 \cdot k)$
$(0 \leq x_1 + k)$	$(0 \leq x_1^+ - x_1^- + 2 \cdot k)$

Fig. 1. The conversion map from $\mathcal{UTVPI}(\mathbb{Q})$ to $\mathcal{DL}(\mathbb{Q})$ [20,16]

in the natural way. We say that $(x_i^+)^- = x_i^-$ and $(x_i^-)^- = x_i^+$. We say that the constraints $(0 \leq u - v)$ and $(0 \leq (v)^- - (u)^-)$ s.t. $u, v \in \{x_i^+, x_i^-\}_i$ are *dual*. We encode each $\mathcal{UTVPI}$ constraint into the conjunction of two dual $\mathcal{DL}(\mathbb{Q})$ constraints, as represented in Figure 1. For each $\mathcal{DL}(\mathbb{Q})$ constraint $(0 \leq v - u + k)$, $(0 \leq \Upsilon(v) - \Upsilon(u) + k)$ is the corresponding $\mathcal{UTVPI}(\mathbb{Q})$ constraint. Notice that the two dual $\mathcal{DL}(\mathbb{Q})$ constraints are just different representations of the original $\mathcal{UTVPI}(\mathbb{Q})$ constraint. (The two dual constraints encoding a single-variable constraint are identical, so that their conjunction is collapsed into one constraint only.) The resulting set of constraints is satisfiable in $\mathcal{DL}(\mathbb{Q})$ if and only if the original one is satisfiable in $\mathcal{UTVPI}(\mathbb{Q})$ [20,16].

Consider the pair (A, B) where A and B are sets of $\mathcal{UTVPI}(\mathbb{Q})$ constraints. We apply the map of Figure 1 and we encode (A, B) into a $\mathcal{DL}(\mathbb{Q})$ pair (A', B'), and build the constraint graph $G(A' \wedge B')$. If $G(A' \wedge B')$ has no negative cycle, we can conclude that $A' \wedge B'$ is $\mathcal{DL}(\mathbb{Q})$-consistent, and hence that $A \wedge B$ is $\mathcal{UTVPI}(\mathbb{Q})$-consistent. Otherwise, $A' \wedge B'$ is $\mathcal{DL}(\mathbb{Q})$-inconsistent, and hence $A \wedge B$ is $\mathcal{UTVPI}(\mathbb{Q})$-inconsistent. In fact, we observe that for any set of $\mathcal{DL}(\mathbb{Q})$ constraints $\{C_1, \ldots, C_n, C\}$ resulting from the encoding of some $\mathcal{UTVPI}(\mathbb{Q})$ constraints, if $\bigwedge_{i=1}^n C_i \models_{\mathcal{DL}(\mathbb{Q})} C$ then $\bigwedge_{i=1}^n \Upsilon(C_i) \models_{\mathcal{UTVPI}(\mathbb{Q})} \Upsilon(C)$.

When $A \wedge B$ is inconsistent, we can generate an $\mathcal{UTVPI}(\mathbb{Q})$-interpolant by extending the graph-based approach used for $\mathcal{DL}(\mathbb{Q})$. We consider a negative-weight cycle of $G(A' \wedge B')$, and we build an interpolant I' in $\mathcal{DL}(\mathbb{Q})$ by means of the graph-based interpolation technique described in §2. We claim that the $\mathcal{UTVPI}(\mathbb{Q})$ formula $I \stackrel{\text{def}}{=} \Upsilon(I')$ is an interpolant for (A, B). The proof is as follows. (i) I' is a conjunction of summary constraints, so it is in the form $\bigwedge_i C_i$. Therefore $A' \models_{\mathcal{DL}(\mathbb{Q})} C_i$ for all i, and so by the observation above $A \models_{\mathcal{UTVPI}(\mathbb{Q})} \Upsilon(C_i)$. Hence, $A \models_{\mathcal{UTVPI}(\mathbb{Q})} I$. (ii) From the $\mathcal{DL}(\mathbb{Q})$-inconsistency of $I' \wedge B'$ we immediately derive that $I \wedge B$ is $\mathcal{UTVPI}(\mathbb{Q})$-inconsistent. (iii) $I \preceq A$ and $I \preceq B$ derive from $I' \preceq A'$ and $I' \preceq B'$ by the definitions of Υ and the map of Figure 1.

Example 1. Consider the following sets of $\mathcal{UTVPI}(\mathbb{Q})$ constraints:

$$A = \{(0 \leq -x_2 - x_1 + 3), (0 \leq x_1 + x_3 + 1), \\ (0 \leq -x_3 - x_4 - 6), (0 \leq x_5 + x_4 + 1)\}$$

$$B = \{(0 \leq x_2 + x_3 + 3), (0 \leq x_6 - x_5 - 1), (0 \leq x_4 - x_6 + 4)\}$$

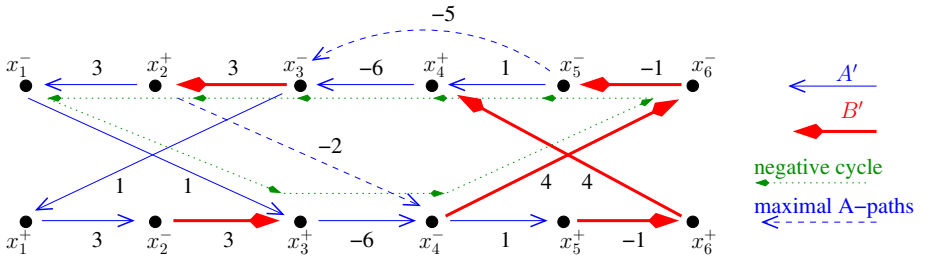


Fig. 2. The constraint graph of Example 1. (We represent only one negative cycle with its corresponding A-paths, because the other is dual.)

By the map of Figure 1, they are converted into the following sets of $\mathcal{DL}(\mathbb{Q})$ constraints:

$$\begin{aligned} A' = \{&(0 \le x_1^- - x_2^+ + 3), (0 \le x_2^- - x_1^+ + 3),\\ &(0 \le x_3^+ - x_1^- + 1), (0 \le x_1^+ - x_3^- + 1),\\ &(0 \le x_4^- - x_3^+ - 6), (0 \le x_3^- - x_4^+ - 6),\\ &(0 \le x_4^+ - x_5^- + 1), (0 \le x_5^+ - x_4^- + 1)\} \end{aligned}$$

$$\begin{aligned} B' = \{&(0 \le x_3^+ - x_2^- + 3), (0 \le x_2^+ - x_3^- + 3),\\ &(0 \le x_6^+ - x_5^+ - 1), (0 \le x_5^- - x_6^- - 1),\\ &(0 \le x_4^+ - x_6^+ + 4), (0 \le x_6^- - x_4^- + 4)\} \end{aligned}$$

whose conjunction corresponds to the constraint graph of Figure 2. This graph has a negative cycle

$$C' \stackrel{\text{def}}{=} x_2^+ \xrightarrow{3} x_1^- \xrightarrow{1} x_3^+ \xrightarrow{-6} x_4^- \xrightarrow{4} x_6^- \xrightarrow{-1} x_5^- \xrightarrow{1} x_4^+ \xrightarrow{-6} x_3^- \xrightarrow{3} x_2^+.$$

Thus, $A \wedge B$ is inconsistent in $\mathcal{UTVPI}(\mathbb{Q})$. From the negative cycle C' we can extract the set of A'-paths $\{x_2^+ \xrightarrow{-2} x_4^-,\ x_5^- \xrightarrow{-5} x_3^-\}$, corresponding to the formula $I' \stackrel{\text{def}}{=} (0 \le x_4^- - x_2^+ - 2) \wedge (0 \le x_3^- - x_5^- - 5)$, which is an interpolant for (A', B'). I' is thus mapped back into $I \stackrel{\text{def}}{=} (0 \le -x_2 - x_4 - 2) \wedge (0 \le x_5 - x_3 - 5)$, which is an interpolant for (A, B).

4 Graph-Based Interpolation for $\mathcal{UTVPI}(\mathbb{Z})$

In order to deal with the more complex case of $\mathcal{UTVPI}(\mathbb{Z})$ (hereafter simply $\mathcal{UTVPI}$), we adopt a layered approach [23]. First, we check the consistency in $\mathcal{UTVPI}(\mathbb{Q})$ using the technique of [20]. If this results in an inconsistency, we compute an $\mathcal{UTVPI}(\mathbb{Q})$-interpolant as described in §3. Clearly, this is also an interpolant in $\mathcal{UTVPI}$: condition (iii) is obvious, and conditions (i) and (ii)

follow immediately from the fact that if an $\mathcal{UTVPI}$-formula is inconsistent over the rationals then it is inconsistent also over the integers.

If the $\mathcal{UTVPI}(\mathbb{Q})$-procedure does not detect an inconsistency, we check the consistency in $\mathcal{UTVPI}$ using the algorithm proposed by Lahiri and Musuvathi in [16], which extends the ideas of [20] to the integer domain. In particular, it gives necessary and sufficient conditions to decide unsatisfiability by detecting particular kinds of zero-weight cycles in the induced $\mathcal{DL}$ constraint graph. This procedure works in $O(n \cdot m)$ time and $O(n+m)$ space, m and n being the number of constraints and variables respectively, which improves the previous $O(n^2 \cdot m)$ time and $O(n^2)$ space complexity of the previous procedure by Jaffar et al. [9].

We build on top of this algorithm and we extend the graph-based approach of §3 for producing interpolants also in $\mathcal{UTVPI}$. In particular, we use the following reformulation of a result of [16].

Theorem 1. *Let ϕ be a conjunction of $\mathcal{UTVPI}$ constraints s.t. ϕ is satisfiable in $\mathcal{UTVPI}(\mathbb{Q})$. Then ϕ is unsatisfiable in $\mathcal{UTVPI}$ iff the constraint graph $G(\phi)$ generated from ϕ has a cycle C of weight 0 containing two vertices x_i^+ and x_i^- s.t. the weight of the path $x_i^- \rightsquigarrow x_i^+$ along C is odd.*

The "only if" part is a corollary of lemmas 1, 2 and 4 in [16]. The "if" comes straightforwardly from the analysis done in [16], whose main intuitions we recall in what follows. Assume the constraint graph $G(\phi)$ generated from ϕ has one cycle C of weight 0 containing two vertices x_i^+ and x_i^- s.t. the weight of the path $x_i^- \rightsquigarrow x_i^+$ along C is $2k+1$ for some integer value k. (Since C has weight 0, the weight of the other path $x_i^+ \rightsquigarrow x_i^-$ along C is $-2k-1$.) Then, the paths $x_i^- \rightsquigarrow x_i^+$ and $x_i^+ \rightsquigarrow x_i^-$ contain at least two constraints, because otherwise their weight would be even (see the last two lines of Figure 1). Then, $x_i^- \rightsquigarrow x_i^+$ is in the form $x_i^- \rightsquigarrow v \xrightarrow{n} x_i^+$, for some v and n. From $x_i^- \rightsquigarrow v$, we can derive the summary constraint $(0 \leq v - x_i^- + (2k+1-n))$, which corresponds to the $\mathcal{UTVPI}$ constraint $(0 \leq \Upsilon(v) + x_i + (2k+1-n))$. (This corresponds to $l-2$ applications of the TRANSITIVE rule of [16], l being the number of constraints in $x_i^- \rightsquigarrow x_i^+$.) Then, by observing that the $\mathcal{UTVPI}$ constraint corresponding to $v \xrightarrow{n} x_i^+$ is $(0 \leq x_i - \Upsilon(v) + n)$, we can apply the TIGHTENING rule of [16] to obtain $(0 \leq x_i + \lfloor (2k+1-n+n)/2 \rfloor)$, which is equivalent to $(0 \leq x_i + k)$. Similarly, from $x_i^+ \rightsquigarrow x_i^-$ we can obtain $(0 \leq -x_i - k - 1)$, and thus an inconsistency using the CONTRADICTION rule of [16].

Consider a pair (A, B) of $\mathcal{UTVPI}$ constraints such that $A \wedge B$ is consistent in $\mathcal{UTVPI}(\mathbb{Q})$ but inconsistent in $\mathcal{UTVPI}$. By Theorem 1, the constraint graph $G(A' \wedge B')$ has a cycle C of weight 0 containing two vertices x_i^+ and x_i^- s.t. the weight of the paths $x_i^- \rightsquigarrow x_i^+$ and $x_i^+ \rightsquigarrow x_i^-$ along C are $2k+1$ and $-2k-1$ respectively, for some value $k \in \mathbb{Z}$. Our algorithm computes an interpolant for (A, B) from the cycle C. Let C_A and C_B be the subsets of the edges in C corresponding to constraints in A' and B' respectively. We have to distinguish four distinct sub-cases.

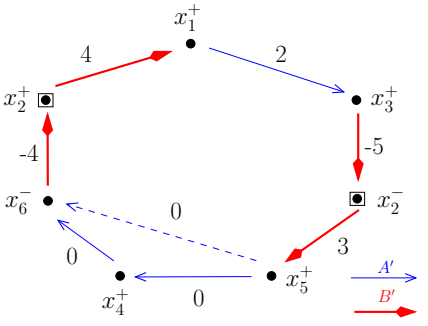


Fig. 3. $\mathcal{UTVPI}$ interpolation, Case 1

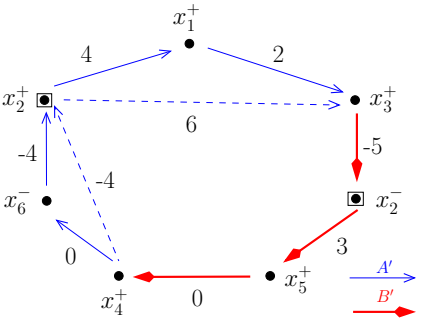


Fig. 4. $\mathcal{UTVPI}$ interpolation, Case 2

Case 1. *x_i occurs in B but not in A.* Consequently, x_i^+ and x_i^- occur in B' but not in A', and hence they occur in C_B but not in C_A. Let I' be the conjunction of the summary constraints of the maximal C_A-paths, and let I be the conjunction of the corresponding $\mathcal{UTVPI}$ constraints. We show that I is an interpolant for (A, B). (i) By construction, $A \models_{\mathcal{UTVPI}} I$, as in §3. (ii) The constraints in I' and C_B form a cycle matching the hypotheses of Theorem 1, from which $I \wedge B$ is $\mathcal{UTVPI}$-inconsistent. (iii) We notice that every variable x_j^+, x_j^- occurring in the conjunction of the summary constraints is an end-point variable, so that $I' \preceq C_A$ and $I' \preceq C_B$, and thus $I \preceq A$ and $I \preceq B$.

Example 2. Consider the following set of constraints:

$$S{=}\{(0 \le x_1 - x_2 + 4), (0 \le -x_2 - x_3 - 5), (0 \le x_2 + x_6 - 4), (0 \le x_5 + x_2 + 3),$$
$$(0 \le -x_1 + x_3 + 2), (0 \le -x_6 - x_4), (0 \le x_4 - x_5)\},$$

partitioned into A and B as follows:

$$A \begin{cases} (0 \le x_3 - x_1 + 2) \\ (0 \le -x_6 - x_4) \\ (0 \le x_4 - x_5) \end{cases} \qquad B \begin{cases} (0 \le x_1 - x_2 + 4) \\ (0 \le -x_2 - x_3 - 5) \\ (0 \le x_2 + x_6 - 4) \\ (0 \le x_5 + x_2 + 3) \end{cases}$$

Figure 3 shows a zero-weight cycle C in $G(A' \wedge B')$ such that the paths $x_2^- \rightsquigarrow x_2^+$ and $x_2^+ \rightsquigarrow x_2^-$ have an odd weight (-1 and 1 resp.) Therefore, by Theorem 1, $A \wedge B$ is $\mathcal{UTVPI}$-inconsistent. The two summary constraints of the maximal C_A paths are $(0 \le x_6^- - x_5^+)$ and $(0 \le x_3^+ - x_1^+ + 2)$. It is easy to see that $I = (0 \le -x_6 - x_5) \wedge (0 \le x_3 - x_1 + 2)$ is an $\mathcal{UTVPI}$-interpolant for (A, B).

Case 2. *x_i occurs in both A and B.* Consequently, x_i^+ and x_i^- occur in both A' and B'. If neither x_i^+ nor x_i^- is such that both the incoming and outgoing edges belong to C_A, then the cycle obtained by replacing each maximal C_A-path with its summary constraint still contains both x_i^+ and x_i^-, so we can apply the same process of Case 1. Otherwise, if both the incoming and outgoing edges of x_i^+

belong to C_A, then we split the maximal C_A-path $u_1 \xrightarrow{c_1} \ldots \xrightarrow{c_k} x_i^+ \xrightarrow{c_{k+1}} \ldots \xrightarrow{c_n} u_n$ containing x_i^+ into the two parts which are separated by x_i^+: $u_1 \xrightarrow{c_1} \ldots \xrightarrow{c_k} x_i^+$ and $x_i^+ \xrightarrow{c_{k+1}} \ldots \xrightarrow{c_n} u_n$. We do the same for x_i^-. Let I' be the conjunction of the resulting summary constraints, and let I be corresponding set of $\mathcal{UTVPI}$ constraints. We show that I is an interpolant for (A, B). (i) As with Case 1, again, $A \models_{\mathcal{UTVPI}} I$. (ii) Since we split the maximal C_A paths as described above, the constraints in I' and C_B form a cycle matching the hypotheses of Theorem 1, from which $I \wedge B$ is $\mathcal{UTVPI}$-inconsistent. (iii) x_i^+, x_i^- occur in both A' and B' by hypothesis, and every other variable x_j^+, x_j^- occurring in the conjunction of the summary constraints is an end-point variable, so that $I' \preceq C_A$ and $I' \preceq C_B$, and thus $I \preceq A$ and $I \preceq B$.

Example 3. Consider again the set of constraints S of Example 2, partitioned into A and B as follows:

$$A \begin{cases} (0 \leq x_3 - x_1 + 2) \\ (0 \leq -x_6 - x_4) \\ (0 \leq x_2 + x_6 - 4) \\ (0 \leq x_1 - x_2 + 4) \end{cases} \qquad B \begin{cases} (0 \leq -x_2 - x_3 - 5) \\ (0 \leq x_5 + x_2 + 3) \\ (0 \leq x_4 - x_5) \end{cases}$$

and the zero-weight cycle C of $G(A' \wedge B')$ shown in Figure 4. As in the previous example, there is a path $x_2^- \rightsquigarrow x_2^+$ of weight -1 and a path $x_2^+ \rightsquigarrow x_2^-$ of weight 1. In this case there is only one maximal C_A path, namely $x_4^+ \rightsquigarrow x_3^+$. Since the cycle obtained by replacing it with its summary constraint $(0 \leq x_3^+ - x_4^+ + 2)$ does not contain x_2^+, we split $x_4^+ \rightsquigarrow x_3^+$ into two paths, $x_4^+ \rightsquigarrow x_2^+$ and $x_2^+ \rightsquigarrow x_3^+$, whose summary constraints are $(0 \leq x_2^+ - x_4^+ - 4)$ and $(0 \leq x_3^+ - x_2^+ + 6)$ respectively. By replacing the two paths above with the two summary constraints, we get a zero-weight cycle which still contains the two odd paths $x_2^- \rightsquigarrow x_2^+$ and $x_2^+ \rightsquigarrow x_2^-$. Therefore, $I \stackrel{\text{def}}{=} (0 \leq x_2 - x_4 - 4) \wedge (0 \leq x_3 - x_2 + 6)$ is an interpolant for (A, B).

Notice that the $\mathcal{UTVPI}$-formula $J \stackrel{\text{def}}{=} (0 \leq x_3 - x_4 + 2)$ corresponding to the summary constraint of the maximal C_A path $x_4^+ \rightsquigarrow x_3^+$ is not an interpolant, since $J \wedge B$ is not $\mathcal{UTVPI}$-inconsistent. In fact, if we replace the maximal C_A path $x_4^+ \rightsquigarrow x_3^+$ with the summary constraint $x_4^+ \xrightarrow{2} x_3^+$, the cycle we obtain has still weight zero, but it contains no odd path between two variables x_i^+ and x_i^-.

Case 3. *x_i occurs in A but not in B, and one of the paths $x_i^+ \rightsquigarrow x_i^-$ or $x_i^- \rightsquigarrow x_i^+$ in C contains only constraints of C_A.* In this case, x_i^+ and x_i^- occur in A' but not in B'. Suppose that $x_i^- \rightsquigarrow x_i^+$ consists only of constraints of C_A (the case $x_i^+ \rightsquigarrow x_i^-$ is analogous). Let $2k+1$ be the weight of the path $x_i^- \rightsquigarrow x_i^+$ (which is odd by hypothesis), and let $\overline{C}$ be the cycle obtained by replacing such path with the edge $x_i^- \xrightarrow{2k} x_i^+$ in C. In the following, we call such a replacement *tightening summarization*. Since C has weight zero, $\overline{C}$ has negative weight. Let C^P be the set of $\mathcal{DL}$-constraints in the path $x_i^- \rightsquigarrow x_i^+$. Let I' be the $\mathcal{DL}$-interpolant computed from $\overline{C}$ for $(C_A \setminus C^P \cup \{(0 \leq x_i^+ - x_i^- + 2k)\}, C_B)$, and let I be the corresponding $\mathcal{UTVPI}$ formula. We show that I is an interpolant for (A, B).

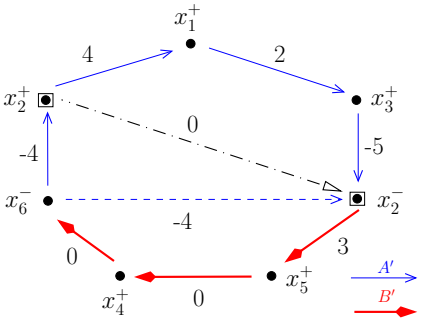


Fig. 5. $\mathcal{UTVPI}$ interpolation, Case 3

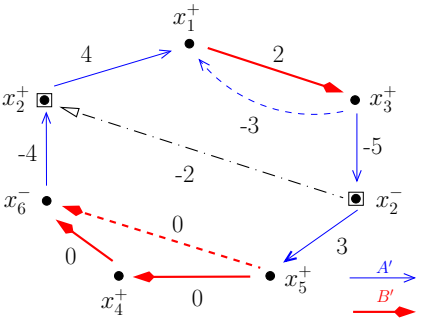


Fig. 6. $\mathcal{UTVPI}$ interpolation, Case 4

(i) Let P be the set of $\mathcal{UTVPI}$ constraints in the path $x_i^- \rightsquigarrow x_i^+$. Since the weight $2k+1$ of such path is odd, we have that $P \models_{\mathcal{UTVPI}} (0 \leq x_i + k)$ (cf. page 174). Since $P \subseteq A$, therefore, $A \models_{\mathcal{UTVPI}} (0 \leq x_i + k)$. By observing that $(0 \leq x_i^+ - x_i^- + 2k)$ is the $\mathcal{DL}$-constraint corresponding to $(0 \leq x_i + k)$ we conclude that $C_A \setminus C^P \cup (0 \leq x_i^+ - x_i^- + 2k) \models_{\mathcal{DL}} I'$ implies that $A \setminus P \cup (0 \leq x_i + k) \models_{\mathcal{UTVPI}} I$, and so that $A \models_{\mathcal{UTVPI}} I$.

(ii) Since all the constraints in C_B occur in $\overline{C}$, we have that $B \wedge I$ is $\mathcal{UTVPI}$-inconsistent.

(iii) Since by hypothesis all the constraints in the path $x_i^- \rightsquigarrow x_i^+$ occur in C_A, from $I' \preceq (C_A \setminus C^P \cup \{(0 \leq x_i^+ - x_i^- + 2k)\})$ we have that $I \preceq A$. Finally, since all the constraints in C_B occur in $\overline{C}$, we have that $I \preceq B$.

Example 4. Consider again the set S of constraints of Example 2, this time partitioned into A and B as follows:

$$A \begin{cases} (0 \leq x_1 - x_2 + 4) \\ (0 \leq x_3 - x_1 + 2) \\ (0 \leq -x_2 - x_3 - 5) \\ (0 \leq x_2 + x_6 - 4) \end{cases} \qquad B \begin{cases} (0 \leq x_5 + x_2 + 3) \\ (0 \leq -x_6 - x_4) \\ (0 \leq x_4 - x_5) \end{cases}$$

Figure 5 shows a zero-weight cycle C of $G(A' \wedge B')$. The only maximal C_A path is $x_6^- \rightsquigarrow x_2^-$. Since the path $x_2^+ \rightsquigarrow x_2^-$ has weight 1, we can add the tightening edge $x_2^+ \xrightarrow{1-1} x_2^-$ to $G(A' \wedge B')$ (shown in dots and dashes in Figure 5), corresponding to the constraint $(0 \leq x_2^- - x_2^+)$. Since all constraints in the path $x_2^+ \rightsquigarrow x_2^-$ belong to A', $A' \models (0 \leq x_2^- - x_2^+)$. Moreover, the cycle obtained by replacing the path $x_2^+ \rightsquigarrow x_2^-$ with the tightening edge $x_2^+ \xrightarrow{0} x_2^-$ has a negative weight (-1). Therefore, we can generate a $\mathcal{DL}$-interpolant $I' \stackrel{\text{def}}{=} (0 \leq x_2^- - x_6^- - 4)$ from such cycle, which corresponds to the $\mathcal{UTVPI}$-interpolant $I \stackrel{\text{def}}{=} (0 \leq -x_2 + x_6 - 4)$.

Notice that, similarly to Example 3, also in this case we cannot obtain an interpolant from the summary constraint $(0 \leq x_2^- - x_6^- - 3)$ of the maximal C_A path $x_6^- \rightsquigarrow x_2^-$, as $(0 \leq -x_2 + x_6 - 3) \wedge B$ is not $\mathcal{UTVPI}$-inconsistent.

Case 4: *x_i occurs in A but not in B, and neither the path $x_i^+ \rightsquigarrow x_i^-$ nor the path $x_i^- \rightsquigarrow x_i^+$ in C consists only of constraints of C_A.* As in the previous case, x_i^+ and x_i^- occur in A' but not in B', and hence they occur in C_A but not in C_B. In this case, however, we can apply a tightening summarization neither to $x_i^+ \rightsquigarrow x_i^-$ nor to $x_i^- \rightsquigarrow x_i^+$, since none of the two paths consists only of constraints of C_A. We can, however, perform a *conditional tightening summarization* as follows. Let C_A^P and C_B^P be the sets of constraints of C_A and C_B respectively occurring in the path $x_i^- \rightsquigarrow x_i^+$, and let $\overline{C}_A^P$ and $\overline{C}_B^P$ be the sets of summary constraints of maximal paths in C_A^P and C_B^P. From $\overline{C}_A^P \cup \overline{C}_B^P$, we can derive $x_i^- \xrightarrow{2k} x_i^+$ (cf. Case 3), where $2k+1$ is the weight of the path $x_i^- \rightsquigarrow x_i^+$. Therefore, $\overline{C}_A^P \cup \overline{C}_B^P \models (0 \leq x_i^+ - x_i^- + 2k)$, and thus $\overline{C}_A^P \models \overline{C}_B^P \rightarrow (0 \leq x_i^+ - x_i^- + 2k)$. We say that $(0 \leq x_i^+ - x_i^- + 2k)$ is the summary constraint for $x_i^- \rightsquigarrow x_i^+$ *conditioned to* $\overline{C}_B^P$.

Using conditional tightening summarization, we generate an interpolant as follows. By replacing the path $x_i^- \rightsquigarrow x_i^+$ with $x_i^- \xrightarrow{2k} x_i^+$, we obtain a negative-weight cycle $\overline{C}$, as in Case 3. Let I' be the $\mathcal{DL}$-interpolant computed from $\overline{C}$ for $(C_A \setminus C_A^P \cup \{(0 \leq x_i^+ - x_i^- + 2k)\}, C_B \setminus C_B^P)$, and let I be the corresponding $\mathcal{UTVPI}$ formula. Finally, let $\overline{P}_B$ be the conjunction of $\mathcal{UTVPI}$ constraints corresponding to $\overline{C}_B^P$. We show that $(\overline{P}_B \rightarrow I)$ is an interpolant for (A, B).

(i) We know that $C_A \setminus C_A^P \cup \{(0 \leq x_i^+ - x_i^- + 2k)\} \models I'$, because I' is a $\mathcal{DL}$-interpolant. Moreover, $\overline{C}_A^P \cup \overline{C}_B^P \models (0 \leq x_i^+ - x_i^- + 2k)$, and so $C_A^P \cup \overline{C}_B^P \models (0 \leq x_i^+ - x_i^- + 2k)$. Therefore, $C_A \cup \overline{C}_B^P \models I'$, and thus $A \cup \overline{P}_B \models_{\mathcal{UTVPI}} I$, from which $A \models_{\mathcal{UTVPI}} (\overline{P}_B \rightarrow I)$.

(ii) Since I' is a $\mathcal{DL}$-interpolant for $(C_A \setminus C_A^P \cup \{(0 \leq x_i^+ - x_i^- + 2k)\}, C_B \setminus C_B^P)$, $I' \wedge (C_B \setminus C_B^P)$ is $\mathcal{DL}$-inconsistent, and thus $I \wedge B$ is $\mathcal{UTVPI}$-inconsistent. Since by construction $B \models_{\mathcal{UTVPI}} \overline{P}_B$, $(\overline{P}_B \rightarrow I) \wedge B$ is $\mathcal{UTVPI}$-inconsistent.

(iii) From $I' \preceq C_B \setminus C_B^P$ we have that $I \preceq B$, and from $I' \preceq C_A \setminus C_A^P \cup \{(0 \leq x_i^+ - x_i^- + 2k)\}$ that $I \preceq A$. Moreover, all the variables occurring in the constraints in $\overline{C}_B^P$ are end-point variables, so that $\overline{C}_B^P \preceq C_A$ and $\overline{C}_B^P \preceq C_B$, and thus $\overline{P}_B \preceq A$ and $\overline{P}_B \preceq B$. Therefore, $(\overline{P}_B \rightarrow I) \preceq A$ and $(\overline{P}_B \rightarrow I) \preceq B$.

Example 5. We partition the set S of constraints of Example 2 into A and B as follows:

$$A \begin{cases} (0 \leq x_1 - x_2 + 4) \\ (0 \leq -x_2 - x_3 - 5) \\ (0 \leq x_5 + x_2 + 3) \\ (0 \leq x_2 + x_6 - 4) \end{cases} \qquad B \begin{cases} (0 \leq x_3 - x_1 + 2) \\ (0 \leq -x_6 - x_4) \\ (0 \leq x_4 - x_5) \end{cases}$$

Consider the zero-weight cycle C of $G(A' \wedge B')$ shown in Figure 6. In this case, neither the path $x_2^+ \rightsquigarrow x_2^-$ nor the path $x_2^- \rightsquigarrow x_2^+$ consists only of constraints of A', and thus we cannot use any of the two tightening edges $x_2^+ \xrightarrow{1-1} x_2^-$ and $x_2^- \xrightarrow{-1-1} x_2^+$ *directly* for computing an interpolant. However, we can compute the summary $x_2^- \xrightarrow{-2} x_2^+$ for $x_2^- \rightsquigarrow x_2^+$ *conditioned to* $x_5^+ \xrightarrow{0} x_6^-$, which is the summary constraint of the B-path $x_5^+ \rightsquigarrow x_6^-$, and whose corresponding $\mathcal{UTVPI}$

constraint is $(0 \leq -x_6 - x_5)$. By replacing the path $x_2^- \rightsquigarrow x_2^+$ with such summary, we obtain a negative-weight cycle $\overline{C}$, from which we generate the $\mathcal{DL}$-interpolant $(0 \leq x_1^+ - x_3^+ - 3)$, corresponding to the $\mathcal{UTVPI}$ formula $(0 \leq x_1 - x_3 - 3)$. Therefore, the generated $\mathcal{UTVPI}$-interpolant is $(0 \leq -x_6 - x_5) \rightarrow (0 \leq x_1 - x_3 - 3)$.

As in Example 4, notice that we cannot generate an interpolant from the conjunction of summary constraints of maximal C_A paths, since the formula we obtain (i.e. $(0 \leq x_1 + x_6) \wedge (0 \leq x_5 - x_3 - 2)$) is not inconsistent with B.

5 Experimental Evaluation

We have implemented the algorithm described in the previous sections within our SMT solver MathSAT [4]. The assessment of the procedure can not be carried out by means of a direct comparison, since there exists no other system able to interpolate over $\mathcal{UTVPI}(\mathbb{Z})$. In order to assess both the efficiency and the usefulness of the procedure, we have performed two kinds of experiments, one on interpolation over $\mathcal{UTVPI}(\mathbb{Q})$, and one on the application of interpolation for $\mathcal{UTVPI}(\mathbb{Z})$ to software model checking.

The programs and benchmark instances used are available at `http://disi.unitn.it/~griggio/papers/cade09_itp_utvpi.tar.gz`. All the tests have been performed on 2.66 GHz Intel Xeon machines, with 16 GB of RAM and 6 MB of cache, running Linux. For each instance, we used a time limit of 20 minutes and a memory limit of 2 GB.

Comparison with $\mathcal{LA}(\mathbb{Q})$ Interpolation. In the first part of our experiments, we compare our novel $\mathcal{UTVPI}(\mathbb{Q})$ interpolation algorithm with our implementation of a state-of-the-art $\mathcal{LA}(\mathbb{Q})$ interpolation algorithm [5], in order to evaluate its efficiency. Both algorithms are implemented within MathSAT, and thus they share the same environment (same DPLL engine, same search strategy, same optimizations, etc.). This ensures that the comparison is fair.

We have randomly generated several $\mathcal{UTVPI}(\mathbb{Q})$ interpolation problems of varying size and difficulty, and run both algorithms. The results are collected in Figure 7. The scatter plots show that the $\mathcal{UTVPI}(\mathbb{Q})$ solver clearly outperforms the $\mathcal{LA}(\mathbb{Q})$ solver (sometimes by more than an order of magnitude), thus justifying the interest for the subclass. Furthermore, it can be seen that the computed interpolants, in addition to being within $\mathcal{UTVPI}(\mathbb{Q})$, are generally smaller, both in terms of nodes in the formula DAG and in number of atoms.

$\mathcal{UTVPI}$ Interpolation in Software Model Checking. In the second part of our experiments, we evaluate the usefulness of the $\mathcal{UTVPI}$ interpolation procedure in the context of software model checking based on the counterexample-guided abstraction refinement (CEGAR) paradigm. This is one of the most successful applications of interpolation in formal verification [8]: in this setting, interpolants are used to automatically refine abstractions when *spurious* error traces are generated. A spurious error trace is an execution of the abstract program that leads to an error, but does not correspond to any execution of the

concrete program. The interpolation procedure receives as input formulas corresponding to such spurious error traces, which are typically conjunctions of literals, and the interpolants generated are used to prevent the same spurious error trace from being generated again in the future. This technique is used for example by the software model checker BLAST, whose description in [2] we refer to for the details.

Due to the limitations of current interpolation procedures,[3] when computing interpolants program variables are interpreted over the rationals even if in the program they have an integral type. This makes it impossible to compute interpolants for spurious error traces which are inconsistent because of the violation of the integrality constraints on the variables. When this happens, automatic abstraction refinement cannot be performed, and the verification of the program fails.

We have written a collection of small C programs which cannot be verified by BLAST because the interpolation procedures that it uses ([3,18,22]) do not handle integrality constraints. When using MATHSAT as interpolation procedure instead, BLAST could successfully verify all the programs.

Example 6. Consider the simple C program on the right.[4] In the process of proving that the `ERROR` label is not reachable, BLAST generates the following spurious counterexample:

```
main() {
  int x, y, z;
  while (*) {
    y = *;
    x = y;
    z = 1 - y;
    if (x == z) {
      ERROR: ;
    }
  }
}
```

$$\mathtt{y} = *;\ \mathtt{x} = \mathtt{y};\ \mathtt{z} = 1 - \mathtt{y};\ (\mathtt{x} == \mathtt{z});\ \mathtt{ERROR},$$

which corresponds to the following formula ξ:

$$\xi \stackrel{\text{def}}{=} (x = y) \wedge (z = 1 - y) \wedge (x = z)$$

In order to refine the abstraction, BLAST asks the interpolation procedure to compute an interpolant for the following partition of ξ into (A, B):

$$\underbrace{(x = y) \wedge (z = 1 - y)}_{A} \wedge \underbrace{(x = z)}_{B}.$$

ξ is unsatisfiable in $\mathbb{Z}$, but satisfiable in $\mathbb{Q}$. Therefore, interpolation procedures that work on the rationals are not able to compute the interpolant, causing BLAST to fail. Using the $\mathcal{UTVPI}$ interpolation algorithm, instead, MATHSAT computes the following interpolant:[5]

$$I \stackrel{\text{def}}{=} (0 \leq x + z - 1) \wedge (0 \leq -x - z + 1),$$

with which BLAST can refine the abstraction and successfully prove that the `ERROR` label is not reachable.

[3] With the exception of [10], which however is limited to equations and modular equations, and cannot handle inequalities.

[4] A '$*$' indicates a nondeterministic value.

[5] After rewriting equalities $(x_1 = x_2 + c)$ into $(0 \leq x_1 - x_2 + c) \wedge (0 \leq x_2 - x_1 - c)$.

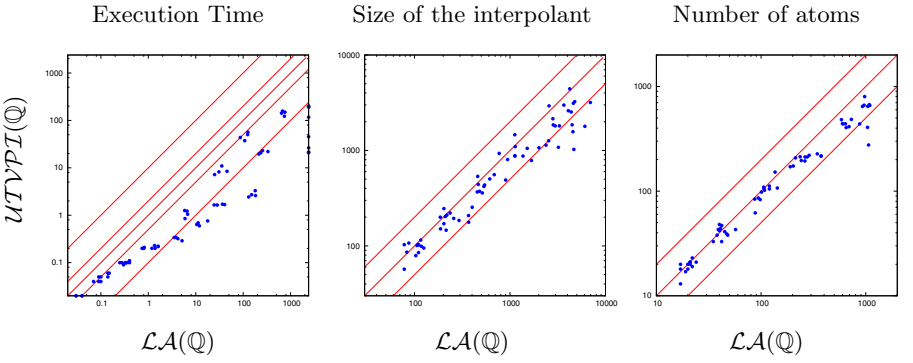


Fig. 7. Comparison between $\mathcal{UTVPI}(\mathbb{Q})$ and $\mathcal{LA}(\mathbb{Q})$ interpolation within MathSAT

6 Conclusions

In this paper we have tackled the problem of generating interpolants in SMT for the theory of Unit-Two-Variable-Per-Inequality ($\mathcal{UTVPI}$), an important fragment of linear arithmetic. Our approach results in interpolants that are within the same theory, and it can be easily implemented on top of efficient graph-based procedures used in many state-of-the-art SMT solvers. Our work covers both the case of rationals, where we experimentally demonstrate the efficiency over a general purpose procedure for $\mathcal{LA}(\mathbb{Q})$, as well as the case of integers, up to now an open problem. In the future, we plan to tackle the full-blown case of interpolation for $\mathcal{LA}(\mathbb{Z})$, where the $\mathcal{UTVPI}$ procedure can be used as a component in a layered approach [23], and to apply SMT-based interpolation procedures in various verification settings, including counterexample guided abstraction refinement.

References

1. Ball, T., Cook, B., Lahiri, S.K., Zhang, L.: Zapato: Automatic Theorem Proving for Predicate Abstraction Refinement. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 457–461. Springer, Heidelberg (2004)
2. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast: Applications to software engineering. STTT 9(5-6), 505–525 (2007)
3. Beyer, D., Zufferey, D., Majumdar, R.: CSIsat: Interpolation for LA+EUF. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 304–308. Springer, Heidelberg (2008)
4. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MathSAT 4 SMT Solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 299–303. Springer, Heidelberg (2008)
5. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient Interpolant Generation in Satisfiability Modulo Theories. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 397–412. Springer, Heidelberg (2008)
6. Cotton, S., Maler, O.: Fast and Flexible Difference Constraint Propagation for DPLL(T). In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 170–183. Springer, Heidelberg (2006)

7. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. In: PLDI (2002)
8. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL. ACM, New York (2004)
9. Jaffar, J., Maher, M.J., Stuckey, P.J., Yap, R.H.C.: Beyond Finite Domains. In: Borning, A. (ed.) PPCP 1994. LNCS, vol. 874. Springer, Heidelberg (1994)
10. Jain, H., Clarke, E.M., Grumberg, O.: Efficient Craig Interpolation for Linear Diophantine (Dis)Equations and Linear Modular Equations. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 254–267. Springer, Heidelberg (2008)
11. Jhala, R., McMillan, K.L.: A Practical and Complete Approach to Predicate Refinement. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
12. Kapur, D., Majumdar, R., Zarba, C.G.: Interpolation for data structures. In: Young, M., Devanbu, P.T. (eds.) SIGSOFT FSE. ACM, New York (2006)
13. Kroening, D., Weissenbacher, G.: Lifting Propositional Interpolants to the Word-Level. In: FMCAD, pp. 85–89. IEEE Computer Society, Los Alamitos (2007)
14. Krstić, S., Fuchs, A., Goel, A., Grundy, J., Tinelli, C.: Ground Interpolation for the Theory of Equality. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 413–427. Springer, Heidelberg (2009)
15. Lahiri, S.K., Bryant, R.E.: Deductive Verification of Advanced Out-of-Order Microprocessors. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 341–354. Springer, Heidelberg (2003)
16. Lahiri, S.K., Musuvathi, M.: An Efficient Decision Procedure for UTVPI Constraints. In: Gramlich, B. (ed.) FroCos 2005. LNCS, vol. 3717, pp. 168–183. Springer, Heidelberg (2005)
17. McMillan, K.: Interpolation and SAT-based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
18. McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. 345(1) (2005)
19. McMillan, K.L.: Lazy Abstraction with Interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
20. Miné, A.: The Octagon Abstract Domain. In: Proc. of WCRE, Washington, DC, USA, pp. 310–319. IEEE Computer Society, Los Alamitos (2001)
21. Nieuwenhuis, R., Oliveras, A.: DPLL(T) with Exhaustive Theory Propagation and Its Application to Difference Logic. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 321–334. Springer, Heidelberg (2005)
22. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint Solving for Interpolation. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 346–362. Springer, Heidelberg (2007)
23. Sebastiani, R.: Lazy Satisfiability Modulo Theories. Journal on Satisfiability, Boolean Modeling and Computation, JSAT 3 (2007)
24. Seshia, S.A., Bryant, R.E.: Deciding Quantifier-Free Presburger Formulas Using Parameterized Solution Bounds. In: LICS. IEEE Computer Society, Los Alamitos (2004)
25. Sofronie-Stokkermans, V.: Interpolation in Local Theory Extensions. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS, vol. 4130, pp. 235–250. Springer, Heidelberg (2006)
26. Yorsh, G., Musuvathi, M.: A combination method for generating interpolants. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS, vol. 3632, pp. 353–368. Springer, Heidelberg (2005)

Ground Interpolation for Combined Theories

Amit Goel[1], Sava Krstić[1], and Cesare Tinelli[2]

[1] Strategic CAD Labs, Intel Corporation
[2] Department of Computer Science, The University of Iowa

Abstract. We give a method for modular generation of ground interpolants in modern SMT solvers supporting multiple theories. Our method uses a novel algorithm to modify the proof tree obtained from an unsatifiability run of the solver into a proof tree without occurrences of troublesome "uncolorable" literals. An interpolant can then be readily generated using existing procedures. The principal advantage of our method is that it places few restrictions (none for convex theories) on the search strategy of the solver. Consequently, it is straightforward to implement and enables more efficient interpolating SMT solvers. In the presence of non-convex theories our method is incomplete, but still more general than previous methods.

1 Introduction

Given mutually inconsistent formulas F and G in some logic, an interpolant I is a formula such that: *(i)* $F \models I$; *(ii)* $G, I \models \mathsf{false}$; and *(iii)* the non-logical symbols in I occur in both F and G. In [13], McMillan presented an algorithm for propositional interpolation and described a complete procedure for model-checking finite-state systems. In this method, interpolants are used to derive property-driven overapproximations of reachable state sets from unsatisfiable symbolic traces. This technique has proven to be efficient in practice, and the recipe in [13] is used as a starting point in many finite-state model checkers.

A natural desire is to extend interpolation-based methods to decidable fragments of richer logics, for use in applications such as software model checking. Most solvers for *satisfiability modulo theories* (SMT) employ a propositional SAT solver in cooperation with theory-specific decision procedures (*theory solvers*) to solve queries in the combined language. The promise of interpolating SMT solvers has been demonstrated by the use of FOCI [14] for model-checking C programs [10,15]. However, their development has not been nearly as widespread as for the propositional case; we know of only two other interpolating SMT solvers [6,3].

The quick adoption of propositional interpolation is in large part due to the simplicity of the propositional interpolation algorithm. It requires a SAT solver enhanced only with the capability to produce a resolution refutation for unsatisfiable formulas. The interpolant is computed by a simple recursive function on resolution proofs. The published solutions for SMT interpolation, on the other hand, either describe an ad hoc solver for a specific collection of theories, or require significant modifications in more general SMT solvers to limit them sufficiently for the described method to work. In this paper we present a simple

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 183–198, 2009.

algorithm for interpolant generation from refutations produced by SMT solvers, while placing minimal restrictions on the solvers' search strategy.

Related Work. In the seminal work [14], McMillan produced a proof system for the ground theory of linear arithmetic with uninterpreted functions and showed how to generate interpolants from such proofs. Yorsh and Musuvathi [18] extended the approach to general combinations of theories that are individually interpolant-generating. These authors were the first to isolate the important requirement that the theories be *equality-interpolating*: if a theory solver can derive $x{=}y$ from $F \wedge G$, where x occurs only in F and y occurs only in G ("uncolorable equality"), then it must be able to derive $x{=}t$ and $t{=}y$ for some term t in the language common to F and G. There are two shortcomings to their approach. Firstly, it requires the generation and propagation of equality-interpolating terms on the fly, thus imposing an overhead during the search procedure of the SMT solver. Secondly, it requires theory solvers to be equality-propagating. As noted in [8], equality propagation can take the majority of time in some decision procedures for little gain. Indeed, modern SMT solvers let the SAT solver split on equalities and either forgo equality propagation completely (*delayed theory combination* (DTC) [4]) or use it sparingly (*model-based theory combination* [7]).

The MathSAT [6] and CSIsat [3] tools avoid the problem of on-the-fly creation of equality interpolants; they create only those equality interpolants that are needed for a series of local proof transformations that modify refutations produced by their solvers into the form suitable for deriving interpolants. The authors of [6] identify the class of *ie-local* refutations which are amenable to such transformations. However, the search strategy in both tools is restricted. CSIsat requires equality-propagating decision procedures, while MathSAT simulates equality propagation with heuristics to restrict delayed theory combination.

The interpolation algorithms in all these methods and ours rely on theory-specific interpolation procedures such as those in [14,17,6,9].

Contributions. We define *almost-colorable* refutations and present a two-phase algorithm for the generation of interpolants from such refutations. In the first phase, the almost-colorable refutation is transformed into a *colorable* refutation. The interpolant is then derived from the colorable refutation in the second phase.

There are several advantages to our approach. The class of almost-colorable refutations is more general than the class of ie-local refutations. We show that for the case of convex theories, any search strategy for an SMT solver will produce almost-colorable refutations as long as the theory solvers satisfy the reasonable requirement of not generating lemmas with redundant equalities. In the more general case with non-convex theories, we require the SAT solver not to split on uncolorable equalities. This compromises the completeness of the SMT solver, but enables us to interpolate for a larger set of formulas than [18] since we do allow splitting on colorable equalities. We also show that for a subset of almost-colorable refutations (including ie-local ones), our colorability algorithm

produces refutations whose size, measured by the number of nodes in a tree representation, is at most twice the size of the input refutation.

Outline. In §2, we review and define the necessary material, including the concepts of proof trees modulo a given collection of theories and a given set S of input clauses. We also define colorability of proof trees with respect to a partition $S = A \cup B$ of the input clause set and recall the algorithm that produces an interpolant for A, B from a given colorable proof tree. In §3, we define the class $\mathcal{P}(A, B)$ of almost-colorable proof trees and prove in Theorem 2 that each proof tree from this class can be transformed into a colorable one. We also give a detailed description of the coloring transformation algorithm. In §4, we define $\text{NODPLL}^{\text{pf}}$, a transition system for abstractly describing modern SMT solvers, and prove in Theorem 4 that it produces almost-colorable proof trees when the theories are convex, or if splitting on uncolorable equalities is disallowed.

2 Preliminaries

2.1 Syntax

We will use the standard terminology. A *signature* is a set of *function symbols* plus a set of *predicate symbols*. *Terms* are built using *variables* and *free constants* by recursive application of function symbols. *Atoms* are applications of predicate symbols to terms. Atoms and their negations are *literals*. A *(quantifier-free) formula* is a boolean combination of atoms. A term or a formula is *ground* if it has no occurrences of variables. See [2] for more details.

A *(ground) clause* is a set of ground literals. Clause γ is a *resolvent* of clauses α and β if there is an atom p such that $\alpha = \alpha' \uplus p$, $\beta = \beta' \uplus \neg p$ and $\gamma = \alpha' \cup \beta'$. We call p the *atom resolved upon*. We say that γ is a *merge* [1] of any common literal in α' and β'. We use $\uplus$ to denote disjoint union and, to avoid clutter, we write l for the singleton $\{l\}$. We will not distinguish between the clause $\{l_1, \ldots, l_n\}$ and the disjunction $l_1 \vee \cdots \vee l_n$.

2.2 Resolution Proof Trees

A *tree* is a finite directed graph with a *root* node that is reachable from every other node, and every other node has exactly one outgoing edge. *Leaves* are nodes with no incoming edges. In a *binary tree* every *internal* (i.e. non-leaf) node n has exactly two incoming edges connecting n with its *parents*.

A *resolution proof tree* (or just *proof tree*) is a binary tree together with a mapping that associates with each node n a ground clause $[\![n]\!]$ so that the clause at each internal node of the tree is a resolvent of the clauses of the node's parents. The atom resolved upon at the node n is called the *pivot at* n. If P is a proof tree, we will write $[\![P]\!]$ for the clause associated with the root of P. A *refutation* is any proof tree P such that $[\![P]\!]$ is the empty clause.

We will write $P = \langle P_1, l, P_2 \rangle$ when P_1 and P_2 are the subtrees of P rooted at the parent nodes of the root of P, l is the literal resolved upon at the root of P, and $l \in [\![P_1]\!]$, $\neg l \in [\![P_2]\!]$. Note that $\langle P_1, l, P_2 \rangle$ and $\langle P_2, \neg l, P_1 \rangle$ represent the same

proof tree. When using $\langle P_1, l, P_2 \rangle$ to define P_1, P_2 we will assume, without loss of generality, that l is an atom.

Lemma 1. *If* $P = \langle P_1, l, P_2 \rangle$, *then* $[\![P_1]\!] \subseteq [\![P]\!] \cup l$ *and* $[\![P_2]\!] \subseteq [\![P]\!] \cup \neg l$.

2.3 Theories

A signature Σ defines the class of Σ-*models*. A Σ-*theory* is a set $\mathcal{T}$ of Σ-models. A ground Σ-formula ϕ is $\mathcal{T}$-*satisfiable* if there is a model of $\mathcal{T}$ and an assignment of elements of the model to free constants that make ϕ true. We write $S \models_{\mathcal{T}} \phi$ when ϕ is true in all $\mathcal{T}$-models that satisfy each formula in the set S, for all assignments to free constants (and abbreviate $\varnothing \models_{\mathcal{T}} \phi$ with $\models_{\mathcal{T}} \phi$). If $\Sigma_1, \ldots, \Sigma_n$ are disjoint signatures, and $\mathcal{T}_i$ is a Σ_i-theory $(i = 1, \ldots, n)$, then there is a well-defined $(\Sigma_1 + \cdots + \Sigma_n)$-theory $\mathcal{T}_1 + \cdots + \mathcal{T}_n$. For more details, see [2].

Let S be a finite set of *input clauses* and $\mathcal{T}_1 + \cdots + \mathcal{T}_n$ be a fixed disjoint union of theories. A clause γ such that $\models_{\mathcal{T}_i} \gamma$ is called a *theory lemma*, or a $\mathcal{T}_i$-lemma to be specific. We define a $(\mathcal{T}_1, \ldots, \mathcal{T}_n)$-*proof tree from* S to be any proof tree in which the clause $[\![n]\!]$ for every leaf n is either an input clause or a theory lemma. It is straightforward to show that $S \models_{\mathcal{T}_1 + \cdots + \mathcal{T}_n} [\![P]\!]$, if P is a $(\mathcal{T}_1, \ldots, \mathcal{T}_n)$-proof tree from S.

When the input set of clauses is given as a union $S = A \cup B$, we will use the following coloring terminology. A term or literal will be called A-*colorable* if all non-logical symbols that occur in it also occur in A. We define B-*colorable* similarly. A term or literal that is both A- and B-colorable will be called AB-*colored*. A term or literal that is A-colorable (resp. B-colorable) but not AB-colored is A-*colored* (resp. B-*colored*). A term or literal is *colorable* if it is A- or B-colorable, and is *uncolorable* otherwise. A clause is colorable if every literal occurring in it is colorable. Define the splitting $\gamma = \gamma_{\setminus B} \uplus \gamma_{\downarrow B}$ of any colorable clause γ into subclauses $\gamma_{\setminus B}$ and $\gamma_{\downarrow B}$ consisting of A-colored and B-colorable literals in γ respectively. A $(\mathcal{T}_1, \ldots, \mathcal{T}_n)$-proof tree from $A \cup B$ is colorable if every literal occurring in it is colorable. A node in a proof tree is *critical* if it is an internal node and its pivot is uncolorable.

A theory $\mathcal{T}$ is *ground interpolating* if for every pair of sets A, B of ground clauses such that $A, B \models_{\mathcal{T}} \mathsf{false}$, there exists an AB-colored ground formula ϕ (a *ground* $\mathcal{T}$-*interpolant* for A, B) such that $A \models_{\mathcal{T}} \phi$ and $B, \phi \models_{\mathcal{T}} \mathsf{false}$. A computable function $\mathsf{itp}_{\mathcal{T}}(A, B)$ that computes a ground $\mathcal{T}$-interpolant for any given input sets A and B of *literals*[1] will be called a *ground interpolation procedure* for $\mathcal{T}$. Such a procedure can be extended to a procedure that computes ground interpolants for arbitrary sets A and B of ground clauses (not just sets of literals); see [14,6] and the special case $n = 1$ of Theorem 1 below.

A theory $\mathcal{T}$ is *equality interpolating* [18] if for every $\mathcal{T}$-lemma $\gamma \uplus x{=}y$ such that γ is colorable and $x{=}y$ is uncolorable, there exists an AB-colored term z such that $\models_{\mathcal{T}} \gamma \cup x{=}z$ and $\models_{\mathcal{T}} \gamma \cup z{=}y$. The term z is called an *equality interpolant* for the clause $\gamma \uplus x{=}y$. It is shown in [18] that not all theories are equality interpolating, but the commonly used ones are.

[1] More precisely, A and B are sets of *one-literal clauses*.

2.4 Deriving Interpolants from Colorable Proof Trees

It is possible to produce a ground interpolant for A, B from any colorable $(\mathcal{T}_1, \ldots, \mathcal{T}_n)$-refutation P from $A \cup B$, if each $\mathcal{T}_i$ has a ground interpolation procedure, itp_i. Define I_P by:

$$I_P = \begin{cases} \mathsf{itp}_i(\neg[\![P]\!]_{\setminus B}, \neg[\![P]\!]_{\downarrow B}) & \text{if } [\![P]\!] \text{ is a } \mathcal{T}_i\text{-lemma} \\ [\![P]\!]_{\downarrow B} & \text{if } [\![P]\!] \in A \\ \mathsf{true} & \text{if } [\![P]\!] \in B \\ I_{P_1} \vee I_{P_2} & \text{if } P = \langle P_1, l, P_2 \rangle \text{ and } l \text{ is } A\text{-colored} \\ I_{P_1} \wedge I_{P_2} & \text{if } P = \langle P_1, l, P_2 \rangle \text{ and } l \text{ is } B\text{-colorable} \end{cases}$$

Theorem 1 ([14,6]). *If P is a colorable $(\mathcal{T}_1, \ldots, \mathcal{T}_n)$-refutation from $A \cup B$, then I_P is a ground interpolant for A, B.*

Proof. By induction on the number of nodes in P, *(i)* $A \models_{\mathcal{T}_1+\cdots+\mathcal{T}_n} I_P \vee [\![P]\!]_{\setminus B}$, *(ii)* $B, I_P \models_{\mathcal{T}_1+\cdots+\mathcal{T}_n} [\![P]\!]_{\downarrow B}$, and *(iii)* I_P is AB-colored. □

Note that I_P as defined here is not unique because the conditions for the cases are not mutually exclusive. For our purposes, this is inconsequential. Note also that this definition is obtained from the propositional interpolation algorithm of [13] by the addition of the first case (for theory lemmas).

2.5 Modifying Proof Trees

When $[\![P']\!] \subseteq [\![P]\!]$, we say that P' is *stronger* than P, and that P is *weaker* than P'. Clearly, any proof tree stronger than a refutation is also a refutation.

We will use a simple, typically unnamed, construction to strengthen a proof, given strengthened subproofs [1]. Let $P = \mathsf{stitch}(P_1, l, P_2)$ be specified as follows: if $l \in [\![P_1]\!]$ and $\neg l \in [\![P_2]\!]$, then $P = \langle P_1, l, P_2 \rangle$; if $l \notin [\![P_1]\!]$ then $P = P_1$; otherwise $P = P_2$. Thus, stitch attempts to resolve two given proof trees over a specified literal, returning one of the input trees when resolution is not possible.

Lemma 2. *Let P_1, P_2 be arbitrary proof trees, l be an arbitrary literal and α, β be arbitrary clauses.*

(i) If $[\![P_1]\!] \subseteq \alpha \cup l$ and $[\![P_2]\!] \subseteq \beta \cup \neg l$, then $[\![\mathsf{stitch}(P_1, l, P_2)]\!] \subseteq \alpha \cup \beta$.
(ii) If $\langle P_1, l, P_2 \rangle$ is defined and the proof trees P_1' and P_2' are stronger than P_1 and P_2 respectively, then $\mathsf{stitch}(P_1', l, P_2')$ is stronger than $\langle P_1, l, P_2 \rangle$.

Another way of strengthening proof trees is by changing the order of pivots. If $P = \langle \langle P_1, l_1, P_2 \rangle, l_2, P_3 \rangle$ and $P' = \mathsf{stitch}(\mathsf{stitch}(P_1, l_2, P_3), l_1, \mathsf{stitch}(P_2, l_2, P_3))$, we say then that P' is obtained from P by a *raising* the pivot l_2 over l_1 [11]; see also Exchange Lemma 4.1.3 of [5].

Lemma 3. *Let P and P' be as above. Then:*

(i) P' is stronger than P.

(ii) If $l_1 \neq l_2$ *and* $l_1 \neq \neg l_2$*, then* $P' = \langle \mathsf{stitch}(P_1, l_2, P_3), l_1, \mathsf{stitch}(P_2, l_2, P_3)\rangle$.

Inductive proofs based on node counts will rely on the simple facts in the following lemma. Here and in the sequel, $|P|$ denotes the number of nodes in P and $|P|_c$ denotes the number of critical nodes in P.

Lemma 4. *Let* P_1 *and* P_2 *be arbitrary proof trees and* l *be an arbitrary literal. Let* ϵ *be* 1 *if* l *is uncolorable and* 0 *otherwise.*

(i) If $\langle P_1, l, P_2\rangle$ *is defined then* $|\langle P_1, l, P_2\rangle| = |P_1| + |P_2| + 1$ *and* $|\langle P_1, l, P_2\rangle|_c = |P_1|_c + |P_2|_c + \epsilon$*;*
(ii) $|\mathsf{stitch}(P_1, l, P_2)| \leq |P_1| + |P_2| + 1$ *and* $|\mathsf{stitch}(P_1, l, P_2)|_c \leq |P_1|_c + |P_2|_c + \epsilon$.

3 Obtaining Colorable Refutations

Theorem 1 tells us how to derive ground interpolants from colorable refutations. In this section, we show how and under what conditions it is possible to obtain colorable refutations from those produced by an SMT solver.

3.1 Prelude

As argued in §4, the only literals occurring in proof trees produced by SMT solvers, under standard assumptions, are (colorable) literals occurring in the input set $A \cup B$ or (dis)equalities between terms that occur in $A \cup B$. Thus, the only uncolorable atoms are equalities $x{=}y$, where x is A-colored and y is B-colored.

Figure 1 shows the basic transformation that removes one such equality from a proof tree. It uses equality interpolation (§2.3) to replace a lemma $\alpha \vee x{=}y$ containing the uncolorable equality $x{=}y$ with two colorable lemmas $\alpha \vee x{=}z$ and $\alpha \vee z{=}y$. Occurrences of the corresponding disequality $x \neq y$ are then split into $x \neq z \vee z \neq y$. This transformation can be applied repeatedly, under appropriate conditions discussed below, to eliminate all uncolorable equalities.

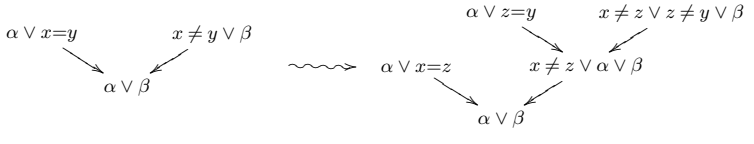


Fig. 1. Basic transformation to eliminate an uncolorable equality $x{=}y$

Clearly, we have to assume that all theories be equality interpolating. Additionally, the use of equality interpolation imposes a hard constraint on the proof trees modifiable by the basic transformation above: there must be at most one uncolorable equality in each leaf clause (see Figure 2). This restriction will define the class of almost-colorable refutations. Note that if all the theories are convex then the restriction causes no loss of generality. In a convex theory, if

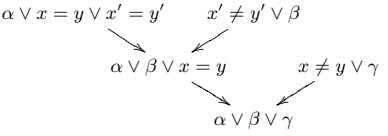


Fig. 2. Equality interpolation cannot be applied to the theory lemma $\alpha \vee x{=}y \vee x'{=}y'$ with two uncolorable equalities

$\alpha \vee x{=}y \vee x'{=}y'$ is a lemma, then either $\alpha \vee x{=}y$ or $\alpha \vee x'{=}y'$ must be a lemma as well.

The method of [6] employs the basic transformation to eliminate all uncolorable equalities from *ie-local* refutations—those in which all uncolorable equalities are resolved before other literals. However, as witnessed by the example in Figure 3, ie-locality is not a necessary condition for the applicability of the basic transformation.

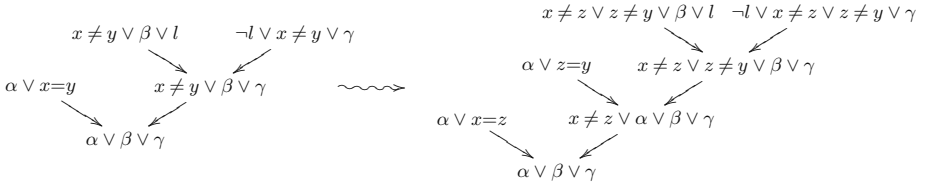


Fig. 3. Basic transformation applied to remove the uncolorable equality $x{=}y$ from a non-ie-local proof. The literal l is assumed colorable.

The real difficulty with producing colorable refutations from uncolorable ones is not the lack of ie-locality, but merges of uncolorable equalities. The example on the left in Figure 4 merges the equality $x{=}y$ from two leaves. If we perform equality interpolation on only one of the two occurrences of this equality in a leaf, we get a strictly weaker proof with the uncolorable equality $x{=}y$ still in the derived clause. If we perform equality interpolation on both occurrences and obtain distinct equality interpolants, then also the modified proof is strictly weaker than the original, irrespective of how we split the disequality $x \neq y$.

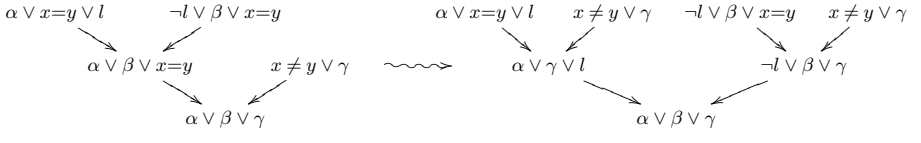


Fig. 4. Raising the merged pivot $x{=}y$ eliminates the merge

It can be shown that refutations that are almost-colorable and ie-local contain no merges of uncolorable equalities. For this reason, the approach of [6] insists on

ie-locality. We place no such restriction, prefering to eliminate the problematic merges by changing the order of pivots as shown in Figure 4.

3.2 The Colorability Theorem

Let $\mathcal{P}(A, B)$ be the set of all $(\mathcal{T}_1, \ldots, \mathcal{T}_n)$-proof trees from $A \cup B$ which use only theory lemmas satisfying the following conditions:

- (col_1) every uncolorable literal in the lemma is an equality or a disequality
- (col_2) at most one literal in the lemma is an uncolorable equality

We will call proofs in $\mathcal{P}(A, B)$ *almost-colorable*. Clearly, all colorable proof trees from $A \cup B$ are also almost-colorable.

Theorem 2. *Let the theories $\mathcal{T}_1, \ldots, \mathcal{T}_n$ be equality-interpolating. If $\mathcal{P}(A, B)$ contains a refutation, then it contains a colorable refutation.*

Proof. Since every literal that occurs in a refutation must be resolved upon at some node, the existence of an uncolorable (dis)equality in a refutation implies the existence of a critical node in it. Thus, to prove the theorem, it suffices to show that there exists a refutation with no critical nodes. We will establish this by proving the following more general statement: *If $P \in \mathcal{P}(A, B)$ has no uncolorable disequalities in its clause $[\![P]\!]$, then there exists a stronger proof tree $P' \in \mathcal{P}(A, B)$ with no critical nodes.* We prove this claim by well-founded induction over the relation $\prec$ defined by:

$$P \prec Q \quad \text{iff} \quad |P|_c < |Q|_c \;\text{ or }\; |P|_c = |Q|_c \text{ and } |P| < |Q|.$$

The proof breaks down into five cases. In all cases, it is easily verified that the offered proof tree P' belongs to $\mathcal{P}(A, B)$, either directly or using the simple fact that $\langle P_1, l, P_2\rangle \in \mathcal{P}(A, B)$ if and only if $P_1, P_2 \in \mathcal{P}(A, B)$. So, we will focus only on verifying that P' is stronger than P and has no critical nodes.

Case 1: P is a single node. We can take P' to be P, which has no internal nodes and, hence, no critical nodes.

Case 2: $P = \langle P_1, l, P_2\rangle$. We assume, without loss of generality, that l is an atom. Lemma 1 implies that there are no uncolorable disequalities in $[\![P_1]\!]$ and by Lemma 4 we infer that $P_1 \prec P$. Thus, the induction hypothesis applies to P_1 ensuring the existence of a proof tree P_1' that is stronger than P_1 and has no critical nodes. If there are no critical nodes in P_1, then we will let P_1' be P_1.

Case 2.1: $l \notin [\![P_1']\!]$. We can take P' to be P_1', which has no critical nodes. Since P_1' is stronger than P_1 and $l \notin [\![P_1']\!]$, it follows by Lemma 1 that P_1' is stronger than P.

Case 2.2: $l \in [\![P_1']\!]$.

Case 2.2.1: l is colorable. Lemma 1 then implies that there are no uncolorable disequalities in $[\![P_2]\!]$. Since $P_2 \prec P$ by Lemma 4, from the induction hypothesis

we obtain a proof tree P_2' that is stronger than P_2 and contains no critical nodes. Let $P' = \mathsf{stitch}(P_1', l, P_2')$. Since l is colorable, it follows from Lemma 4 that P' does not contain any critical nodes, either. It also follows, by Lemma 2, that P' is stronger than P.

Case 2.2.2: l is uncolorable. By property (col_1), we have that l is an uncolorable equality $x{=}y$. We can infer from the absence of uncolorable disequalities in $[\![P]\!]$ and Lemma 1 that $x \neq y$ is the only uncolorable disequality in P_2.

Case 2.2.2.1: P_1' is a single node. Let $[\![P_1']\!] = \gamma \uplus (x{=}y)$. We know that there are no uncolorable disequalities in $[\![P_1']\!]$. This, together with uncolorability of $x{=}y$ and the fact $P_1' \in \mathcal{P}(A, B)$, implies that all the literals in γ are colorable. Now, $[\![P_1']\!]$ must be a theory lemma because $x{=}y$ is not colorable. Since our theories are assumed to be equality-interpolating, there exists an equality interpolant z for the clause $[\![P_1']\!]$. Let Q^x be the single-node proof tree with $[\![Q^x]\!] = \gamma \cup (x{=}z)$ and let Q^y be the single-node proof tree with $[\![Q^y]\!] = \gamma \cup (z{=}y)$. Note that Q^x and Q^y are colorable and, since they have no internal nodes, they have no critical nodes either.

Let $[\![P_2]\!] = \delta \uplus (x \neq y)$. From Lemma 5 below, we obtain a proof tree $P_2^* \in \mathcal{P}(A, B)$ such that $|P_2^*|_c \leq |P_2|_c$ and $[\![P_2^*]\!] \subseteq \delta \cup \{x \neq z, z \neq y\}$. Since $x \neq y$ is the only uncolorable disequality in $[\![P_2]\!]$, and the disequalities $x \neq z$ and $z \neq y$ are colorable, there can be no uncolorable disequalities in $[\![P_2^*]\!]$. Since the root is a critical node in P, we have by Lemma 4 that $|P_2|_c < |P|_c$. Thus, the induction hypothesis applies to P_2^*, yielding a proof tree P_2' stronger than P_2^* and without critical nodes. We take P' to be $\mathsf{stitch}(Q_1^x, x{=}z, \mathsf{stitch}(Q_1^y, z{=}y, P_2'))$. By Lemma 4 there are no critical nodes in P'. Since $[\![P']\!] \subseteq \gamma \cup \delta$, P' is stronger than P.

Case 2.2.2.2: $P_1' = \langle P_{11}, l', P_{12} \rangle$. Since there are no critical nodes in P_1' and no uncolorable disequalities in $[\![P_1']\!]$, it follows that the literal l' is colorable and, from Lemma 1, that there are no uncolorable disequalities in $[\![P_{11}]\!]$ and $[\![P_{12}]\!]$. Let $P^\dagger = \langle P_1', x{=}y, P_2 \rangle$. Note that $P^\dagger$ is well-defined (since $x{=}y \in [\![P_1']\!]$) and stronger than P (by Lemma 2). Note also that $x{=}y$ differs from l' and $\neg l'$ since l' is colorable and $x{=}y$ is uncolorable. We raise the pivot $x{=}y$ over l' in $P^\dagger$ to get $P^\ddagger = \mathsf{stitch}(Q_1, l', Q_2)$, where $Q_i = \mathsf{stitch}(P_{1i}, x{=}y, P_2)$ $(i = 1, 2)$. By Lemma 3, we have $P^\ddagger = \langle Q_1, l', Q_2 \rangle$.

We now show that the induction hypothesis applies to Q_i $(i = 1, 2)$. Since $x \neq y \in [\![P_2]\!]$, we have that Q_i is either P_{1i} or $\langle P_{1i}, x{=}y, P_2 \rangle$. Since $x \neq y$ is the only uncolorable disequality in $[\![P_2]\!]$ and there are no uncolorable disequalities in $[\![P_{1i}]\!]$, we can infer using Lemma 2 that there are no uncolorable disequalities in $[\![Q_i]\!]$. We also have (by Lemma 4) that $|Q_i|_c \leq 1 + |P_1'|_c + |P_2|_c = 1 + |P_2|_c$ and $|P|_c = 1 + |P_1|_c + |P_2|_c$. Thus, $|Q_i|_c \leq |P|_c$. Moreover, if $|Q_i|_c = |P|_c$ then we must have $|P_1|_c = 0$, in which case P_1' is P_1 (see Case 2.2) and by Lemma 4, we have $|Q_i| < |P|$. It follows that $Q_i \prec P$.

Thus, we have proof trees Q_i' that have no critical nodes and are stronger than Q_i. We take P' to be $\mathsf{stitch}(Q_1', l', Q_2')$. There are no critical nodes in P' (Lemma 4) and P' is stronger than $P^\ddagger$ (Lemma 2), which in turn is stronger than $P^\dagger$ (Lemma 3), which, as we have already noticed, is stronger than P. □

Lemma 5. *Let P be a proof in $\mathcal{P}(A,B)$, $x \neq y$ be an uncolorable disequality and z be an arbitrary term. Then, there exists $P^* \in \mathcal{P}(A,B)$ such that:*

(i) $[\![P^*]\!] \subseteq [\![P]\!] \cup \{x \neq z, z \neq y\} \setminus \{x \neq y\}$;
(ii) $|P^*|_c \leq |P|_c$.

Proof. We argue by induction on the number of nodes in P.

Case 1: $x \neq y$ does not occur in $[\![P]\!]$. Take P^* to be P.

Case 2: $[\![P]\!] = \delta \uplus (x \neq y)$.

Case 2.1: P is a single node. The uncolorability of $x \neq y$ implies that $[\![P]\!]$ is a theory lemma. Take P^* be the single node with $[\![P^*]\!] = \delta \cup \{x \neq z, z \neq y\}$. By the transitivity of equality, $[\![P^*]\!]$ is also a theory lemma.

Case 2.2: $P = \langle P_1, l, P_2 \rangle$. We know from Lemma 1 that $[\![P_1]\!] \subseteq \delta \cup \{x \neq y, l\}$ and $[\![P_2]\!] \subseteq \delta \cup \{x \neq y, \neg l\}$. By the induction hypothesis, there exist P_1^* and P_2^* such that $[\![P_1^*]\!] \subseteq \delta \cup \{x \neq z, z \neq y, l\}$, $[\![P_2^*]\!] \subseteq \delta \cup \{x \neq z, z \neq y, \neg l\}$ and P_1^*, P_2^* have no more critical nodes than P_1, P_2 respectively. Let $P^* = \mathsf{stitch}(P_1^*, l, P_2^*)$. It follows from Lemma 2 that $[\![P^*]\!] \subseteq \delta \cup \{x \neq z, z \neq y\}$. Finally, by Lemma 4, $|P^*|_c \leq \epsilon + |P_1^*|_c + |P_2^*|_c \leq \epsilon + |P_1|_c + |P_2|_c = |P|_c$, for suitable $\epsilon \in \{0, 1\}$. □

3.3 The Colorability Algorithm

The proofs of Theorem 2 and Lemma 5 are constructive and directly lead to Algorithm 1 and Algorithm 2. The algorithms use the following functions: is_lit_colorable tests if a literal is colorable; eq_interp computes an equality interpolant for the input clause; node creates a single-node proof annotated with the given clause.

Merges of uncolorable equalities have the potential to exponentially blow-up the size of $\mathsf{mk\_colorable}(P)$ because raising an uncolorable-equality pivot doubles the right subproof, as in Figure 4. The following result guarantees linear growth in the absence of these problematic merges.

Theorem 3. *If P is a refutation in $\mathcal{P}(A,B)$ such that there are no merges of uncolorable equalities in P, then $|\mathsf{mk\_colorable}(P)| \leq 2 \cdot |P|$.*

Proof. We will prove the following more general statement: *Let P be a proof in $\mathcal{P}(A,B)$ such that there are no uncolorable equalities in $[\![P]\!]$ and no merges of uncolorable equalities in P. Let $P' = \mathsf{mk\_colorable}(P)$. Then:*

(i) $|P'| \leq 2 \cdot |P|$;
(ii) If $P = \langle P_1, l, P_2 \rangle$ and P_1 has no critical nodes, then $|P'| \leq |P_1| + 2 \cdot |P_2| + 3$.

We will use the easily proven facts that $|\mathsf{split}(P, x \neq y, z)| \leq |P|$ and that if there are no critical nodes in P, then $\mathsf{mk\_colorable}(P) = P$. The proof will follow the structure of the proof of Theorem 2.
Case 1: Trivial.
Case 2.1: We have $|P'| = |P_1'|$.

Algorithm 1. mk_colorable(P)

```
1: if |P| = 1 then (* Case 1 *)
2:    P' ← P
3: else (* Case 2 *)
4:    let P be ⟨P1, l, P2⟩
5:    P1' ← mk_colorable(P1)
6:    if l ∉ [[P1']] then (* Case 2.1 *)
7:       P' ← P1'
8:    else (* Case 2.2 *)
9:       if is_lit_colorable(l) then (* Case 2.2.1 *)
10:         P2' ← mk_colorable(P2)
11:         P' ← stitch(P1', l, P2')
12:      else (* Case 2.2.2 *)
13:         let l be x=y
14:         if |P1'| = 1 then (* Case 2.2.2.1 *)
15:            let [[P1']] be γ ⊎ x=y
16:            z ← eq_interp(γ ⊎ x=y)
17:            Q^x ← node(γ ∪ x=z)
18:            Q^y ← node(γ ∪ z=y)
19:            P2* ← split(P2, x ≠ y, z)
20:            P2' ← mk_colorable(P2*)
21:            P' ← stitch(Q^x, x=z, stitch(Q^y, z=y, P2'))
22:         else (* Case 2.2.2.2 *)
23:            let P1' be ⟨P11, l', P12⟩
24:            Q1 ← stitch(P11, x=y, P2)
25:            Q2 ← stitch(P12, x=y, P2)
26:            Q1' ← mk_colorable(Q1)
27:            Q2' ← mk_colorable(Q2)
28:            P' ← stitch(Q1', l', Q2')
29: return P'
```

Algorithm 2. split($P, x \neq y, z$)

```
1: if x ≠ y ∉ [[P]] then (* Case 1 *)
2:    P* ← P
3: else (* Case 2 *)
4:    let [[P]] be δ ⊎ (x ≠ y)
5:    if |P| = 1 then (* Case 2.1 *)
6:       P* ← node(δ ∪ {x ≠ z, z ≠ y})
7:    else (* Case 2.2 *)
8:       let P be ⟨P1, l, P2⟩
9:       P1* ← split(P1, x ≠ y, z)
10:      P2* ← split(P2, x ≠ y, z)
11:      P* ← stitch(P1*, l, P2*)
12: return P*
```

(i) By the induction hypothesis, $|P_1'| \leq 2 \cdot |P_1|$. But $|P_1| < |P|$.
(ii) If P_1 has no critical nodes, then $P_1' = P_1$.

Case 2.2.1:

(i) $|P'| \leq |P'_1| + |P'_2| + 1 \leq 2 \cdot |P_1| + 2 \cdot |P_2| + 1 = 2 \cdot (|P_1| + |P_2|) + 1$. We have $|P| = |P_1| + |P_2| + 1$. Thus, $|P'| \leq 2 \cdot (|P| - 1) + 1 < 2 \cdot |P|$.
(ii) If P_1 has no critical nodes, then $P'_1 = P_1$. Thus, $|P'| \leq |P_1| + |P'_2| + 1 \leq |P_1| + 2 \cdot |P_2| + 1$.

Case 2.2.2.1:

(i) $|P'| \leq |Q^x| + |Q^y| + |P'_2| + 2$. We have $|Q^x| = |Q^y| = 1$, and by induction hypothesis, $|P'_2| \leq 2 \cdot |P^*_2| \leq 2 \cdot |P_2|$. Thus $|P'| \leq 2 \cdot |P_2| + 4$. We also know that $|P_2| \leq |P| - 2$. Thus $|P'| \leq 2 \cdot |P|$.
(ii) If P_1 has no critical nodes, then $P'_1 = P_1$. The assumption for this case is that $|P'_1| = 1$. Thus, $|P'| \leq |P'_2| + 4 \leq |P_1| + 2 \cdot |P_2| + 3$.

Case 2.2.2.2: Without loss of generality, assume $x{=}y \notin [\![P_{11}]\!]$ and $x{=}y \in [\![P_{12}]\!]$. Thus, $Q_1 = P_{11}$ and $Q_2 = \langle P_{12}, l, P_2 \rangle$.

(i) $|P'| \leq |Q'_1| + |Q'_2| + 1$. Note that there are no critical nodes in either P_{11} or in P_{12}. Thus, $Q'_1 = P_{11}$ and $|Q'_2| \leq |P_{12}| + 2 \cdot |P_2| + 3$. Thus, $|P'| \leq |P_{11}| + |P_{12}| + 2 \cdot |P_2| + 3 = |P'_1| + 2 \cdot |P_2| + 2 \leq 2 \cdot (|P_1| + |P_2| + 1) = 2 \cdot |P|$.
(ii) Assume no critical nodes in P_1. Then $P'_1 = P_1 = \langle P_{11}, l, P_{12} \rangle$. Also, P_{11} and P_{12} have no critical nodes and $Q'_1 = P_{11}$, $|Q'_2| \leq |P_{12}| + 2 \cdot |P_2| + 3$. Thus, $|P'| \leq |Q'_1| + |Q'_2| + 1 \leq |P_{11}| + |P_{12}| + 2 \cdot |P_2| + 3 + 1 = |P_1| + 2 \cdot |P_2| + 3$.□

More substantial complexity analysis is left for future work. Our algorithms can be easily modified (by memoization) to operate on proof DAGs instead on proof trees. It would be particularly interesting to understand the complexity of these optimized versions.

4 Almost-Colorable Refutations from SMT Solvers

Modern SMT solvers integrate a SAT solver and several solvers for specific theories. An abstract model of an SMT solver that covers the essentials of the cooperation algorithm is given in [12] in the form of a transition system called NODPLL (Nelson-Oppen with DPLL), which in turn is an elaboration of the abstract system DPLL($\mathcal{T}$) of [16].

In this section, starting with a simplified (more abstract) version of the system NODPLL described in [12], we obtain the system NODPLL$^{\text{pf}}$ which tracks the derivations of all conflict clauses and thus produces $(\mathcal{T}_1, \ldots, \mathcal{T}_n)$-refutations when it finds that the input set of clauses is inconsistent.

The main parameters of the system NODPLL$^{\text{pf}}$ are theories $\mathcal{T}_1, \ldots, \mathcal{T}_n$ with disjoint signatures $\Sigma_1, \ldots, \Sigma_n$. The union signature and the union theory will be denoted Σ and $\mathcal{T}$ respectively. Additional parameters of NODPLL$^{\text{pf}}$ are a set L of Σ-literals and a set E of equalities between Σ-terms. Intuitively, the set L consists of literals that the SAT solver can decide on, and E is the set of equalities that theory solvers may share without sharing them with the SAT solver. It is

not required that L and E be disjoint. (In extensions of the system, one can also promote L and E from parameters to system variables, adding rules to grow them dynamically.)

NODPLL$^{\mathrm{pf}}$ is a transition system over states of the form $\langle \mathsf{P}, \mathsf{M}, \mathsf{C} \rangle$ where *(i)* P is a set of proof trees over Σ-clauses; *(ii)* M is a *checkpointed sequence*, any element of which is either the special symbol $\Box$, or a literal from $\mathsf{L} \cup \mathsf{E}$; *(iii)* C, the state's *conflict proof tree*, is either a proof tree for a clause that is a subset of $\mathsf{L} \cup \mathsf{E}$, or the special symbol none, denoting the absence of conflict.

As before, we use the notation $\mathsf{node}(\gamma)$ for the proof tree with a single node whose associated clause is γ.

The input to NODPLL$^{\mathrm{pf}}$ is a set S of ground $\mathcal{T}$-clauses. With a given S, the initialization procedure specifies the sets L and E, and an initial state of NODPLL$^{\mathrm{pf}}$. The initial state naturally has $\mathsf{P} = \{\mathsf{node}(\gamma) \,|\, \gamma \in S\}$, M equal to the empty sequence, and $\mathsf{C} = \mathsf{none}$. As for the parameter literal sets L and E, there are two main options. To define them, let L_S denote the set of all literals that occur in S, and let E_S be the set of all equalities between distinct terms that occur in S. For *Nelson-Oppen initialization*, we take $\mathsf{L} = L_S^{\pm 1}$ and $\mathsf{E} = E_S$. For *DTC initialization*, we take $\mathsf{L} = L_S^{\pm 1} \cup E_S^{\pm 1}$ and $\mathsf{E} = \varnothing$. (The notation $X^{\pm 1}$ stands for the set that contains the literals of X and their negations.) To be general, we will assume only that $\mathsf{L} \subseteq L_S^{\pm 1} \cup E_S^{\pm 1}$ and $\mathsf{E} \subseteq E_S$.

The transition rules of NODPLL$^{\mathrm{pf}}$ are given in Figure 5. The index i ranges over $\{0, \ldots, n\}$. The symbol $\models_i$ stands for the theory entailment $\models_{\mathcal{T}_i}$ in the case when $i > 0$. For $i = 0$, the symbol stands for the propositional entailment from a single clause of P. More precisely, the condition $\mathsf{M} \models_0 l$ in the rule Infer_0 stands for "there exist a proof tree $P \in \mathsf{P}$ such that $[\![P]\!] = \{\neg l_1, \ldots, \neg l_k, l\}$ and $l_1, \ldots, l_k \in$

$$\textsf{Decide} \quad \frac{l \in \mathsf{L} \qquad l, \neg l \notin \mathsf{M}}{\mathsf{M} := \mathsf{M} \,\Box\, l}$$

$$\mathsf{Infer}_i \quad \frac{l \in \mathsf{L} \cup \mathsf{E} \qquad \mathsf{M} \models_i l \qquad l, \neg l \notin \mathsf{M}}{\mathsf{M} := \mathsf{M}\; l}$$

$$\mathsf{Conflict}_i \quad \frac{\mathsf{C} = \mathsf{none} \qquad l_1, \ldots, l_k \in \mathsf{M} \qquad l_1, \ldots, l_k \models_i \mathsf{false} \qquad k > 0}{\mathsf{C} := \mathsf{pf}_i\{\neg l_1, \ldots, \neg l_l\}}$$

$$\mathsf{Explain}_i \quad \frac{\neg l \in [\![\mathsf{C}]\!] \qquad l_1, \ldots, l_k \prec_{\mathsf{M}} l \qquad l_1, \ldots, l_k \models_i l}{\mathsf{C} := \langle \mathsf{pf}_i\{\neg l_1, \cdots, \neg l_k, l\}, l, \mathsf{C} \rangle}$$

$$\textsf{Learn} \quad \frac{[\![\mathsf{C}]\!] \subseteq \mathsf{L} \qquad \mathsf{C} \notin \mathsf{P}}{\mathsf{P} := \mathsf{P} \cup \{\mathsf{C}\}}$$

$$\textsf{Backjump} \quad \frac{\mathsf{C} \in \mathsf{P} \qquad [\![\mathsf{C}]\!] = \{l, l_1, \ldots, l_k\} \qquad \mathsf{level}\, l_1, \ldots, \mathsf{level}\, l_k \leq m < \mathsf{level}\, l}{\mathsf{C} := \mathsf{none} \qquad \mathsf{M} := \mathsf{M}^{[m]}\; \neg l}$$

Fig. 5. Rules of NODPLL$^{\mathrm{pf}}$. Above each line is the rule's *guard*, below is its *action*.

M". Similarly, $l_1, \ldots, l_k \models_0 \mathsf{false}$ and $l_1, \ldots, l_k \not\models_0 l$ in the rules $\mathsf{Conflict}_0$ and $\mathsf{Explain}_0$ stand for the existence of $P \in \mathsf{P}$ satisfying $[\![P]\!] = \{\neg l_1, \ldots, \neg l_k\}$ and $[\![P]\!] = \{\neg l_1, \ldots, \neg l_k, l\}$ respectively.

When $i > 0$, the notation $\mathsf{pf}_i\gamma$ is synonymous with $\mathsf{node}(\gamma)$. As for $\mathsf{pf}_0\gamma$, it is used only in rules $\mathsf{Conflict}_0$ and $\mathsf{Explain}_0$, and it stands for a proof tree $P \in \mathsf{P}$ such that $[\![P]\!] = \gamma$. In view of the definitions in the previous paragraph, such a proof tree P always exists.

The number of occurrences of $\square$ in M is the *current decision level*. Thus, we can write $\mathsf{M} = \mathsf{M}^{\langle 0\rangle} \square \mathsf{M}^{\langle 1\rangle} \square \cdots \square \mathsf{M}^{\langle d\rangle}$, where d is the current decision level, and $\square$ does not occur in any $\mathsf{M}^{\langle k\rangle}$. It is an invariant that for every $k > 0$, $\mathsf{M}^{\langle k\rangle}$ is non-empty. The first element of $\mathsf{M}^{\langle k\rangle}$ $(k > 0)$ is the k^{th} *decision literal* of M. By $\mathsf{M}^{[k]}$, where $0 \leq k \leq d$, we denote the prefix $\mathsf{M}^{\langle 0\rangle} \square \cdots \square \mathsf{M}^{\langle k\rangle}$ of M.

The rule $\mathsf{Explain}_i$ uses the notation $l \prec_{\mathsf{M}} l'$; by definition, this means that both literals are in M and the (unique) occurrence of l precedes in M the (unique) occurrence of l'. For correctness of this definition, we need to know that *any literal can occur at most once* in M, which is another easily verified invariant of NODPLL$^{\mathrm{pf}}$. Finally, the function level used in the $\mathsf{Backjump}$ rule is defined only for literals that occur in M; for these literals $\mathsf{level}\, l = k$ holds if l occurs in $\mathsf{M}^{\langle k\rangle}$.

A NODPLL$^{\mathrm{pf}}$ *execution* is a finite or infinite sequence $s_0, s_1, \ldots$ such that s_0 is an initial state and each state s_{i+1} is obtained from s_i by the application of one of the transition rules of the system. We can prove the following lemma by induction on the length of execution sequences.

Lemma 6. *If* NODPLL$^{\mathrm{pf}}$ *is given a clause set S as input, then, in any state,* C *is either* none *or a $(\mathcal{T}_1, \ldots, \mathcal{T}_n)$-proof tree from S.*

The results of [12] for the original NODPLL system apply to NODPLL$^{\mathrm{pf}}$ as well, with straightforward modifications of the proofs. Specifically, one can prove that the system NODPLL$^{\mathrm{pf}}$ is *terminating*: every execution is finite and ends in a state in which $\mathsf{C} = \mathsf{none}$ or $[\![\mathsf{C}]\!] = \varnothing$. The *soundness* of NODPLL$^{\mathrm{pf}}$ is actually a consequence of Lemma 6: if the system reaches a state in which C is a refutation ($[\![\mathsf{C}]\!] = \varnothing$), then S is $\mathcal{T}$-unsatisfiable. There are two *completeness* results: [2] if on an input S the system terminates in a state in which $\mathsf{C} = \mathsf{none}$, then S is $\mathcal{T}$-satisfiable, provided *(i)* the system is given the Nelson-Oppen initialization, and all the $\mathcal{T}_i$ are convex; or *(ii)* the system is given the DTC initialization.

Consider now the colorability of proof trees C of our system. The initialization assumption $\mathsf{L} \subseteq L_S^{\pm 1} \cup E_S^{\pm 1}$ and colorability of all literals in L_S (each of them occurs in A or in B) imply that the only uncolorable literals in $\mathsf{L} \cup \mathsf{E}$ are equalities from E_S or their negations. Thus, proof trees C always satisfy the property (col_1).

One way to satisfy (col_2) is to ensure that all literals in L are colorable; for instance, by initializing the system with L being the union of $L_S^{\pm 1}$ and all colorable (dis)equalities from $E_S^{\pm 1}$. To see that (col_2) holds in this case, note first that (by induction) all uncolorable literals in M are equalities from $\mathsf{E} \setminus \mathsf{L}$. This ensures the clause of C introduced by $\mathsf{Conflict}_i$ contains no uncolorable

[2] In the context of [12], we assume that the theories are *parametric*; for the classical first-order combination, we need to assume that the theories are stably-infinite [2].

equalities, and clauses introduced by $\mathsf{Explain}_i$ contain at most one uncolorable equality. Thus, (col_2) is satisfied, but note that the restriction we put on L makes NODPLL$^{\mathrm{pf}}$ potentially incomplete.

Another way to guarantee proof trees C satisfying (col_2) is to run the system NODPLL$^{\mathrm{pf}}$ with the following *convexity restriction*: allow rule $\mathsf{Conflict}_i$ to fire only when at most one of the literals $l_1, \ldots, l_k$ is a disequality and allow rule $\mathsf{Explain}_i$ to fire only when none of the literals $l_1, \ldots, l_k$ is a disequality. It is easy to see that if all theories $\mathcal{T}_i$ are convex, then the convexity restriction does not jeopardize the completeness of NODPLL$^{\mathrm{pf}}$.

Theorem 4. *Suppose $S = A \cup B$ is given as input to* NODPLL$^{\mathrm{pf}}$*. Suppose, in addition, that either* (a) *the system is run with the convexity restriction; or* (b) *the system is initialized so that all literals in L are colorable. Then, in all reachable states, the proof tree C is in $\mathcal{P}(A, B)$.*

Proof. Sketched in the preceding paragraphs. □

5 Conclusion

We have presented a simple approach for the generation of ground interpolants by SMT solvers supporting multiple theories. Our main contribution is an algorithm that transforms any *almost-colorable* refutation into one that is *colorable* and thus suitable for straigthforward interpolant extraction using known algorithms.

The definition of almost-colorable refutations is minimally demanding. We show that modern SMT solvers can produce such refutations with the slightest restrictions on their search strategy. What constitutes a good search strategy for interpolation remains an open question, but by being more general than previous approaches, we enable the design of more efficient interpolating SMT solvers.

The colorability algorithm uses a sequence of elementary proof transformations to convert an almost-colorable refutation into a colorable one. There is some flexibility in the order in which these transformations are applied. Our particular choice of the colorability algorithm ensures that for a subset of almost-colorable refutations—including the class of ie-local refutations that could be used with previous methods for ground interpolation—we at most double the size of the input tree. In practice, however, proofs are represented compactly as DAGs. More work is required to understand the effect of various transformation choices on DAG size.

Acknowledgment. We thank Alexander Fuchs, Jim Grundy and anonymous reviewers for suggestions that helped improve the paper.

References

1. Andrews, P.B.: Resolution with merging. J. ACM 15(3), 367–381 (1968)
2. Barrett, C., et al.: Satisfiability Modulo Theories. In: Biere, A., et al. (eds.) Handbook of Satisfiability, pp. 825–885. IOS Press, Amsterdam (2009)

3. Beyer, D., Zufferey, D., Majumdar, R.: CSIsat: Interpolation for LA+EUF. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 304–308. Springer, Heidelberg (2008)
4. Bozzano, M., et al.: Efficient theory combination via Boolean search. Information and Computation 204(10), 1493–1525 (2006)
5. Büning, H.K., Lettmann, T.: Propositional Logic: Deduction and Algorithms. Cambridge University Press, New York (1999)
6. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient interpolant generation in Satisfiability Modulo Theories. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 397–412. Springer, Heidelberg (2008)
7. de Moura, L., Bjørner, N.: Model-based theory combination. ENTCS 198, 37–49 (2008)
8. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
9. Fuchs, A., et al.: Ground interpolation for the theory of equality. In: TACAS. LNCS, vol. 5505, pp. 413–427. Springer, Heidelberg (2009)
10. Henzinger, T.A., et al.: Abstractions from proofs. In: POPL, pp. 232–244. ACM, New York (2004)
11. Jhala, R., McMillan, K.L.: Interpolant-based transition relation approximation. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 39–51. Springer, Heidelberg (2005)
12. Krstić, S., Goel, A.: Architecting solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In: Konev, B., Wolter, F. (eds.) FroCos 2007. LNCS, vol. 4720, pp. 1–27. Springer, Heidelberg (2007)
13. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
14. McMillan, K.L.: An interpolating theorem prover. Theoretical Computer Science 345(1), 101–121 (2005)
15. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
16. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Abstract DPLL and abstract DPLL modulo theories. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS, vol. 3452, pp. 36–50. Springer, Heidelberg (2005)
17. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint solving for interpolation. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 346–362. Springer, Heidelberg (2007)
18. Yorsh, G., Musuvathi, M.: A combination method for generating interpolants. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS, vol. 3632, pp. 353–368. Springer, Heidelberg (2005)

Interpolation and Symbol Elimination*

Laura Kovács[1] and Andrei Voronkov[2]

[1] EPFL
[2] University of Manchester

Abstract. We prove several results related to local proofs, interpolation and superposition calculus and discuss their use in predicate abstraction and invariant generation. Our proofs and results suggest that symbol-eliminating inferences may be an interesting alternative to interpolation.

1 Introduction

The study of interpolation in connection to verification has been pioneered by McMillan in connection with model checking [16], and by McMillan [17] and Henzinger et.al. [7] in connection with predicate abstraction. A number of papers appeared later discussing generation of interpolants for various theories and their use in verification, for example invariant generation [25,8,10,23,22,18,2].

In this paper we discuss interpolation and its use in verification. We start with preliminaries in Section 2. In Section 3 we define so-called local derivations and prove a general form of a result announced in [8], namely that interpolants can be extracted from proofs of special form (so-called local proofs) in *arbitrary theories and inference systems* sound for these theories. We also show that interpolants extracted from such proofs are boolean combinations of conclusions of so-called *symbol-eliminating* inferences. By observing that a similar form of symbol elimination turned out to be useful for generating complex quantified invariants in [13] and that interpolants obtained from proofs seem to be better for predicate abstraction and invariant generation than those obtained by quantifier elimination, we conclude that symbol elimination can be a key concept for applications in verification.

Further, in Section 4 we consider interpolation for inference systems dealing with universal formulas. We point out that a result announced in [18] is incorrect by giving a counterexample. We also further study the superposition inference system and show that, when we use a certain family of orderings, all ground proofs are local. This gives us a way of extracting interpolants from ground superposition proofs. We extend this result to the LASCA calculus of [12], which gives us a new procedure for generating interpolants for the quantifier-free theory of uninterpreted functions and linear rational arithmetic. Finally, in Section 5 we investigate the use of interpolants in invariant generation.

* This research was partly done in the frame of the Transnational Access Programme at RISC, Johannes Kepler University Linz, supported by the European Commission Framework 6 Programme for Integrated Infrastructures Initiatives under the project SCIEnce (contract No 026133). The first author was supported by the Swiss NSF.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 199–213, 2009.

2 Preliminaries

We will deal with the standard first-order predicate logic with equality. The equality symbol will be denoted by $\simeq$; instead of writing $\neg(s \simeq t)$ we will simply write $s \not\simeq t$. We allow all standard boolean connectives and quantifiers in the language and, in addition, assume that it contains the logical constants $\top$ for always true and $\bot$ for always false formulas.

We will denote formulas by A, B, C, D, terms by r, s, t, variables by x, y, z, constants by a, b, c and function symbols by f, g, possibly with indices. Let A be a formula with free variables $\bar{x}$, then $\forall A$ (respectively, $\exists A$) denotes the formula $(\forall \bar{x})A$ (respectively, $(\exists \bar{x})A$). A formula is called *closed*, or a *sentence*, if it has no free variables. We call a *symbol* a predicate symbol, a function symbol or a constant. Thus, variables are not symbols. We consider equality $\simeq$ part of the language, that is, equality is not a symbol. A formula or a term is called *ground* if it has no occurrences of variables. A formula is called *universal* if it has the form $(\forall \bar{x})A$, where A is quantifier-free. We write $C_1, \ldots, C_n \vdash C$ to denote that the formula $C_1 \wedge \ldots \wedge C_1 \rightarrow C$ is a tautology. Note that $C_1, \ldots, C_n, C$ may contain free variables.

A *signature* is any finite set of symbols. The *signature of a formula* A is the set of all symbols occurring in this formula. For example, the signature of $f(x) \simeq a$ is $\{f, a\}$. The *language of a formula* A, denoted by $\mathcal{L}_A$, is the set of all formulas built from the symbols occurring in A, that is formulas whose signatures are subsets of the signature of A.

Theorem 1 (Craig's Interpolation Theorem [3]). Let A, B be closed formulas and let $A \vdash B$. Then there exists a closed formula $I \in \mathcal{L}_A \cap \mathcal{L}_B$ such that $A \vdash I$ and $I \vdash B$.

In other words, every symbol occurring in I also occurs in both A and B. Every formula I satisfying this theorem will be called an *interpolant* of A and B.

Let us emphasise that Craig's Interpolation Theorem 1 makes no restriction on the signatures of A and B. There is a stronger version of the interpolation property proved in [15] (see also [19]) and formulated below.

Theorem 2 (Lyndon's Interpolation Theorem). Let A, B be closed formulas and let $A \vdash B$. Then there exists a closed formula $I \in \mathcal{L}_A \cap \mathcal{L}_B$ such that $A \vdash I$ and $I \vdash B$. Moreover, every predicate symbol occurring positively (respectively, negatively) in I, occurs positively (respectively, negatively), in both A and B.

By inspecting proofs of the two mentioned interpolation theorems one can also conclude that in the case when A and B are ground, they also have a ground interpolant; we will use this property later.

We call a *theory* any set of closed formulas. If T is a theory, we write $C_1, \ldots, C_n \vdash_T C$ to denote that the formula $C_1 \wedge \ldots \wedge C_1 \rightarrow C$ holds in all models of T. In fact, our notion of theory corresponds to the notion of *axiomatisable theory* in logic. When we work with a theory T, we call symbols occurring in T *interpreted* while all other symbols *uninterpreted*.

Note that Craig's interpolation also holds for theories in the following sense.

Theorem 3. Let A, B be formulas and let $A \vdash_T B$. Then there exists a formula I such that

1. $A \vdash_T I$ and $I \vdash B$;
2. every uninterpreted symbol of I occurs both in A and B;
3. every interpreted symbol of I occurs in B.

Likewise, there exists a formula I such that

1. $A \vdash I$ and $I \vdash_T B$;
2. every uninterpreted symbol of I occurs both in A and B;
3. every interpreted symbol of I occurs in A.

Proof. We start with proving the first part. By $A \vdash_T B$ and compactness there exists a finite number of formulas $T' \subseteq T$ such that $A, T' \vdash B$. Denote by C the conjunction of formulas in T', then we have $A \wedge C \vdash B$. By Craig's interpolation theorem 1 there exists a formula I whose symbols occur both in $A \wedge C$ and B such that $A \wedge C \vdash I$ and $I \vdash B$. Let us prove that I satisfies all conditions of the theorem. Note that $A \wedge C \vdash I$ implies $A \vdash_T I$, so the first condition is satisfied. Now take any uninterpreted symbol of I. Note that it cannot occur in C, since all symbols in C are interpreted, so it occurs in A, and so in both A and B. The condition on interpreted symbols is obvious since all symbols occurring in I also occur in B.

The second part is proved similarly, in this case take Craig's interpolant of A and $C \rightarrow B$. □

The proof of Theorem 3 is similar to a proof in [10]. It is interesting that this formulation of interpolation is not symmetric with respect to A and B since it does not state $A \vdash_T I$ and $I \vdash_T B$: only one of the implications $A \rightarrow I$ and $I \rightarrow B$ should be a theorem of T while the other one is a tautology in first-order logic. Thus, theory reasoning is required only to show one of these implications.

In the sequel we will be interested in the interpolation property with respect to a given theory T. For this reason, we will use $\vdash_T$ instead of $\vdash$ and relativise all definitions to T. To be precise, we call an *interpolant* of A and B any formula I with the properties $A \vdash_T I$ and $I \vdash_T B$.

If E is a set of expressions, for example, formulas, and constants $c_1, \ldots, c_n$ do not occur in E, then we say that $c_1, \ldots, c_n$ are *fresh* for E. We will less formally simply say *fresh constants* when E is the set of all expressions considered in the current context.

There is a series of papers on using interpolants in model checking and verification starting with [16]. Unfortunately, in some of these papers the notion of interpolant has been changed. Although the change seems to be minor, it affects Lyndon's interpolation property and also does not let one use interpolation for formulas with free variables. Namely [17,18] call an interpolant of A and B any formula I such that $A \vdash I$ and $B \wedge I$ is unsatisfiable. To avoid any confusion between the two notions of interpolant we introduce the following notion. We call a *reverse interpolant* of A and B any formula I such that $A \vdash_T I$ and $I, B \vdash_T \bot$. It is not hard to argue that reverse interpolants for A and B are exactly interpolants of A and $\neg B$ and that, when B is closed, reverse interpolants are exactly interpolants in the sense of [17,18].

3 Inference Systems and Local Derivation

In this section we will recall some terminology related to inference systems. It is commonly used in the theory of resolution and superposition [1,20]; we do not restrict ourselves to the superposition calculus.

Definition 1. An *inference rule* is an n-ary relation on formulas, where $n \geq 0$. The elements of such a relation are called *inferences* and usually written as

$$\frac{A_1 \quad \dots \quad A_n}{A} .$$

The formulas $A_1, \dots, A_n$ are called the *premises*, and the formula A the *conclusion*, of this inference. An *inference system* is a set of inference rules. An *axiom* of an inference system is any conclusion of an inference with 0 premises.

Any inferences with 0 premises and a conclusion A will be written without the bar, simply as A.

A *derivation* in an inference system is a tree built from inferences in this inference system. If the root of this derivation is A, then we say it is a *derivation of* A. A derivation of A is called a *proof* of A if it is finite and all leaves in the derivation are axioms. A formula A is called *provable* in I if it has a proof. We say that a derivation of A is *from assumptions* $A_1, \dots, A_m$ if the derivation is finite and every leaf in it is either an axiom or one of the formulas $A_1, \dots, A_m$. A formula A is said to be *derivable from* assumptions $A_1, \dots, A_m$ if there exists a derivation of A from $A_1, \dots, A_m$. A *refutation* is a derivation of $\bot$. □

Note that a proof is a derivation from the empty set of assumptions. Any derivation from a set of assumptions S can be considered as a derivation from any larger set of assumptions $S' \supseteq S$.

Let us now fix two sentences A and B. In the sequel we assume A and B to be fixed and give all definitions relative to A and B. Denote by $\mathcal{L}$ the intersection of the languages of A and B, that is, $\mathcal{L}_A \cap \mathcal{L}_B$. We call signature symbols occurring both in A and B *clean* and all other signature symbols *dirty*. For a formula C, we say that C is *clean* if $C \in \mathcal{L}$, otherwise we say that C is *dirty*. In other words, clean formulas contain only clean symbols and every dirty formula contains at least one dirty symbol.

Definition 2 (AB-derivation). Let us call an *AB-derivation* any derivation Π satisfying the following conditions.

(AB1) For every leaf C of Π one of following conditions holds:
1. $A \vdash_T \forall C$ and $C \in \mathcal{L}_A$ or
2. $B \vdash_T \forall C$ and $C \in \mathcal{L}_B$.

(AB2) For every inference

$$\frac{C_1 \quad \dots \quad C_n}{C}$$

of Π we have $\forall C_1, \dots, \forall C_n \vdash_T \forall C$.

We will refer to property (AB2) as *soundness*. □

We will be interested in finding reverse interpolants of A and B. The case $\mathcal{L}_A \subseteq \mathcal{L}_B$ is obvious, since in this case A is a reverse interpolant of A and B. Likewise, if $\mathcal{L}_B \subseteq \mathcal{L}_A$, then $\neg B$ is a reverse interpolant of A and B. For this reason, in the sequel we assume that $\mathcal{L}_A \not\subseteq \mathcal{L}_B$ and $\mathcal{L}_B \not\subseteq \mathcal{L}_A$, that is, *both A and B contain dirty symbols*.

We are especially interested in a special kind of derivation introduced in [8] and called *local* (or sometimes called *split-proofs*). The definition of a local derivation is relative to formulas A and B.

Definition 3 (Local AB-derivation). An inference

$$\frac{C_1 \quad \dots \quad C_n}{C}$$

in an AB-derivation is called *local* if the following two conditions hold.

(L1) Either $\{C_1, \dots, C_n, C\} \subseteq \mathcal{L}_A$ or $\{C_1, \dots, C_n, C\} \subseteq \mathcal{L}_B$.
(L2) If all of the formulas $C_1, \dots, C_n$ are clean, then C is clean, too.

A derivation is called *local* if so is every inference of this derivation. □

In other words, (L1) says that either all premises and the conclusion are in the language of A or all of them are in the language of B. Condition (L2) is natural (inferences should not introduce irrelevant symbols) but it is absent in other work. This condition is essential for us since without it the proof of our key Lemma 2 does not go through.

Papers [8,18] claim that from a local AB-refutation one can extract a reverse interpolant for A and B. Moreover, the proofs of these papers imply that the interpolant is universal when both A and B are universal and ground when both A and B are ground. However, the proofs of these properties use unsound arguments. First, the proof for the ground case from [8] uses an argument that for a sound inference

$$\frac{C_1 \quad \dots \quad C_n}{C}$$

the set of formulas $C_1, \dots, C_n, \neg C$ is propositionally unsatisfiable and then refers to a result on interpolation for propositional derivations from [16]. Unfortunately, this argument cannot be used: for example, the ground formula $a \not\simeq a$ is unsatisfiable in the theory of equality but not propositionally unsatisfiable. Second, the proof for the universal case in [18] refers to the ground case, but one cannot use this reduction since substituting terms for variables in non-ground local derivations may give a non-local derivation. Let us give an example showing that the result for the universal case is incorrect.

Example 1. Let A be the formula $a \not\simeq b$ and B the formula $\forall x(x \simeq c)$. Then a, b, c are dirty symbols and there is no clean symbol. The following is a local refutation in the superposition calculus:

$$\dfrac{\dfrac{\dfrac{x \simeq c \quad y \simeq c}{x \simeq y} \qquad a \not\simeq b}{y \not\simeq b}}{\bot}$$

One possible reverse interpolant of A and B is $\exists x \exists y (x \not\simeq y)$, however, this reverse interpolant is not universal. Let us show that there exist no universal reverse interpolant of A and B. Suppose, by contradiction, that such a reverse interpolant exists. Then it has the form $\forall x_1 \ldots \forall x_n I(x_1, \ldots, x_n)$, where $I(x_1, \ldots, x_n)$ is a quantifier-free formula, $x_1, \ldots, x_n$ are the only variables of $I(x_1, \ldots, x_n)$, and $I(x_1, \ldots, x_n)$ does not contain a, b. Take fresh constants $c_1, \ldots, c_n$, then we have $a \not\simeq b \vdash I(c_1, \ldots, c_n)$. By Craig's interpolation applied to ground formulas there exists a ground reverse interpolant J of $a \not\simeq b$ and $I(c_1, \ldots, c_n)$. By the conditions on the signature of J, J can contain no symbols. Therefore J is either equivalent to $\bot$ or equivalent to $\top$. The former is impossible since we do not have $a \not\simeq b \vdash \bot$, hence J is equivalent to $\top$. But then $I(c_1, \ldots, c_n)$ is a tautology. Since the constants c_i's are fresh, $\forall x_1 \ldots \forall x_n I(x_1, \ldots, x_n)$ is also a tautology. But we have $\forall x_1 \ldots \forall x_n I(x_1, \ldots, x_n), \forall x (x \simeq y) \vdash \bot$, so $\forall x (x \simeq y) \vdash \bot$ too. This contradicts the obvious observation that $\forall x (x \simeq y)$ has models. □

Below we will prove a general result on extracting interpolants from local refutations from which the ground case will follow. Moreover, in Section 4 we note that in the ground case, if we use the superposition calculus and a certain family of orderings, all superposition proofs are local, so an arbitrary superposition prover can be used for finding interpolants for ground formulas.

The proofs of our results will also show the structure of interpolants extracted from refutations. Any such interpolant is a boolean combination of some key inferences of the refutation, called *symbol-eliminating* inferences. This suggests that in addition to studying interpolants one can study symbol elimination in proofs.

Consider any AB-derivation Π. Note that by the soundness condition (AB2) we can replace every formula C occurring in this derivation by its universal closure $\forall C$ and obtain an AB-derivation Π' where inferences are only done on closed formulas. We will call such derivations Π' *closed*.

We want to show how one can extract interpolants from local proofs and also investigate the structure of such interpolants. To this end, we will prove key Lemma 2 about local proofs and introduce a notion that will be used to characterise interpolants. Let Π be a local AB-derivation and C a formula occurring in Π. We say that C is *justified by A (respectively by B) in Π* if C is clean and one of the following conditions hold:

(J1) C is a leaf of Π and $A \vdash_T C$.
(J2) C is a conclusion of an inference in Π of the form

$$\frac{C_1 \quad \cdots \quad C_n}{C} \ ,$$

such that for some $k \in \{1, \ldots, n\}$ the formula C_k is dirty and $C_k \in \mathcal{L}_A$ (respectively, $C_k \in \mathcal{L}_B$).

Note that the fact that C is justified by A does not necessarily imply that $A \vdash_T C$. Yet, the derivation of any such formula C uses at least one formula derived from A (see the proof of the following lemma).

Let us introduce a key notion of symbol-eliminating inference. We call a *symbol-eliminating inference* any inference of the form described in (J1) or (J2). That is, a

symbol-eliminating inference is an inference having a clean conclusion and either no premises at all, as in (J1), or at least one dirty premise, as in (J2). The name is due to the fact that the inference of (J2) has at least one dirty symbol occurring in premises and this symbol is "eliminated" in the conclusion. In the case of (J1) one can use the following explanation. Suppose, for example, that C in (J1) is justified by A. The we can consider C as derived from A by an inference

$$\frac{A}{C}\ ,$$

which also "eliminates" a dirty symbol occurring in A.

Lemma 1. Let Π be a local AB-derivation. Further, let C be a clean formula such that C is justified by A in Π and C is not a leaf of Π. Take the largest sub-derivation Π' of Π with the following properties:

1. The last inference of Π' is an inference of C satisfying (J2).
2. All formulas in Π' are in the language $\mathcal{L}_A$.

Then for every leaf C' of Π' one of the following conditions hold:

1. C' is clean and justified by B.
2. $C' \in \mathcal{L}_A$ and $A \vdash_T C'$.

The same holds if we swap A and B in the conditions.

Proof. First, consider the case when C' is dirty. Since every formula in Π' is in the language $\mathcal{L}_A$, then C' is in the language $\mathcal{L}_A$ too, but not in $\mathcal{L}_B$. Consider two cases. If C' is also a leaf of Π, then by (L2) and (AB1) we have $A \vdash_T C'$ and we are done. If C' is not a leaf of Π, consider the inference of C' in Π. Since Π is local, all premises of this inference are in the language $\mathcal{L}_A$, so the inference must be in Π', which contradicts the assumptions that C' is a leaf of Π' and that Π' is the largest sub-derivation satisfying (1) and (2).

It remains to consider the case when C' is clean. If $A \vdash_T C'$, then (since $C' \in \mathcal{L}_A$) we are done. If $A \nvdash C'$, then there exists an inference of C' in Π from some premises $C_1, \ldots, C_n$. If all these premises are in the language $\mathcal{L}_A$, then the inference itself belongs to Π', which contradicts the assumptions that C' is a leaf of Π' and that Π' is the largest sub-derivation satisfying (1) and (2). Therefore, at least one of the premises is dirty and belongs to $\mathcal{L}_B$, then C' is justified by B and we are done. □

Note that in the sub-derivation Π' of this lemma *every leaf is a conclusion of a symbol-eliminating inference*.

Our aim now is to show how one can extract an interpolant from a local derivation. To this end we will first generalise the notion of interpolant by relativising it to a quantifier-free formula C.

Definition 4 (C-interpolant). Let C be a quantifier-free formula. A formula I is called a *C-interpolant* of A and B if it has the following properties.

(C1) Every free variable of I is also a variable of C.
(C2) I is clean;
(C3) $\neg C, A \vdash_T I$;
(C4) $I, B \vdash_T C$. □

Note that the notion of reverse interpolant is a special case of the notion of C-interpolant. Indeed, take C to be $\bot$: then we have $A \vdash_T I$ and $I, B \vdash_T \bot$.

Lemma 2. Let Π be a local closed AB-derivation of a clean formula C. Then there exists a C-interpolant of A and B. Moreover, this C-interpolant is a boolean combination of conclusions of symbol-eliminating inferences of Π.

Proof. The proof is by induction on the number of inferences in Π. If Π consists of a single formula C, then by property (AB1) of AB-derivation we should consider the following cases: $A \vdash_T C$ (so C is justified by A) and $B \vdash_T C$ (so C is justified by B). Consider the first case. We claim that C is a C-interpolant. Indeed, (C1) and (C2) are obvious since I coincides with C. (C3) becomes $\neg C, A \vdash_T C$ and is implied by $A \vdash_T C$. Finally, (C4) becomes $C, B \vdash_T C$ and holds trivially.

For the second case we claim that $\neg C$ is a C-interpolant. Indeed, (C1) and (C2) are obvious as in the previous case. (C3) becomes $\neg C, A \vdash_T \neg C$ and is trivial. Finally, (C4) becomes $\neg C, B \vdash_T C$ and is implied by $B \vdash_T C$.

Now suppose that Π consists of more than one formula. Consider the last inference of the derivation

$$\frac{C_1 \quad \cdots \quad C_n}{C}\ .$$

Let S be the set of formulas $\{C_1, \ldots, C_n\}$. Let us consider the following three cases.

1. S contains a dirty formula and $S \subseteq \mathcal{L}_A$ (note that in this case C is justified by A);
2. S contains a dirty formula and $S \subseteq \mathcal{L}_B$ (in this case C is justified by B).
3. $S \subseteq \mathcal{L}$ (that is, all premises of the inference are clean);

By the property (L1) of local derivations, these three cases cover all possibilities. We will show how to build a C-interpolant in each of the cases.

Case 1 ***(C is justified by A).*** Consider the largest sub-derivation Π' of Π deriving C and satisfying Lemma 1. This sub-derivation has zero or more clean leaves $C_1, \ldots, C_n$ justified by B and one or more leaves in the language $\mathcal{L}_A$ implied by A. Without loss of generality we assume that there is exactly one leaf D of the latter kind (if there is more than one, we can take their conjunction as D). By the soundness property of derivations we have $C_1, \ldots, C_n, D \vdash_T C$, which implies $C_1, \ldots, C_n, A \vdash_T C$. By the induction hypothesis, for all $j = 1, \ldots, n$ one can build a C_j-interpolant I_j of A and B satisfying the conditions of the lemma. We claim that

$$I \stackrel{\text{def}}{=} (C_1 \vee I_1) \wedge \ldots \wedge (C_n \vee I_n) \wedge \neg(C_1 \wedge \ldots \wedge C_n)$$

is a C-interpolant of A and B. We have to prove $\neg C, A \vdash_T I$ and $I, B \vdash_T C$. Since each I_j is a C_j-interpolant of A and B, we have $A \vdash_T C_j \vee I_j$, for all $j = 1, \ldots, n$.

In addition, we have $C_1, \ldots, C_n, A \vdash_T C$, and so $\neg C, A \vdash_T \neg(C_1 \wedge \ldots \wedge C_n)$. This proves $\neg C, A \vdash_T I$.

It remains to prove $I, B \vdash_T C$. We will prove a stronger property $I, B \vdash_T \bot$. By the induction hypothesis we have $I_j, B \vdash_T C_j$, for all $j = 1, \ldots, n$. But $I \vdash I_j \vee C_j$, hence $I, B \vdash_T C_j$ for all $j = 1, \ldots, n$. Finally, we have $I \vdash \neg(C_1 \wedge \ldots \wedge C_n)$, which yields $I, B \vdash_T \bot$.

Note that I in this proof is a boolean combination of $C_1, \ldots, C_n, I_1, \ldots, I_n$, which implies that it is a boolean combination of conclusions of symbol-eliminating inferences.

Case 2 ***(C is justified by B).*** Consider the largest sub-derivation Π' of Π deriving C and satisfying Lemma 1. This sub-derivation has zero or more clean leaves $C_1, \ldots, C_n$ justified by A and one or more leaves in the language $\mathcal{L}_B$ implied by B. As in the previous case, we assume that there is exactly one leaf D of the latter kind. By the induction hypothesis, for all $j = 1, \ldots, n$ one can build a C_j-interpolant I_j of A and B satisfying the conditions of the lemma. We claim that

$$I \stackrel{\text{def}}{=} (C_1 \vee I_1) \wedge \ldots \wedge (C_n \vee I_n)$$

is a C-interpolant of A and B. The proof is similar to case 1.

Case 3 ***(All Premises are Clean).*** In this case one can build a C-interpolant as in the previous cases by replacing D by $\top$. □

This lemma implies the following key result.

Theorem 4. Let Π be a closed local AB-refutation. Then one can extract from Π in linear time a reverse interpolant I of A and B. This reverse interpolant is a boolean combination of conclusions of symbol-eliminating inferences of Π. □

When we speak about "linear time" in this theorem we mean that we build a dag representation of the interpolant. As a corollary of this theorem we obtain the following one.

Theorem 5. Let Π be a closed local AB-refutation. Then one can extract from Π in linear time a reverse interpolant I of A and B. This interpolant is ground if all formulas in Π are ground. □

If all formulas in the derivation are universal, the reverse interpolant is a boolean combination of universal formulas but not necessarily a universal formula. Example 1 shows that this result cannot be improved, since there may be no universal reverse interpolant even when all formulas in the local derivation are universal.

It is interesting to consider the use of interpolants in verification in view of this theorem. Most papers on the use of interpolation outside of propositional logic do not use the interpolant per se, but use the set of atoms occurring in some interpolant extracted from a proof [8]. Theorem 4 says that this set of atoms is exactly the set of atoms occurring in the conclusions of symbol-eliminating inferences. There is also a strong evidence that symbol elimination is a key to finding loop invariants [13]. This poses an interesting problem of studying symbol elimination in various theories. More precisely, given formulas A and B, we are interested in formulas C such that C is a conclusion of a symbol-eliminating inference in a derivation from A and B.

4 Superposition and Interpolation

In this section we investigate extraction of interpolants from superposition refutations. This is motivated by potential applications of such interpolants to verification. We consider only ground formulas and two kinds of superposition calculus: the standard one and its extension LASCA from [12].

We have already pointed out using Example 1 that the result on extracting interpolants from superposition proofs in [18] is incorrect. The flaw in the proofs of this paper is as follows. It cites (without proof) the following property of the superposition calculus: if a clause C is implied by a saturated set of clauses S, then C is implied by a subset of S consisting of clauses strictly smaller than C. This property does not hold even if we use the subset consisting of clauses smaller than or equal to C. For example, the set consisting of a single clause $f(a) \not\simeq f(b)$ is saturated and $a \not\simeq b$ follows from it but $a \not\simeq b$ is strictly smaller than $f(a) \not\simeq f(b)$ in all simplification orderings.

In the rest of this section, unless stated otherwise, we assume to deal with ground formulas only. A detailed description of the superposition calculus can be found in [20], see [20,12] for more details. The calculus LASCA [12] for ground linear rational arithmetic and uninterpreted functions is given in Figure 1. It is a two-sorted theory and uses the symbol $=$ for equality on the sort of rationals and the symbol $\simeq$ for equality on the second sort. In all rules we have the condition $l \succ r$. The standard superposition calculus for ground formulas can be obtained from LASCA by

- removing all arithmetical rules;
- replacing equality modulo AC $=_{AC}$ by the syntactic equality.
- Replacing the $\bot$-elimination rule with the equality resolution rule

$$\frac{s \not\simeq s \vee C}{C} \ .$$

Let us call a simplification ordering $\succ$ on ground terms *separating* if each dirty ground term is greater in this ordering than any clean ground term. It is not hard to argue that such orderings exists. For example, both the Knuth-Bendix [11] and the LPO [9] families of orderings can be made into separating orderings using the following ideas. In the case of KBO one should use ordinal-based KBO of [14], make every dirty symbol of weight w have weight $w\omega$ and preserve the weights of clean symbols. Then the weight of every dirty ground term will be at least ω and the weight of every clean ground term will be finite. Likewise, for LPO we should make all dirty symbols have higher precedence than any clean symbol. Note that in practice KBO (with finite weights) and LPO and the only orderings used in theorem provers.[1]

Theorem 6. If $\succ$ is separating, then every AB-derivation in LASCA is local.

Proof. We will prove by induction that every inference in an AB-derivation is local. Note that this implies that every clause in this inference has either only symbols in $\mathcal{L}_A$ or only symbols in $\mathcal{L}_B$.

[1] Vampire [21] uses KBO where predicates may have weights greater than ω.

Ordered Paramodulation:

$$\frac{C \vee l \simeq r \quad L[l']_p \vee D}{C \vee D \vee L[r]_p}$$

(i) $l =_{AC} l'$,
(ii) $(l \simeq r) \succ C$.

Equality Factoring:

$$\frac{C \vee t' \simeq s' \vee t \simeq s}{C \vee s \not\simeq s' \vee t \simeq s'}$$

(i) $t =_{AC} t'$,
(ii) $(t \simeq s) \succeq C \vee t' \simeq s'$.

Gaussian Elimination:

$$\frac{C \vee l = r \quad L[l']_p \vee D}{C \vee D \vee L[r]_p}$$

(i) $l =_{AC} l'$,
(ii) $(l = r) \succ C$.

Theory Equality Factoring:

$$\frac{C \vee l' = r' \vee l = r}{C \vee r > r' \vee r' > r \vee l = r'}$$

(i) $l =_{AC} l'$,
(ii) $(l = r) \succeq C \vee l' = r'$.

Fourier-Motzkin Elimination:

$$\frac{C \vee l > r \quad -l' > r' \vee D}{C \vee D \vee -r' > r}$$

(i) $l =_{AC} l'$,
(ii) $(l > r) \succ C$,
(iii) there is no $l'' > r'' \in C$ such that $l'' =_{AC} l$
(iv) $(-l' > r') \succ D$
(v) there is no $-l'' > r'' \in D$ such that $l'' =_{AC} l$.

Inequality Factoring (InF1):

$$\frac{C \vee \pm l' > r' \vee \pm l > r}{C \vee r > r' \vee \pm l > r}$$

(i) $l =_{AC} l'$,
(ii) $(\pm l > r) \succeq C \vee \pm l' > r'$.

Inequality Factoring (InF2):

$$\frac{C \vee \pm l' > r' \vee \pm l > r}{C \vee r' > r \vee \pm l > r'}$$

(i) $l =_{AC} l'$,
(ii) $(\pm l > r) \succeq C \vee \pm l' > r'$.

$\bot$*-Elimination:*

$$\frac{C \vee \bot}{C}$$

(i) C contains only $\top, \bot$ literals.

Fig. 1. Linear Arithmetic Superposition Calculus (LASCA) for ground clauses

We will only consider the ordered paramodulation rule:

Ordered Paramodulation:

$$\frac{C \vee l \simeq r \quad L[l']_p \vee D}{C \vee D \vee L[r]_p} \qquad \begin{array}{l}\text{(i) } l =_{AC} l',\\ \text{(ii) } (l \simeq r) \succ C.\end{array}$$

The proof for all other rules is similar.

By the induction hypothesis both premises have either only symbols in $\mathcal{L}_A$ or only symbols in $\mathcal{L}_B$. Note that if the left premise is clean, then the inference is local: indeed, all symbols occurring in the conclusion also occur in premises. Suppose that the left premise is dirty. Without loss of generality we assume that it belongs to $\mathcal{L}_A$. Note that the conditions $l \succ r$ and $(l \simeq r) \succ C$ guarantee that l is the greatest term in the left premise: this implies that l contains a dirty symbol. But the right premise contains a term l' that is AC-equal to l, so l' contains this symbol too. Hence, the right premise cannot contain dirty symbols occurring in B and so the inference is local. □

Theorems 4 and 6 yield a new algorithm for generating interpolants in the combination of linear rational arithmetic and superposition calculus. Namely, one should search for proofs in LASCA using a separating ordering and then extract interpolants from them.

5 Interpolation and Invariant Generation

In this section we discuss the use of interpolants in invariant generation for proving loop properties. For proving an assertion for a program containing loops one needs to find loop invariants. Jhala and McMillan [8] propose the following technique for extracting predicates used in loop invariants. Suppose that $P(\bar{s})$ is a post-condition to be proved, where $\bar{s}$ is the set of state variables of the program; the program variables are also considered as constants in a first-order language. One can generate one or more loop unrollings and consider the corresponding set of paths, each path leading from an initial state to a state where $P(\bar{s})$ must hold. Take any such path π, which uses n transitions represented by quantifier-free formulas

$$T_1(\bar{s}, \bar{s}'), \ldots, T_n(\bar{s}, \bar{s}').$$

Let $Q(\bar{s})$ be a formula representing the set of initial states. Then one can write the (ground) formula

$$Q(\bar{s}_0) \wedge T_1(\bar{s}_0, \bar{s}_1) \wedge \ldots \wedge T_n(\bar{s}_{n-1}, \bar{s}_n) \wedge \neg P(\bar{s}_n), \tag{1}$$

expressing that we can follow the path π from an initial state to a state satisfying the negation of $P(\bar{s}_n)$. If (1) is satisfiable, then the path gives a counterexample for the post-condition $P(\bar{s}_n)$. Otherwise (1) is unsatisfiable, so it should have some kind of refutation. For simplicity consider the case when $n = 1$, then (1) becomes

$$Q(\bar{s}_0) \wedge T_1(\bar{s}_0, \bar{s}_1) \wedge \neg P(\bar{s}_1).$$

This is a formula containing constants referring to two states. Note that the reverse interpolant of $Q(\bar{s}_0) \wedge T_1(\bar{s}_0, \bar{s}_1)$ and $\neg P(\bar{s}_1)$ is a state formula using only the constants

$\bar{s}_1$. It is proposed to generate such a reverse interpolant from the proof and try to build an invariant state formula from the collection of atoms occurring in interpolants for all paths. It turns out that this approach works well in practice, however notice that what is used in the invariant is not the interpolant but only atoms occurring in it. We know that, for local proofs, these atoms are exactly those occurring in conclusions of symbol-eliminating inferences. Jhala and McMillan [8] define a more general notion of interpolant referring to a sequence of formulas; we can reformulate all results of this section for this more general notion too.

Let us make a few observations suggesting that symbol elimination is another interesting property in this context. Note that [8] defines signatures in such a way that all symbols apart from the state constants are clean. Therefore a reverse interpolant is always sought for a pair of formulas $A(\bar{s}_0, \bar{s}_1)$ and $B(\bar{s}_1)$, where $\bar{s}_0$ are the only dirty symbols.

Theorem 7. The formula I defined as $\exists \bar{x} A(\bar{x}, \bar{s}_1)$ is a reverse interpolant of $A(\bar{s}_0, \bar{s}_1)$ and $B(\bar{s}_1)$. Moreover, it is the strongest interpolant of these formulas, that is, for every other interpolant I' we have $I \vdash_T I'$. □

The strongest reverse interpolant I from this theorem is not a ground formula. However, if T has quantifier elimination, one can obtain an equivalent ground formula by quantifier elimination. A similar observation is made by Kapur et.al. [10]. This implies, for example, that the Fourier-Motzkin variable elimination procedure gives interpolants for the theory of linear rational arithmetic. One can also note that the interpolant I of Theorem 7 represents the image of the set of all states under the transition having A as its symbolic representation. The interesting point is that in practice the image turned out to be not a good formula for generating invariants, see e.g. [16], so it is only interpolants extracted from refutations that proved to be useful in verification and model checking.

Let us also point out that an interesting related notion has recently been introduced by Gulwani and Musuvathi [6]. Namely, they call a *cover* of $\exists \bar{x} A(\bar{x}, \bar{s}_1)$ the strongest ground formula C such that $\exists \bar{x} A(\bar{x}, \bar{s}_1) \vdash_T C$. In general covers do not necessarily exist. Evidently, in a theory with quantifier elimination every formula has a cover. Gulwani and Musuvathi [6] show that there are theories having no quantifier elimination property but having the cover property, that is every existential formula has a cover. It is not hard to argue that the notion of cover captures all ground interpolants:

Theorem 8. Suppose that $\exists \bar{x} A(\bar{x}, \bar{s}_1)$ has a cover I. If $A(\bar{s}_0, \bar{s}_1)$ and $B(\bar{s}_1)$ have a ground interpolant I', then I is also a ground interpolant of these formulas and we have $I \vdash I'$. □

6 Related Work

Most of the existing approaches for generating interpolants, for example Henzinger et.al. [7], McMillan [17], Jhala and McMillan [8] require explicit construction of resolution proofs in the combined theory of linear arithmetic with uninterpreted function symbols. Interpolants are then derived from these proofs. Their method of extraction is quite complex compared to ours, especially when equality reasoning is involved.

Cimatti, Griggio and Sebastiani [2] show how to extract interpolants in the combined theory of linear arithmetic with uninterpreted function symbols. The interpolants are extracted from proofs produced by an SMT solver.

Unlike the aforementioned works, Rybalchenko and Sofronie-Stokkermans [22] do not require a priori constructed proofs. Instead, they reduce the problem of generating interpolants in the combination of the theories of linear arithmetic with uninterpreted functions to solving constraints in these theories in the hierarchical style of [23]. An approach for constructing interpolants in combinations of theories over disjoint signatures is proposed by Yorsh and Musuvathi [25]. Note that [25,22] do not present experiments of whether interpolants generated using their methods are useful for verification.

Kapur, Majumdar and Zarba [10] discuss connections of interpolation to quantifier elimination. Some of their results have been cited here.

7 Conclusion

Our results suggest that local proofs and symbol-eliminating inferences can be an interesting alternative to interpolation. Note that one can search for local proofs even for theories not having the interpolation property. For example, the theory of arrays does not have this property [10] but there exists a simple axiomatisation of arrays that can be used in a superposition prover [24,4,5]. This would give an (incomplete) procedure for generating interpolants or finding symbol-eliminating proofs in the combination of the theories of uninterpreted functions, linear rational arithmetic and arrays. The procedure adds an axiomatisation of arrays to the non-ground version of LASCA and searches for local proofs in the resulting theory. We believe it is an interesting direction for future research.

Acknowledgments. We thank Konstantin Korovin for stimulating discussions.

References

1. Bachmair, L., Ganzinger, H.: Resolution Theorem Proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 2, vol. I, pp. 19–99. Elsevier Science, Amsterdam (2001)
2. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient Interpolant Generation in Satisfiability Modulo Theories. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 397–412. Springer, Heidelberg (2008)
3. Craig, W.: Three uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. Journal of Symbolic Logic 22(3), 269–285 (1957)
4. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Deciding Extensions of the Theory of Arrays by Integrating Decision Procedures and Instantiation Strategies. In: Fisher, M., van der Hoek, W., Konev, B., Lisitsa, A. (eds.) JELIA 2006. LNCS, vol. 4160, pp. 177–189. Springer, Heidelberg (2006)
5. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Decision Procedures for Extensions of the Theory of Arrays. Ann. Math. Artif. Intell. 50(3-4), 231–254 (2007)
6. Gulwani, S., Musuvathi, M.: Cover Algorithms and Their Combination. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 193–207. Springer, Heidelberg (2008)

7. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from Proofs. In: Proc. of POPL, pp. 232–244 (2004)
8. Jhala, R., McMillan, K.L.: A Practical and Complete Approach to Predicate Refinement. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
9. Kamin, S., Lévy, J.-J.: Two Generalizations of the Recursive Path Ordering (January 1980) (unpublished)
10. Kapur, D., Majumdar, R., Zarba, C.G.: Interpolation for Data Structures. In: Young, M., Devanbu, P.T. (eds.) SIGSOFT FSE, pp. 105–116. ACM, New York (2006)
11. Knuth, D., Bendix, P.: Simple Word Problems in Universal Algebras. In: Leech, J. (ed.) Computational Problems in Abstract Algebra, pp. 263–297. Pergamon Press, Oxford (1970)
12. Korovin, K., Voronkov, A.: Integrating Linear Arithmetic into Superposition Calculus. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 223–237. Springer, Heidelberg (2007)
13. Kovacs, L., Voronkov, A.: Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In: Proc. of FASE (to appear, 2009)
14. Ludwig, M., Waldmann, U.: An Extension of the Knuth-Bendix Ordering with LPO-Like Properties. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS, vol. 4790, pp. 348–362. Springer, Heidelberg (2007)
15. Lyndon, R.C.: An Interpolation Theorem in the Predicate Calculus. Pacific Journal of Mathematics 9, 129–142 (1959)
16. McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
17. McMillan, K.L.: An Interpolating Theorem Prover. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 16–30. Springer, Heidelberg (2004)
18. McMillan, K.L.: Quantified Invariant Generation Using an Interpolating Saturation Prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008)
19. Motohashi, N.: Equality and Lyndon's Interpolation Theorem. Journal of Symbolic Logic 49(1), 123–128 (1984)
20. Nieuwenhuis, R., Rubio, A.: Paramodulation-Based Theorem Proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 7, vol. I, pp. 371–443. Elsevier Science, Amsterdam (2001)
21. Riazanov, A., Voronkov, A.: The Design and Implementation of Vampire. AI Communications 15(2-3), 91–110 (2002)
22. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint Solving for Interpolation. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 346–362. Springer, Heidelberg (2007)
23. Sofronie-Stokkermans, V.: Interpolation in local theory extensions. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS, vol. 4130, pp. 235–250. Springer, Heidelberg (2006)
24. Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.: A Decision Procedure for an Extensional Theory of Arrays. In: LICS (2001)
25. Yorsh, G., Musuvathi, M.: A Combination Method for Generating Interpolants. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS, vol. 3632, pp. 353–368. Springer, Heidelberg (2005)

Complexity and Algorithms for Monomial and Clausal Predicate Abstraction

Shuvendu K. Lahiri and Shaz Qadeer

Microsoft Research

Abstract. In this paper, we investigate the asymptotic complexity of various predicate abstraction problems relative to the asymptotic complexity of checking an annotated program in a given assertion logic. Unlike previous approaches, we pose the predicate abstraction problem as a decision problem, instead of the traditional inference problem. For assertion logics closed under weakest (liberal) precondition and Boolean connectives, we show two restrictions of the predicate abstraction problem where the two complexities match. The restrictions correspond to the case of *monomial* and *clausal* abstraction. For these restrictions, we show a symbolic encoding that reduces the predicate abstraction problem to checking the satisfiability of a single formula whose size is polynomial in the size of the program and the set of predicates. We also provide a new iterative algorithm for solving the *clausal* abstraction problem that can be seen as the dual of the `Houdini` algorithm for solving the *monomial* abstraction problem.

1 Introduction

Predicate abstraction [8] is a method for constructing inductive invariants for programs or transition systems over a given set of predicates $\mathcal{P}$. It has been an enabling technique for several automated hardware and software verification tools. SLAM [1], BLAST [11] use predicate abstraction to construct invariants for sequential software programs. Predicate abstraction has also been used in tools for verifying hardware descriptions [12] and distributed algorithms [14]. Although predicate abstraction is an instance of the more general theory of abstract interpretation [5], it differs from most other abstract interpretation techniques (e.g. for numeric domains [6], shape analysis [17]) in that it does not require a fixed abstract domain; it is parameterized by decision procedures for the assertion logic in which the predicates are expressed.

Most previous work on predicate abstraction has been concerned with *constructing* an inductive invariant over a set of predicates $\mathcal{P}$ using abstract interpretation techniques. A (finite) abstract domain over the predicates is defined and the program is executed over the abstract domain until the set of abstract states do not change. At that point, the abstract state can be shown to be an inductive invariant. These techniques are usually *eager* in that the inductive invariant is computed without recourse to the property of interest. The property directedness is somewhat recovered using various forms of refinements

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 214–229, 2009.

(e.g. counterexample-guided [13,3,11], or proof-guided [10] refinements) by varying the set of predicates. However, these techniques do not usually address the problem for the case of a fixed set of predicates.

In this paper, we study the problem of predicate abstraction as a decision problem. For a given program $Prog(pre, post, body)$

$$\begin{array}{l}\{pre\}\\ \mathsf{while}\ (*)\ \mathsf{do}\ body\\ \{post\}\end{array}$$

where *pre* and *post* are assertions, *body* is loop-free code, and a set of predicates $\mathcal{P}$, we are interested in the question:

> Does there exist a loop invariant I over $\mathcal{P}$ such that program *Prog* can be proved correct?

We define this decision problem to be $InferPA(Prog, \mathcal{P})$. By posing the problem as a decision problem, we do not have to adhere to any particular way to *construct* the loop invariant I (say using abstract interpretation), and it allows us to study the complexity of the problem. Besides, the problem formulation requires us to search for only those I that can prove the program. This problem formulation has been first proposed in the context of the annotation inference problem in `ESC/Java` [7] for a restricted case, and more recently by Gulwani et al. [9]. Although the latter has the same motivation as our work, they do not study the complexity of the predicate abstraction problems.

In this paper, we study the asymptotic complexity of the decision problem $InferPA(Prog, \mathcal{P})$ relative to the decision problem $Check(Prog, I)$ which checks if I is a loop invariant that proves the program correct. Throughout the paper, we assume that the assertion logic of assertions in *pre*, *post* and predicates in $\mathcal{P}$ is closed under *weakest (liberal) precondition* predicate transformer *wp*, and also closed under Boolean connectives. The assertion logic determines the logic in which $Check(Prog, I)$ is expressed. We are also most interested in logics for which the decision problem is **Co-NP** complete— this captures a majority of the assertion logics for which efficient decision procedures have been implemented using Satisfiability Modulo Theories (SMT) solvers [20]. In addition to propositional logic, this includes the useful theories of uninterpreted functions, arithmetic, select-update arrays, inductive datatypes, and more recently logics about linked lists [15] and types in programs [4].

For such assertion logics, we show that if checking $Check(Prog, I)$ is in **PSPACE**, then checking $InferPA(Prog, \mathcal{P})$ is **PSPACE** complete. We also study the problem of template abstraction [19,21] where the user provides a formula J with some free Boolean variables $\mathcal{W}$, and is interested in finding whether there is a valuation $\sigma_{\mathcal{W}}$ of $\mathcal{W}$ such that $Check(Prog, J[\sigma_{\mathcal{W}}/\mathcal{W}])$ is true. We call this decision problem $InferTempl(Prog, J, \mathcal{W})$, and show $InferTempl(Prog, J, \mathcal{W})$ is $\mathbf{\Sigma_2^P}$ complete ($\mathbf{NP^{NP}}$ complete), when $Check(Prog, I)$ is in **Co-NP**.

Given that the general problem of predicate or template abstraction can be much more complex than checking $Check(Prog, I)$, we focus on two restrictions:

- *InferMonome*$(Prog, \mathcal{P})$: Given a set of predicates $\mathcal{P}$, does there exist a $\mathcal{R} \subseteq \mathcal{P}$ such that *Check*$(Prog, \bigwedge_{p \in \mathcal{R}} p)$ is true, and
- *InferClause*$(Prog, \mathcal{P})$: Given a set of predicates $\mathcal{P}$, does there exist a $\mathcal{R} \subseteq \mathcal{P}$ such that *Check*$(Prog, \bigvee_{p \in \mathcal{R}} p)$ is true.

These problems can also be seen as restrictions on the template abstraction problem. For example, the problem *InferMonome*$(Prog, \mathcal{P})$ can also be interpreted as *InferTempl*$(Prog, J, \mathcal{W})$ where J is restricted as $(\bigwedge_{p \in \mathcal{P}} b_p \implies p)$ and $\mathcal{W} = \{b_p | p \in \mathcal{P}\}$.

We show that for both *InferMonome*$(Prog, \mathcal{P})$ and *InferClause*$(Prog, \mathcal{P})$, the complexity of the decision problem matches the complexity of *Check*$(Prog, I)$. For instance, when the complexity of *Check*$(Prog, I)$ is **Co-NP** complete, both *InferMonome*$(Prog, \mathcal{P})$ and *InferClause*$(Prog, \mathcal{P})$ are **Co-NP** complete. We obtain these results by providing a symbolic encoding of both problems into logical formulas, such that the logical formulas are satisfiable if and only if the inference problems return false. The interesting part is that these logical formulas are polynomially bounded in *Prog* and $\mathcal{P}$.

The symbolic encoding of the inference problems also provides algorithms to answer these decision problems, in addition to establishing the complexity results. In the process, we also describe a new iterative algorithm for checking *InferClause*$(Prog, \mathcal{P})$ that can be seen as the dual of an existing algorithm `Houdini` [7] that checks *InferMonome*$(Prog, \mathcal{P})$.

2 Background

We describe some background on programs and their correctness, assertion logics, weakest liberal preconditions, and motivate the inference problems.

2.1 A Simple Programming Language for Loop-Free Programs

Figure 1 describes a simple programming language SimpPL for loop-free programs. The language supports scalar variables *Scalars* and mutable maps or arrays *Maps*. The language supports arithmetic operations on scalar expressions *Expr* and *select-update* reasoning for array expressions *MapExpr*. The symbols **sel** and **upd** are interepreted symbols for selecting from or updating an array. The operation havoc assigns a type-consistent (scalar or map) arbitrary value to its argument. $s; t$ denotes the sequential composition of two statements s and t, $s \diamond t$ denotes a non-deterministic choice to either execute statements in s or t. This statement along with the assume models conditional statements. For example, the statement if (e) $\{s\}$ is desugared into $\{\mathsf{assume}\ e; s\} \diamond \mathsf{assume}\ \neg e$.

The assertion language in *Formula* is extensible and contains the theories for equality, arithmetic, arrays, and is closed under Boolean connectives. Any formula $\phi \in$ *Formula* can be interpreted as a set of states of a program that satisfy ϕ. For any $s \in$ *Stmt*, and $\phi \in$ *Formula*, the *weakest liberal precondition* $wp(s, \phi)$ corresponds to a formula such that from any state in $wp(s, \phi)$, the statement s does not fail any assertions and any terminating execution ends in

$$
\begin{array}{lll}
\mathsf{x}, \mathsf{y} & \in \mathit{Scalars} & \\
\mathsf{X}, \mathsf{Y} & \in \mathit{Maps} & \\
e & \in \mathit{Expr} & ::= \mathsf{x} \mid c \mid e \pm e \mid \mathbf{sel}(E, e) \\
E & \in \mathit{MapExpr} & ::= \mathsf{X} \mid \mathbf{upd}(E, e, e) \\
s, t & \in \mathit{Stmt} & ::= \mathsf{skip} \mid \mathsf{assert}\ \phi \mid \mathsf{assume}\ \phi \mid \mathsf{x} := e \mid \mathsf{X} := E \\
 & & \quad\ \mathsf{havoc}\ \mathsf{x} \mid \mathsf{havoc}\ \mathsf{X} \mid s; s \mid s \diamond s \\
 & & \\
\phi, \psi & \in \mathit{Formula} & ::= e \leq e \mid \phi \wedge \phi \mid \neg\phi \mid \ldots
\end{array}
$$

Fig. 1. A simple programming language SimpPL and an extensible assertion logic *Formula*

$$
\begin{array}{ll}
wp(\mathsf{skip}, \phi) & = \phi \\
wp(\mathsf{assert}\ \psi, \phi) & = \psi \wedge \phi \\
wp(\mathsf{assume}\ \psi, \phi) & = \psi \implies \phi \\
wp(\mathsf{x} := e, \phi) & = \phi[e/\mathsf{x}] \\
wp(\mathsf{X} := E, \phi) & = \phi[E/\mathsf{X}]
\end{array}
\qquad
\begin{array}{ll}
wp(\mathsf{havoc}\ \mathsf{x}, \phi) & = \phi[\mathsf{v}/\mathsf{x}] \\
wp(\mathsf{havoc}\ \mathsf{X}, \phi) & = \phi[\mathsf{V}/\mathsf{X}] \\
wp(s; t, \phi) & = wp(s, wp(t, \phi)) \\
wp(s \diamond t, \phi) & = wp(s, \phi) \wedge wp(t, \phi)
\end{array}
$$

Fig. 2. Weakest liberal precondition for the logic without any extensions. Here v and V represent fresh symbols.

a state satisfying ϕ. For our assertion logic (without any extensions), Figure 2 shows the *wp* for statements in the programming language. For more complex extensions to the assertion logic (e.g. [15]), the rule for $wp(\mathsf{X} := E, \phi)$ is more complex. Although applying *wp* can result in an exponential blowup in the size of a program, standard methods generate a linear-sized formula that preserves validity by performing static single assignment (SSA) and introducing auxiliary variables [2].

We say that the assertion logic is *closed under wp* (for SimpPL) when for any $\phi \in \mathit{Formula}$ and for any $s \in \mathit{Stmt}$, $wp(s, \phi) \in \mathit{Formula}$. In the rest of the paper, we will abstract from the details of the particular assertion logic, with the following restrictions on the assertion logic:

- the assertion logic is closed under Boolean connectives (i.e. subsumes propositional logic).
- the assertion logic is closed under *wp*.
- *wp* distributes over $\wedge$, i.e., $wp(s, \phi \wedge \psi) \equiv wp(s, \phi) \wedge wp(s, \psi)$.

A *model* $\mathcal{M}$ assigns a type-consistent valuation to symbols in a formula. Any model assigns the standard values for interpreted symbols such as $=$, $+$, $-$, **sel**, **upd**, and assigns an integer value to symbols in *Scalars* and function values to symbols in *Maps*. For a given model $\mathcal{M}$, we say that the model satisfies a formula $\phi \in \mathit{Formula}$ (written as $\mathcal{M} \models \phi$) if and only if the result of evaluating ϕ in $\mathcal{M}$ is true; in such a case, we say that ϕ is *satisfiable*. We use $\models \phi$ to denote that ϕ is *valid* when ϕ is satisfiable for any model.

Definition 1. *For any $\phi, \psi \in \mathit{Formula}$ and $s \in \mathit{Stmt}$, the Floyd-Hoare triple $\{\phi\}\ s\ \{\psi\}$ holds if and only if the logical formula $\phi \implies wp(s, \psi)$ is valid.*

Intuitively, the Floyd-Hoare triple $\{\phi\}\ s\ \{\psi\}$ captures the specification that from a state satisfying ϕ, no execution of s fails any assertions and every terminating execution ends in a state satisfying ψ. Given our assumptions on the assertion logic *Formula*, checking the correctness of a program in SimpPL reduces to checking validity in the assertion logic.

2.2 Loops and Loop Invariants

Having defined the semantics of loop-free code blocks, consider the following class of programs $Prog(pre, post, body)$ below (ignore the annotation $\{\mathsf{inv}\ I\}$ initially) where $pre, post \in Formula$ and $body \in Stmt$:

$$\begin{array}{l} \{pre\} \\ \mathsf{while}\ (*)\{\mathsf{inv}\ I\}\ \mathsf{do}\ body \\ \{post\} \end{array}$$

Since this program can have unbounded computations due to the while loop, generating a verification condition requires a loop invariant. This is indicated by inv annotation. The loop invariant I is a formula in *Formula*. The Floyd-Hoare triple for *Prog* holds if and only if there exists a loop invariant I such that the three formula are valid:

$$\begin{array}{lcl} \models pre & \implies & I \\ \models I & \implies & wp(body, I) \\ \models I & \implies & post \end{array}$$

For any such program $Prog(pre, post, body)$, we define $Check(Prog, I)$ to return true if and only if all the three formulas are valid. Intuitively, $Check(Prog, I)$ *checks* whether the supplied loop invariant I holds on entry to the loop, is preserved by an arbitrary loop iteration and implies the postcondition.

We now define the decision problem that corresponds to *inferring* the loop invariant I for *Prog*. For a given program $Prog(pre, post, body)$, $Infer(Prog)$ returns true if and only if there exists a formula $I \in Formula$ such that $Check(Prog, I)$ is true.

Although $Check(Prog, I)$ is efficiently decidable for our programming language for a rich class of assertions (including the one shown in Figure 1), checking $Infer(Prog)$ is undecidable even when *Prog* consists of only scalar integer variables and include arithmetic operations — one can encode the reachability problem for a two counter machine, checking which is undecidable. Therefore, most approaches search for I within some restricted space. Predicate abstraction is one such technique that searches for an I within a finite space.

3 Complexity of Predicate and Template Abstraction

In this section, we study the complexity of two inference techniques (relative to the complexity of $Check(Prog, I)$) that search for loop invariants over a finite space:

1. The first technique is based on *predicate abstraction*, where the loop invariant is searched over Boolean combination of an input set of predicates.
2. The second technique is based on *templates*, where the loop invariant is searched over the valuations of a set of free Boolean variables in a candidate template assertion.

3.1 Predicate Abstraction

Predicate abstraction [8], an instance of the more general theory of abstract interpretation [5], is a mechanism to make the $\mathit{Infer}(\mathit{Prog})$ problem more tractable by searching for I over restricted formulas. In predicate abstraction, in addition to *Prog*, we are given a set of predicates $\mathcal{P} = \{p_1, \ldots, p_n\}$ where $p_i \in \mathit{Formula}$. Throughout this paper, we assume that the set $\mathcal{P}$ is closed under negation, i.e., if $p \in \mathcal{P}$ then $\neg p \in \mathcal{P}$. Instead of looking for a loop invariant I over arbitrary formulas, predicate abstraction restricts the search to those formulas that are a Boolean combination (using $\vee$ or $\wedge$) over the predicates in $\mathcal{P}$. More formally, for a program $\mathit{Prog}(\mathit{pre}, \mathit{post}, \mathit{body})$ and a set of predicates $\mathcal{P}$, $\mathit{InferPA}(\mathit{Prog}, \mathcal{P})$ returns true if and only if there exists a formula $I \in \mathit{Formula}$ which is a Boolean combination over $\mathcal{P}$ such that $\mathit{Check}(\mathit{Prog}, I)$ is true.

Theorem 1. *If checking* $\mathit{Check}(\mathit{Prog}, I)$ *is in* **PSPACE** *in the size of Prog and* I*, then checking* $\mathit{InferPA}(\mathit{Prog}, \mathcal{P})$ *is* **PSPACE** *complete in the size of Prog and* $\mathcal{P}$*.*

Proof sketch. Showing that $\mathit{InferPA}(\mathit{Prog}, \mathcal{P})$ is **PSPACE** hard is easy. We can encode the reachability problem for a propositional transition system (which is **PSPACE** complete) into $\mathit{InferPA}(\mathit{Prog}, \mathcal{P})$ by encoding the transition system as *Prog*, and setting $\mathcal{P}$ to be the set of state variables in the transition system.

To show that $\mathit{InferPA}(\mathit{Prog}, \mathcal{P})$ is in **PSPACE**, we will provide a nondeterministic algorithm for checking if $\neg \mathit{post}$ can be reached by an abstract interpretation of the program $\{\mathit{pre}\}$ while $(*)$ do body $\{\mathit{post}\}$ over the set of predicates in $\mathcal{P}$. Moreover, the algorithm will only use polynomial space in *Prog* and $\mathcal{P}$.

The abstract state of the program corresponds to an evaluation of the predicates in $\mathcal{P}$ to $\{\mathsf{true}, \mathsf{false}\}$. The algorithm performs an abstract *run* of size upto $2^{|\mathcal{P}|}$ to determine if $\neg \mathit{post}$ can be reachable starting from *pre*. The nondeterministic algorithm returns false if and only if some abstract run ends up in a state satisfying $\neg \mathit{post}$.

We need to store two successive states in a run and the length of the run, both of which only require linear space over the inputs. Moreover, to check that an abstract state is a successor of another, one needs to make a query to check that there is some concrete transition in *Prog* between the concretizations of the two abstract states — this can be done in **PSPACE** since the decision problem is in **PSPACE**. □

3.2 Template Abstraction

In template abstraction [18,9,21], the user provides a *template* formula $J \in \mathit{Formula}^{\mathcal{W}}$ in addition to $\mathit{Prog}(\mathit{pre}, \mathit{post}, \mathit{body})$, where

- $\mathcal{W} = \{w_1, w_2, \ldots, w_m\}$ is a set of Boolean valued symbols, and
- $\mathit{Formula}^{\mathcal{W}}$ extends *Formula* to contain formulas whose symbols range over both state variables (*Scalars* and *Maps*) and $\mathcal{W}$.

Given J, the goal is to infer a loop invariant I by searching over the different valuations of the symbols in $\mathcal{W}$.

For a set of symbols X, let σ_X denote an *assignment* of values of appropriate types to each symbols in X. For any expression e and an assignment σ_X, $e[\sigma_X/X]$ replaces a symbol $x \in X$ in e with $\sigma_X(x)$. For a program *Prog* and a template $J \in \mathit{Formula}^{\mathcal{W}}$, $\mathit{InferTempl}(\mathit{Prog}, J, \mathcal{W})$ returns true if and only if there exists an assignment $\sigma_{\mathcal{W}}$ to the symbols in $\mathcal{W}$ such that $\mathit{Check}(\mathit{Prog}, J[\sigma_{\mathcal{W}}/\mathcal{W}])$ is true.

Example 1. Consider the simple program $\{x = 0 \wedge y = 10\}$ while $(y \neq 0)$ do $x := x+1; y := y-1;$ $\{x = 10\}$. In our language, the program would be written as $\{x = 0 \wedge y = 10\}$ while $(*)$ do assume $y \neq 0; x := x+1; y := y-1;$ $\{y = 0 \implies x = 10\}$. A potential loop invariant that proves the program is $x+y = 10$. A template J for which $\mathit{InferTempl}(\mathit{Prog}, J, \mathcal{W})$ is true is $(w_1 \implies x + y = 10) \wedge (w_2 \implies x = y)$. Clearly, for the assignment $\sigma_{w_1} = \mathsf{true}$ and $\sigma_{w_2} = \mathsf{false}$, the template is a loop invariant.

The complexity class $\mathbf{\Sigma_2^P}$ contains all problems that can be solved in **NP** using an **NP** oracle.

Theorem 2. *If checking* $\mathit{Check}(\mathit{Prog}, I)$ *is in* **Co-NP** *in the size of Prog and I, then checking* $\mathit{InferTempl}(\mathit{Prog}, J, \mathcal{W})$ *is* $\mathbf{\Sigma_2^P}$ *complete in Prog, J, and* $\mathcal{W}$.

Proof. The problem is in $\mathbf{\Sigma_2^P}$ because we can non-deterministically guess an assignment of $\mathcal{W}$ and check if the resulting J is an inductive invariant. The **NP** oracle used in this case is a checker for $\neg \mathit{Check}(\mathit{Prog}, I)$.

On the other hand, one can formulate $\mathit{InferTempl}(\mathit{Prog}, J, \mathcal{W})$ as the formula $\exists \mathcal{W}. \mathit{Check}(\mathit{Prog}, J)$. Given a quantified Boolean formula (QBF) $\exists X. \forall Y. \phi(X, Y)$ where ϕ is quantifier-free, for which checking the validity is $\mathbf{\Sigma_2^P}$ complete, we can encode it to $\mathit{InferTempl}(\mathit{Prog}, J, \mathcal{W})$, by constructing $\mathit{pre} = \mathit{post} = \mathsf{true}$, $\mathit{body} = \mathsf{skip}$, $\mathcal{W} = X$ and $J = \phi(X, Y)$. □

Having shown that the complexity of both predicate abstraction and template abstraction are considerably harder than checking an annotated program, we will focus on two restricted versions of the abstraction problem for monomials and clauses. As mentioned in the introduction, these problems can be seen as restrictions of either predicate abstraction or template abstraction.

4 Monomial Abstraction

For any set $\mathcal{R} \subseteq \mathcal{P}$, a *monome* over $\mathcal{R}$ is the formula $\bigwedge_{p \in \mathcal{R}} p$. For a set of predicates $\mathcal{P}$, let us define $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ to return true if and only if there exists $\mathcal{R} \subseteq \mathcal{P}$ such that $\mathit{Check}(\mathit{Prog}, \bigwedge_{p \in \mathcal{R}} p)$ is true.

4.1 `Houdini` Algorithm

Figure 3 describes an algorithm *FindInv* for solving the *InferMonome* problem. Initially, ignore the shaded regions of the algorithm. The algorithm iteratively prunes the set of predicates in the invariant starting from $\mathcal{P}$, removing a predicate whenever *RemovesPredicate* holds. The algorithm terminates with a *FAIL* when *FailsCheck* holds, denoting that there is no monomial invariant that satisfies the postcondition *post*. On the other hand, the algorithm terminates with $SUCCESS(\mathcal{R})$ when it finds an invariant. It is easy to see that the procedure *FindInvAux* terminates within a recursion depth of $|\mathcal{P}|$ since its argument $\mathcal{R}$ monotonically decreases along any call chain.

Lemma 1. *The procedure* $FindInv(\mathcal{P})$ *satisfies the following:*

1. $FindInv(\mathcal{P})$ *returns FAIL if and only if* $InferMonome(Prog, \mathcal{P})$ *is* false.
2. *If* $FindInv(\mathcal{P})$ *returns* $SUCCESS(\mathcal{R})$*, then for any* $\mathcal{S} \subseteq \mathcal{P}$ *such that* $Check(Prog, \bigwedge_{p \in \mathcal{S}} p)$*, we have* $S \subseteq \mathcal{R}$.

The algorithm is a variant of the `Houdini` algorithm in `ESC/Java` [7], where *RemovesPredicate* considers each predicate p in isolation instead of the conjunction $(\bigwedge_{q \in \mathcal{S}} q)$. The `Houdini` algorithm solves the $InferMonome(Prog, \mathcal{P})$ problem with at most $|\mathcal{P}|$ number of theorem prover calls. However, this only provides an upper bound on the complexity of the problem. For example, making $|\mathcal{P}|$ number of queries to a **Co-NP** complete oracle (a theorem prover) does not

$$
\begin{array}{lll}
FailsCheck(\mathcal{S}, \mathcal{M}) & \triangleq & \mathcal{M} \models (\bigwedge_{q \in \mathcal{S}} q) \wedge \neg post \\
RemovesPredicate(\mathcal{S}, \mathcal{M}, p) & \triangleq & \vee \mathcal{M} \models pre \wedge \neg p \\
 & & \vee \mathcal{M} \models (\bigwedge_{q \in \mathcal{S}} q) \wedge \neg wp(body, p)
\end{array}
$$

```
proc FindInvAux(R)
  if (exists a model M s.t. FailsCheck(R, M))
    M_guess ← M;
    return FAIL;
  if (exists q ∈ R and model M s.t. RemovesPredicate(R, M, q))
    M_guess ← M;
    Q_guess+1 ← Q_guess ∪ {q};
    guess ← guess + 1;
    return FindInvAux(R \ {q});
  return SUCCESS(R);

proc FindInv(P)
  guess ← 1;
  Q_guess ← {};
  FindInvAux(P);
```

Fig. 3. Procedure to construct either a monomial invariant or a witness to show its absence. The *shaded* lines represent extensions for computing the witness.

establish that the complexity of the inference problem is **Co-NP** complete; it only establishes that the upper bound of the complexity is $\mathbf{P^{NP}}$.

In the next subsection, we provide a model-theoretic justification for the correctness of *FindInv*. Our construction will provide insight into the complexity of $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ relative to the complexity of $\mathit{Check}(\mathit{Prog}, I)$.

4.2 A Model-Theoretic Proof of *FindInv*

For a $\mathit{guess} \in \mathbb{N}$, an indexed set of models $\{\mathcal{M}_i\}_i$, an indexed set of sets of predicates $\{\mathcal{Q}_i\}_i$, we define a predicate $\mathit{NoMonomeInv}(\mathcal{P}, \mathit{guess}, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$, that is true if and only if:

1. $1 \leq \mathit{guess} \leq |\mathcal{P}| + 1$, and
2. $\mathcal{Q}_1 = \{\}$, and
3. For $1 \leq i < \mathit{guess}$, $\mathcal{Q}_{i+1} \setminus \mathcal{Q}_i = \{p_i\}$ for some $p_i \in \mathcal{P}$, and
4. For each $1 \leq i < \mathit{guess}$, and for $p_i \in \mathcal{Q}_{i+1} \setminus \mathcal{Q}_i$, either $\mathcal{M}_i \models \mathit{pre} \wedge \neg p_i$ or $\mathcal{M}_i \models (\bigwedge_{p \in \mathcal{P} \setminus \mathcal{Q}_i} p) \wedge \neg \mathit{wp}(\mathit{body}, p_i)$, and
5. $\mathcal{M}_{\mathit{guess}} \models (\bigwedge_{p \in \mathcal{P} \setminus \mathcal{Q}_{\mathit{guess}}} p) \wedge \neg \mathit{post}$.

The following three lemmas, whose proofs are given in the appendix, establish the connection between the *InferMonome* problem, the *NoMonomeInv* predicate, and the *FindInv* algorithm.

Lemma 2. *If* $\mathit{FindInv}(\mathcal{P})$ *returns FAIL, then* $\mathit{NoMonomeInv}(\mathcal{P}, \mathit{guess}, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$*holds on the values computed by the procedure.*

Lemma 3. *If* $\mathit{NoMonomeInv}(\mathcal{P}, \mathit{guess}, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$ *holds for some guess* $\in \mathbb{N}$*, a set of models* $\mathcal{M}_1, \ldots, \mathcal{M}_{\mathit{guess}}$*, and sets of predicates* $\mathcal{Q}_1, \ldots, \mathcal{Q}_{\mathit{guess}}$*, then* $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ *is* false*.*

Lemma 4. *If* $\mathit{FindInv}(\mathcal{P})$ *returns* $\mathit{SUCCESS}(\mathcal{R})$*, then* $\mathit{Check}(\mathit{Prog}, \bigwedge_{p \in \mathcal{R}} p)$ *is* true*, and therefore* $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ *is* true*.*

The proofs of these lemmas requires the use of the additional shaded lines in the Figure 3, which compute the witness to show that no monomial invariant suffices to prove the program. Together, these three lemmas allow us to conclude that $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ is false iff $\mathit{NoMonomeInv}(\mathcal{P}, \mathit{guess}, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$ holds for some $\mathit{guess} \in \mathbb{N}$, a set of models $\mathcal{M}_1, \ldots, \mathcal{M}_{\mathit{guess}}$, sets of predicates $\mathcal{Q}_1, \ldots, \mathcal{Q}_{\mathit{guess}}$. This fact is used to define the symbolic encoding of the problem $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ in the next subsection.

4.3 Symbolic Encoding and Complexity of *InferMonome*(*Prog*, $\mathcal{P}$)

In this section, we provide a symbolic encoding of $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$. That is, given a program $\mathit{Prog}(\mathit{pre}, \mathit{post}, \mathit{body})$ and a set of predicates $\mathcal{P}$, we will construct a formula $\mathit{SymbInferMonome}(\mathit{Prog}, \mathcal{P})$ which is satisfiable if and only if $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ is false. The formula can be seen as a symbolic encoding

of an iterative version of the *FindInv* algorithm that symbolically captures all executions of the algorithm for any input. Finally, we will use the encoding to relate the complexity of $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ to the complexity of $\mathit{Check}(\mathit{Prog}, I)$.

The following notations are used:

- The set of predicates is $\mathcal{P}$, and $n = |\mathcal{P}|$
- For any formula ϕ, ϕ^i represents the formula with any variable x is replaced with a fresh variable x^i. We will use this for each predicate $p \in \mathcal{P}$, *pre*, *post* and $\mathit{wp}(\mathit{body}, p)$.
- The symbols b_p^j for a predicate p denotes that the predicate p was removed from consideration for the invariant in the j-th iteration.

For each $p \in \mathcal{P}$ and $i \in [1, n+1]$, we define

$$\mathit{present}_p^i \stackrel{\triangle}{=} \textstyle\bigwedge_{j \in [0,i)} \neg b_p^j$$

For each $i \in [1, n]$, we define

$$\begin{array}{rl} \mathit{iter}_i \stackrel{\triangle}{=} & \vee\ \mathit{pre}^i \wedge \bigwedge_{p \in \mathcal{P}} (b_p^i \implies \neg p^i) \\ & \vee \bigwedge_{p \in \mathcal{P}} ((\mathit{present}_p^i \implies p^i) \wedge (b_p^i \implies \neg \mathit{wp}(\mathit{body}, p)^i)) \end{array}$$

For each $i \in [1, n+1]$, we define

$$\mathit{check}_i \stackrel{\triangle}{=} \textstyle\bigwedge_{p \in \mathcal{P}} (\mathit{present}_p^i \implies p^i) \wedge \neg \mathit{post}^i$$

Finally, the desired symbolic encoding $\mathit{SymbInferMonome}(\mathit{Prog}, \mathcal{P})$ is the following formula:

$$\begin{array}{l} \wedge\ 1 \leq \mathit{guess} \leq n+1 \\ \wedge \bigwedge_{p \in \mathcal{P}} \neg b_p^0 \\ \wedge (\bigwedge_{i \in [1,n]} i < \mathit{guess} \implies \Sigma_{p \in \mathcal{P}} b_p^i = 1) \wedge (\bigwedge_{p \in \mathcal{P}} \Sigma_{i \in [1,n]} (i \leq \mathit{guess} \wedge b_p^i) \leq 1) \\ \wedge \bigwedge_{i \in [1,n]} i < \mathit{guess} \implies \mathit{iter}_i \\ \wedge \bigwedge_{i \in [1,n+1]} i = \mathit{guess} \implies \mathit{check}_i \end{array}$$

To get some intuition behind the formula, observe that each of the five conjuncts in this formula resembles closely the five conjuncts in the definition of $\mathit{NoMonomeInv}(\mathcal{P}, \mathit{guess}, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$. The symbols shared across the different $i \in [1, n+1]$ are the b_p^i symbols and *guess*. This is similar to the definition of $\mathit{NoMonomeInv}(\mathcal{P}, \mathit{guess}, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$, where the different models in $\{\mathcal{M}_i\}_i$ only agree on the evaluation of the sets $\mathcal{Q}_i$ and *guess*. The role of the b_p^i variable is precisely to encode the sets $\mathcal{Q}_{i+1}$; $b_p^i = \mathsf{true}$ denotes that $\{p\} = \mathcal{Q}_{i+1} \setminus \mathcal{Q}_i$. The second conjunct denotes that $\mathcal{Q}_1 = \{\}$ where no predicate has been removed. The third conjunct has two parts. The first part $i < \mathit{guess} \implies \Sigma_{p \in \mathcal{P}} b_p^i = 1$ denotes that $\mathcal{Q}_{i+1}$ and $\mathcal{Q}_i$ differ by exactly one predicate, for any $i < \mathit{guess}$. The second part $\bigwedge_{p \in \mathcal{P}} \Sigma_{i \in [1,n]} (i \leq \mathit{guess} \wedge b_p^i) \leq 1$ denotes that a predicate is removed at most once in any one of the *guess* iterations. Similarly, the fourth conjunct justifies the removal of the predicate in $\mathcal{Q}_{i+1} \setminus \mathcal{Q}_i$.

Theorem 3. *The formula* $SymbInferMonome(Prog, \mathcal{P})$ *is satisfiable if and only if* $InferMonome(Prog, \mathcal{P})$ *is* false.

Proof sketch. We provide a proof sketch in this paper.

"$\Longrightarrow$": Let us assume that $SymbInferMonome(Prog, \mathcal{P})$ is satisfiable. Given a satisfying model $\mathcal{M}$ to $SymbInferMonome(Prog, \mathcal{P})$, one can split $\mathcal{M}$ into a set of models $\{\mathcal{M}_i\}_i$ where $\mathcal{M}_i$ assigns values to the i-th copy of the variables in ϕ^i, only agreeing on the values of b_p^i and *guess*. Also, the valuation of the b_p^i can be used to construct the sets $\mathcal{Q}_i$; $\mathcal{Q}_{i+1} \leftarrow \mathcal{Q}_i \cup \{p\}$ when b_p^i is true in $\mathcal{M}$.

"$\Longleftarrow$": Let us assume that $InferMonome(Prog, \mathcal{P})$ returns false. Then by Lemma 2, we can construct a model $\mathcal{M}$ by the union of the models $\{\mathcal{M}_i\}_i$, and construct an evaluation for b_p^i as follows: If $\mathcal{Q}_{i+1} \setminus \mathcal{Q}_i = \{p\}$, then assign b_p^i to true. In all other cases, assign b_p^i to be false. The model $\mathcal{M}$ satisfies $SymbInferMonome(Prog, \mathcal{P})$. □

Theorem 4. *For an assertion logic closed under wp and Boolean connectives, the complexity of* $InferMonome(Prog, \mathcal{P})$ *matches the complexity of* $Check(Prog, I)$.

Proof sketch. Since $SymbInferMonome(Prog, \mathcal{P})$ results in a formula which is polynomial in $\mathcal{P}$ and the size of $wp(body, p)$ for any predicate $p \in \mathcal{P}$, the complexity of checking the satisfiability of $SymbInferMonome(Prog, \mathcal{P})$ is simply the complexity of checking assertions in the assertion logic in which $Check(Prog, I)$ is expressed. □

Corollary 1. *If the decision problem for* $Check(Prog, I)$ *is* **Co-NP** *complete, then the decision problem for* $InferMonome(Prog, \mathcal{P})$ *is* **Co-NP** *complete.*

The encoding $SymbInferMonome(Prog, \mathcal{P})$ can also be seen as an alternative algorithm for the $InferMonome(Prog, \mathcal{P})$ problem. However, when the formula $SymbInferMonome(Prog, \mathcal{P})$ is unsatisfiable, it does not readily provide us with the invariant I. We believe this can be extracted from the unsatisfiable core, and we are currently working on it.

5 Clausal Abstraction

For any set $\mathcal{R} \subseteq \mathcal{P}$, a *clause* over $\mathcal{R}$ is the formula $\bigvee_{p \in \mathcal{R}} p$. For a program $Prog(pre, post, body)$ and a set of predicates $\mathcal{P}$, let us define $InferClause(Prog, \mathcal{P})$ to return true if and only if there exists a $\mathcal{R} \subseteq \mathcal{P}$ such that $Check(Prog, \bigvee_{p \in \mathcal{R}} p)$ is true.

5.1 Dual `Houdini` Algorithm

First, let us describe an algorithm for solving the $InferClause(Prog, \mathcal{P})$ problem. Recall that the `Houdini` algorithm for solving the $InferMonome(Prog, \mathcal{P})$ problem starts with the conjunction of all predicates in $\mathcal{P}$ and iteratively removes predicates until a fixpoint is reached. Conversely, the dual `Houdini` algorithm starts with the disjunction of all predicates in $\mathcal{P}$ and removes predicates until a

fixpoint is reached. The algorithm invokes $\mathit{FindInv}(\mathcal{P})$ (in Figure 3), only this time using the following definitions of *FailsCheck* and *RemovesPredicate* macros:

$$\begin{aligned}
&\mathit{FailsCheck}(\mathcal{S},\mathcal{M}) && \triangleq \mathcal{M} \models (\textstyle\bigwedge_{q\in\mathcal{S}} \neg q) \wedge \mathit{pre}\\
&\mathit{RemovesPredicate}(\mathcal{S},\mathcal{M},p) && \triangleq \vee\mathcal{M} \models \neg \mathit{post} \wedge p\\
&&& \quad \vee\mathcal{M} \models \neg \mathit{wp}(\mathit{body}, \textstyle\bigvee_{q\in\mathcal{S}} q) \wedge p
\end{aligned}$$

In the remainder of this section, we let $\mathit{FindInv}(\mathcal{P})$ denote the algorithm with the above definitions of $\mathit{FailsCheck}(\mathcal{S},\mathcal{M})$ and $\mathit{RemovesPredicate}(\mathcal{S},\mathcal{M},p)$, rather than those given in Figure 3.

Theorem 5. *The procedure* $\mathit{FindInv}(\mathcal{P})$ *enjoys the following properties:*

1. $\mathit{FindInv}(\mathcal{P})$ *returns FAIL if and only if* $\mathit{InferClause}(\mathit{Prog},\mathcal{P})$ *is* false.
2. *If* $\mathit{FindInv}(\mathcal{P})$ *returns* $\mathit{SUCCESS}(\mathcal{R})$*, then for any* $\mathcal{S} \subseteq \mathcal{P}$ *such that* $\mathit{Check}(\mathit{Prog}, \bigvee_{p\in\mathcal{S}} p)$*, we have* $S \subseteq \mathcal{R}$.

The theorem signifies that the dual `Houdini` constructs the *weakest* clause I that satisfies $\mathit{Check}(\mathit{Prog}, I)$, as opposed to `Houdini`, which computes the *strongest* monome I that satisfies $\mathit{Check}(\mathit{Prog}, I)$. This is not surprising because `Houdini` solves the problem in the *forward* direction starting from *pre*, whereas the dual algorithm solves the problem *backwards* staring from *post*.

The structure of the rest of the section is similar to Section 4. For brevity, we mostly state the analogues of lemmas, theorems and symbolic encoding in the next two subsections, without details of the proofs.

5.2 Model-Theoretic Proof

For a $\mathit{guess} \in \mathbb{N}$, an indexed set of models $\{\mathcal{M}_i\}_i$, an indexed set of sets of predicates $\{\mathcal{Q}_i\}_i$, we define a predicate $\mathit{NoClauseInv}(\mathcal{P}, \mathit{guess}, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$, that is true if and only if the following conditions hold:

1. $1 \leq \mathit{guess} \leq |\mathcal{P}| + 1$,
2. $\mathcal{Q}_1 = \{\}$,
3. For $1 \leq i < \mathit{guess}$, $\mathcal{Q}_{i+1} \setminus \mathcal{Q}_i = \{p_i\}$ for some $p_i \in \mathcal{P}$,
4. For each $1 \leq i < \mathit{guess}$, and for $p_i \in \mathcal{Q}_{i+1} \setminus \mathcal{Q}_i$, either $\mathcal{M}_i \models \neg \mathit{post} \wedge p_i$ or $\mathcal{M}_i \models \neg \mathit{wp}(\mathit{body}, \bigvee_{p\in\mathcal{P}\setminus\mathcal{Q}_i} p) \wedge p_i$, and
5. $\mathcal{M}_{\mathit{guess}} \models (\bigwedge_{p\in\mathcal{P}\setminus\mathcal{Q}_{\mathit{guess}}} \neg p) \wedge \mathit{pre}$.

The following three lemmas establish the connection between the *InferClause* problem, the *NoClauseInv* predicate, and the *FindInv* algorithm.

Lemma 5. *If* $\mathit{FindInv}(\mathcal{P})$ *returns FAIL, then* $\mathit{NoMonomeInv}(\mathcal{P}, \mathit{guess}, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$*holds on the values computed by the procedure.*

Lemma 6. *If* $\mathit{NoClauseInv}(\mathcal{P}, \mathit{guess}, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$ *holds for some* $\mathit{guess} \in \mathbb{N}$*, a set of models* $\mathcal{M}_1, \ldots, \mathcal{M}_{\mathit{guess}}$*, and sets of predicates* $\mathcal{Q}_1, \ldots, \mathcal{Q}_{\mathit{guess}}$*, then* $\mathit{InferClause}(\mathit{Prog},\mathcal{P})$ *is* false.

Lemma 7. *If* $\mathit{FindInv}(\mathcal{P})$ *returns* $\mathit{SUCCESS}(\mathcal{R})$*, then* $\mathit{Check}(\mathit{Prog}, \bigvee_{p\in\mathcal{R}} p)$ *is* true*, and therefore* $\mathit{InferClause}(\mathit{Prog},\mathcal{P})$ *is* true.

5.3 Symbolic Encoding

Similar to the monomial abstraction, we define the $\mathit{SymbInferClause}(\mathit{Prog}, \mathcal{P})$ which is satisfiable if and only if $\mathit{InferClause}(\mathit{Prog}, \mathcal{P})$ returns false. The symbolic encoding for clausal abstraction retains the structure of the symbolic encoding for monomial abstraction. The only difference is that the definitions of the predicates iter_i and check_i change as follows:

For each $i \in [1, n]$, we define

$$\begin{array}{rl} \mathit{iter}_i \triangleq & \vee \neg \mathit{post}^i \wedge \bigwedge_{p \in \mathcal{P}} (b_p^i \implies p^i) \\ & \vee \neg \mathit{wp}(s, \bigvee_{p \in \mathcal{P}} \mathit{present}_p^i \wedge p)^i \wedge \bigwedge_{p \in \mathcal{P}} (b_p^i \implies p^i) \end{array}$$

For each $i \in [1, n+1]$, we define

$$\mathit{check}_i \triangleq \bigwedge_{p \in \mathcal{P}} (\mathit{present}_p^i \implies \neg p^i) \wedge \mathit{pre}^i$$

Finally, the analogues of Theorem 3, Theorem 4, and Corollary 1 can be shown for the clausal abstraction as well.

Theorem 6. *The formula $\mathit{SymbInferClause}(\mathit{Prog}, \mathcal{P})$ is satisfiable if and only if $\mathit{InferClause}(\mathit{Prog}, \mathcal{P})$ is* false.

Theorem 7. *For an assertion logic closed under wp and Boolean connectives, the complexity of $\mathit{InferClause}(\mathit{Prog}, \mathcal{P})$matches the complexity of $\mathit{Check}(\mathit{Prog}, I)$.*

Corollary 2. *If the decision problem for $\mathit{Check}(\mathit{Prog}, I)$ is* **Co-NP** *complete, then the decision problem for $\mathit{InferClause}(\mathit{Prog}, \mathcal{P})$ is* **Co-NP** *complete.*

6 Conclusions

Formulation of predicate abstraction as a decision problem allows us to infer annotations for programs in a property guided manner, by leveraging off-the-shelf and efficient verification condition generators. In this work, we have studied the complexity of the decision problem of predicate abstraction relative to the complexity of checking an annotated program. The monomial and clausal restrictions considered in this paper are motivated by practical applications, where most invariants are monomes and a handful of clauses [16]. We have also provided a new algorithm for solving the $\mathit{InferClause}(\mathit{Prog}, I)$ problem.

There are several questions that are still unanswered. We would like to construct an invariant from the unsatisfiable core, when the symbolic encoding of the $\mathit{InferMonome}(\mathit{Prog}, \mathcal{P})$ or $\mathit{InferClause}(\mathit{Prog}, \mathcal{P})$ returns unsatisfiable. It is also not clear what the complexity is for inferring invariants that are either a disjunction of up to c monomes, or a conjunction of up to c clauses, for a fixed constant c.

References

1. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Programming Language Design and Implementation (PLDI 2001), pp. 203–213 (2001)
2. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Program Analysis For Software Tools and Engineering (PASTE 2005), pp. 82–87 (2005)
3. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
4. Condit, J., Hackett, B., Lahiri, S.K., Qadeer, S.: Unifying type checking and property checking for low-level code. In: Principles of Programming Languages (POPL 2009), pp. 302–314 (2009)
5. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages (POPL 1977), pp. 238–252 (1977)
6. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Principles of Programming Languages (POPL 1978), pp. 84–96 (1978)
7. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001)
8. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
9. Gulwani, S., Srivastava, S., Venkatesan, R.: Constraint-based invariant inference over predicate abstraction. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 120–135. Springer, Heidelberg (2009)
10. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Principles of Programming Languages (POPL), pp. 232–244 (2004)
11. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Principles of Programming Languages (POPL 2002), pp. 58–70 (2002)
12. Jain, H., Kroening, D., Sharygina, N., Clarke, E.M.: Word level predicate abstraction and refinement for verifying rtl verilog. In: Design Automation Conference (DAC 2005), pp. 445–450. ACM, New York (2005)
13. Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press, Princeton (1995)
14. Lahiri, S.K., Bryant, R.E.: Constructing quantified invariants via predicate abstraction. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 267–281. Springer, Heidelberg (2004)
15. Lahiri, S.K., Qadeer, S.: Back to the future: Revisiting precise program verification using SMT solvers. In: Principles of Programming Languages (POPL 2008), pp. 171–182 (2008)
16. Lahiri, S.K., Qadeer, S., Galeotti, J.P., Voung, J.W., Wies, T.: Intra-module inference. In: Computer-Aided Verification (CAV 2009) (July 2009)
17. Sagiv, S., Reps, T.W., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. ACM Transactions on Programming Languages and Systems (TOPLAS) 20(1), 1–50 (1998)
18. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constraint-based linear-relations analysis. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 53–68. Springer, Heidelberg (2004)

19. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 25–41. Springer, Heidelberg (2005)
20. Satisfiability Modulo Theories Library (SMT-LIB), `http://goedel.cs.uiowa.edu/smtlib/`
21. Solar-Lezama, A., Rabbah, R.M., Bodík, R., Ebcioglu, K.: Programming by sketching for bit-streaming programs. In: Programming Language Design and Implementation (PLDI 2005), pp. 281–294. ACM, New York (2005)

Appendix

Proof of Lemma 2

Proof. The first four conditions of $NoMonomeInv(\mathcal{P}, guess, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$ are easily satisfied by construction. The fifth condition can be shown by observing that $\mathcal{Q}_{guess} \cup \mathcal{R} = \mathcal{P}$ is a precondition to *FindInvAux*. □

Proof of Lemma 3

Proof. Let us assume that there exists $guess$, $\{\mathcal{M}_i\}_{i\in[1,guess]}$, $\{\mathcal{Q}_i\}_{i\in[1,guess]}$ satisfying $NoMonomeInv(\mathcal{P}, guess, \{\mathcal{M}_i\}_i, \{\mathcal{Q}_i\}_i)$. We will show that in such a case, $InferMonome(Prog, \mathcal{P})$ returns false.

We will prove this by contradiction. Let us assume that there is a $\mathcal{R} \subseteq \mathcal{P}$ such that $Check(Prog, \bigwedge_{p\in\mathcal{R}} p)$ holds. Let $I = \bigwedge_{p\in\mathcal{R}} p$. We claim that $\mathcal{R} \cap \mathcal{Q}_{guess} = \{\}$. We will prove this by induction on i for $1 \leq i \leq guess$, showing $\mathcal{R} \cap \mathcal{Q}_i = \{\}$. The base case for $i = 1$ holds vacuously since $\mathcal{Q}_1 = \{\}$. Let us assume that the induction hypothesis holds for all $j \leq i$. Consider the set $\{p_i\} = \mathcal{Q}_{i+1} \setminus \mathcal{Q}_i$. We show that p_i cannot be in $\mathcal{R}$. Consider the two cases how p_i gets removed.

- If $\mathcal{M}_i \models pre \wedge \neg p_i$, then we know that $\not\models pre \implies p_i$. If $p_i \in \mathcal{R}$, then $\models pre \implies p_i$, which is a contradiction.
- On the other hand, suppose $\mathcal{M}_i \models (\bigwedge_{p\in\mathcal{P}\setminus\mathcal{Q}_i} p) \wedge \neg wp(body, p_i)$. By induction hypothesis, we know that $\mathcal{R} \cap \mathcal{Q}_i = \{\}$, therefore $\mathcal{R} \subseteq \mathcal{P} \setminus \mathcal{Q}_i$. This implies $\mathcal{M}_i \models (\bigwedge_{p\in\mathcal{R}} p) \wedge \neg wp(body, p_i)$. If $p_i \in \mathcal{R}$, then we can conclude that $\mathcal{M}_i \models (\bigwedge_{p\in\mathcal{R}} p) \wedge \neg(\bigwedge_{p\in\mathcal{R}} wp(body, p))$. Since wp distributes over $\wedge$, we have $\bigwedge_{p\in\mathcal{R}} wp(body, p)) = wp(body, \bigwedge_{p\in\mathcal{R}} p)$ and thus $\mathcal{M}_i \models I \wedge \neg wp(body, I)$. Since I satisfies $Check(Prog, I)$, we have arrived at a contradiction.

Having shown that $\mathcal{R} \cap \mathcal{Q}_{guess} = \{\}$, we know that $\mathcal{R} \subseteq \mathcal{P} \setminus \mathcal{Q}_{guess}$. Since $\mathcal{M}_{guess} \models (\bigwedge_{p\in\mathcal{P}\setminus\mathcal{Q}_{guess}} p) \wedge \neg post$, it implies that $\mathcal{M}_{guess} \models (\bigwedge_{p\in\mathcal{R}} p) \wedge \neg post$, which in turn implies $\not\models I \implies post$, which contradicts our assumption that $Check(Prog, I)$. □

Proof of Lemma 4

Proof. Let *FindInvAux* return $SUCCESS(\mathcal{R})$ for an argument $\mathcal{R}$ to *FindInvAux*. Since both the if branches are not taken, all the following conditions hold:

1. $\models (\bigwedge_{q \in \mathcal{R}} q) \implies post$.
2. For each $q \in \mathcal{R}$, $\models pre \implies q$, and therefore $\models pre \implies (\bigwedge_{q \in \mathcal{R}} q)$.
3. For each $q \in \mathcal{R}$, $\models (\bigwedge_{p \in \mathcal{R}} p) \implies wp(body, q)$. Since wp distributes over $\wedge$, this implies $\models (\bigwedge_{p \in \mathcal{R}} p) \implies wp(body, (\bigwedge_{p \in \mathcal{R}} p))$.

These conditions mean that $Check(Prog, \bigwedge_{q \in \mathcal{R}} q)$ holds. □

Proof of Lemma 1

Proof. Part (1) is proved easily by combining Lemmas 2, 3, and 4.

To prove part (2), we establish the following precondition for *FindInvAux* procedure: For any set of predicates $\mathcal{S} \subseteq \mathcal{P}$ such that $Check(Prog, \bigwedge_{p \in \mathcal{S}} p)$ holds, $\mathcal{S} \cap \mathcal{Q}_{guess} = \{\}$. The proof follows by induction on *guess* similar to the proof of Lemma 3. Similarly, $\mathcal{R} \cup \mathcal{Q}_{guess} = \mathcal{P}$ is another precondition for *FindInvAux*. Therefore, whenever *FindInvAux* returns $SUCCESS(\mathcal{R})$, $(\bigwedge_{p \in \mathcal{R}} p)$ is the strongest monomial invariant over $\mathcal{P}$ that satisfies the program. □

Efficient Intuitionistic Theorem Proving with the Polarized Inverse Method

Sean McLaughlin and Frank Pfenning

Department of Computer Science
Carnegie Mellon University

Abstract. The inverse method is a generic proof search procedure applicable to non-classical logics satisfying cut elimination and the subformula property. In this paper we describe a general architecture and several high-level optimizations that enable its efficient implementation. Some of these rely on logic-specific properties, such as polarization and focusing, which have been shown to hold in a wide range of non-classical logics. Others, such as rule subsumption and recursive backward subsumption apply in general. We empirically evaluate our techniques on first-order intuitionistic logic with our implementation *Imogen* and demonstrate a substantial improvement over all other existing intuitionistic theorem provers on problems from the ILTP problem library.

1 Introduction

The *inverse method* [11,6] uses forward saturation, generalizing resolution to non-classical logics satisfying the subformula property and cut elimination. *Focusing* [1,10] reduces the search space in a sequent calculus by restricting the application of inference rules based on the *polarities* [9] of the connectives and atomic formulas. In this paper we describe a framework for reasoning in such logics, and exhibit a concrete implementation of a theorem prover for intuitionistic predicate logic. The implementation, called *Imogen*,[1] is by some measure the most effective first order intuitionistic theorem prover: On the ILTP library of intuitionistic challenge problems [16], a collection of intuitionistic problems similar to the well known TPTP [17] library, Imogen solves over 150 more problems than its closest competitor.

This work continues a line of research on building efficient theorem provers for non-classical logics using the inverse method, following Tammet [18] (for intuitionistic and linear logic), Linprover [4,3] (for intuitionistic linear logic), and the early propositional version of Imogen [12].

There are two primary contributions of this paper. On the logical side, *explicit polarization* of a given input formula determines basic characteristics of the search space. The ability to choose from different polarizations of a formula, refining ideas from Chaudhuri et al. [5], allows for logically motivated optimizations that do not compromise soundness or completeness. On the implementation side, our architecture provides a clean interface between the specification of basic logical inference (the *front end*) on one side and saturating proof search (the *back end*) on the other. This separation allows

[1] Imogen is available at `http://www.cs.cmu.edu/~seanmcl/research/imogen/`

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 230–244, 2009.

for both theory-specific optimizations in the front end and logic-independent optimizations in the back end. As examples of the later, we present two simple but novel redundancy elimination techniques: *inference rule subsumption* and *recursive backward subsumption*.

2 A Polarized Sequent Calculus

We can limit the search space of the inverse method by searching only for *focused proofs* [1]. In this section we give the rules for the (ground) backward polarized sequent calculus. This ground calculus will then be lifted to a free variable calculus and proof search will proceed in the forward direction, from the initial sequents to the goal, following the inverse method recipe [6]. One novelty of our approach is the use of explicit polarization in formulas that syntactically mark the polarity of the atoms and connectives. We first describe polarized formulas, and then show the backward sequent calculus.

2.1 Polarized Formulas

A connective is *positive* if its left rule in the sequent calculus is invertible and *negative* if its right rule is invertible. As shown below, our proof search fundamentally depends on the polarity of connectives. In intuitionistic logic, the status of conjunction and truth is ambiguous in the sense that they are both positive and negative, while their status in linear logic is uniquely determined. We therefore syntactically distinguish positive and negative formulas with so-called *shift* operators [9] explicitly coercing between them. Even though we use the notation of linear logic, the behavior of the connectives is not linear.

In the following, the meta-variable P ranges over atomic formulas which have the form $p(t_1, \ldots, t_n)$ for predicates p. Note also that both in formulas and sequents, the signs are not actual syntax but mnemonic guides to the reader.

$$\text{Positive formulas } A^+ ::= P^+ \mid A^+ \otimes A^+ \mid 1 \mid A^+ \oplus A^+ \mid 0 \mid \exists x.\, A^+ \mid \downarrow A^-$$
$$\text{Negative formulas } A^- ::= P^- \mid A^- \mathbin{\&} A^- \mid \top \mid A^+ \multimap A^- \mid \forall x.\, A^- \mid \uparrow A^+$$

The translation A^- of an (unpolarized) formula F in intuitionistic logic is nondeterministic, subject only to the constraints that the *erasure* defined below coincides with the original formula ($|A^-| = F$) and all predicates are assigned a consistent polarity.

For example, the formula $((p \vee r) \wedge (q \supset r)) \supset (p \supset q) \supset r$ can be interpreted as any of the following polarized formulas (among others):

$$((\downarrow p^- \oplus \downarrow r^-) \otimes \downarrow(\downarrow q^- \multimap r^-)) \multimap (\downarrow(\downarrow p^- \multimap q^-) \multimap r^-)$$
$$\downarrow\uparrow((\downarrow p^- \oplus \downarrow r^-) \otimes \downarrow(\downarrow q^- \multimap r^-)) \multimap (\downarrow\uparrow\downarrow(\downarrow p^- \multimap q^-) \multimap r^-)$$
$$\downarrow(\uparrow(p^+ \oplus r^+) \mathbin{\&} (q^+ \multimap \uparrow r^+)) \multimap (\downarrow(p^+ \multimap \uparrow q^+) \multimap \uparrow r^+)$$

Shift operators have highest binding precedence in our presentation of the examples. As we will see from the inference rules given below, the choice of translation determines

$$
\begin{array}{lll}
|A^+ \oplus B^+| = |A^+| \vee |B^+| & |0| = \bot & |1| = \top \\
|A^+ \otimes B^+| = |A^+| \wedge |B^+| & |{\downarrow}A^-| = |A^-| & |P^+| = P \\
|A^- \& B^-| = |A^-| \wedge |B^-| & |\top| = \top & |P^-| = P \\
|A^+ \multimap B^-| = |A^+| \supset |B^-| & |{\uparrow}A^+| = |A^+| & \\
|\forall x.\, A^-| = \forall x.\, |A^-| & |\exists x.\, A^+| = \exists x.\, |A^+| &
\end{array}
$$

Fig. 1. Erasure of polarized formulas

the search behavior on the resulting polarized formula. Different choices can lead to search spaces with radically different structure [5,12].

2.2 Backward Polarized Sequent Calculus

The backward calculus is a refinement of Gentzen's LJ that eliminates don't-care nondeterministic choices, and manages don't-know nondeterminism by chaining such inferences in sequence. Andreoli was the first to define this *focusing* strategy and prove it complete [1] for linear logic. Similar proofs for other logics soon followed [8,10,19], demonstrating that polarization and focusing can be applied to optimize search in a wide variety of logics.

The polarized calculus is defined via four mutually recursive judgments. In the judgments, we separate the antecedents into positive and negative zones. We write Γ for an unordered collection of negative formulas or positive atoms. Dually, C stands for a positive formula or a negative atom.

The first two judgments concern formulas with invertible rules on the right and left. Together, the two judgments form the *inversion phase* of focusing. In the rules RA-$\forall$ and LA-$\exists$, a is a new parameter.[2]

The context Δ^+ is consists entirely of positive formulas and is ordered so that inference rules can only be applied to the rightmost formula, eliminating don't-care nondeterminism.

$\boxed{\Gamma; \Delta^+ \Longrightarrow A^-; \cdot}$ (Right Inversion)

$$\frac{\Gamma; \Delta^+ \Longrightarrow \cdot; P^-}{\Gamma; \Delta^+ \Longrightarrow P^-; \cdot}\ \text{RA-Atom} \qquad \frac{\Gamma; \Delta^+ \Longrightarrow A_1^-; \cdot \quad \Gamma; \Delta^+ \Longrightarrow A_2^-; \cdot}{\Gamma; \Delta^+ \Longrightarrow A_1^- \& A_2^-; \cdot}\ \text{RA-\&}$$

$$\frac{\Gamma; \Delta^+, A_1^+ \Longrightarrow A_2^-; \cdot}{\Gamma; \Delta^+ \Longrightarrow A_1^+ \multimap A_2^-; \cdot}\ \text{RA-}\multimap \qquad \frac{}{\Gamma; \Delta^+ \Longrightarrow \top; \cdot}\ \text{RA-}\top$$

$$\frac{\Gamma; \Delta^+ \Longrightarrow A(a)^-; \cdot}{\Gamma; \Delta^+ \Longrightarrow \forall x.\, A(x)^-; \cdot}\ \text{RA-}\forall^a \qquad \frac{\Gamma; \Delta^+ \Longrightarrow \cdot; A^+}{\Gamma; \Delta^+ \Longrightarrow {\uparrow}A^+; \cdot}\ \text{RA-}\uparrow$$

$\boxed{\Gamma; \Delta^+ \Longrightarrow \cdot; C}$ (Left Inversion)

[2] In our calculus, parameters differ syntactically from term variables, and are thus slightly different than the *eigenvariables* found in other presentations of the inverse method. A formalization of parameters and their effect on unification can be found in Chaudhuri [3].

$$\frac{\Gamma, P^+; \Delta^+ \Longrightarrow \cdot; C}{\Gamma; \Delta^+, P^+ \Longrightarrow \cdot; C}\ \text{LA-Atom} \qquad \frac{\Gamma; \Delta^+, A_1^+, A_2^+ \Longrightarrow \cdot; C}{\Gamma; \Delta^+, A_1^+ \otimes A_2^+ \Longrightarrow \cdot; C}\ \text{LA-}\otimes$$

$$\frac{\Gamma; \Delta^+, A(a)^+ \Longrightarrow \cdot; C}{\Gamma; \Delta^+, \exists x.\, A(x)^+ \Longrightarrow \cdot; C}\ \text{LA-}\exists^a \qquad \frac{\Gamma; \Delta^+, A_1^+ \Longrightarrow \cdot; C \quad \Gamma; \Delta^+, A_2^+ \Longrightarrow \cdot; C}{\Gamma; \Delta^+, A_1^+ \oplus A_2^+ \Longrightarrow \cdot; C}\ \text{LA-}\oplus$$

$$\frac{\Gamma; \Delta^+ \Longrightarrow \cdot; C}{\Gamma; \Delta^+, 1 \Longrightarrow \cdot; C}\ \text{LA-1} \qquad \frac{}{\Gamma; \Delta^+, 0 \Longrightarrow \cdot; C}\ \text{LA-0} \qquad \frac{\Gamma, A^-; \Delta^+ \Longrightarrow \cdot; C}{\Gamma; \Delta^+, \downarrow A^- \Longrightarrow \cdot; C}\ \text{LA-}\downarrow$$

The next two judgments are concerned with non-invertible rules. These two judgments make up the *focusing phase*.

$\boxed{\Gamma \gg A^+}$ (Right Focusing)

$$\frac{}{\Gamma, P^+ \gg P^+}\ \text{RS-Atom} \qquad \frac{\Gamma \gg A_1^+ \quad \Gamma \gg A_2^+}{\Gamma \gg A_1^+ \otimes A_2^+}\ \text{RS-}\otimes \qquad \frac{}{\Gamma \gg 1}\ \text{RS-1}$$

$$\frac{\Gamma \gg A_1^+}{\Gamma \gg A_1^+ \oplus A_2^+}\ \text{RS-}\oplus_1 \qquad \frac{\Gamma \gg A_2^+}{\Gamma \gg A_1^+ \oplus A_2^+}\ \text{RS-}\oplus_2 \qquad \text{No rule for } 0$$

$$\frac{\Gamma \gg A(t)^+}{\Gamma \gg \exists x.\, A^+}\ \text{RS-}\exists \qquad \frac{\Gamma; \cdot \Longrightarrow A^-; \cdot}{\Gamma \gg \downarrow A^-}\ \text{RS-}\downarrow$$

$\boxed{\Gamma; A^- \ll C}$ (Left Focusing)

$$\frac{}{\Gamma; P^- \ll P^-}\ \text{LS-Atom} \qquad \frac{\Gamma; A_1^- \ll C}{\Gamma; A_1^- \& A_2^- \ll C}\ \text{LS-}\&_1 \qquad \frac{\Gamma; A_2^- \ll C}{\Gamma; A_1^- \& A_2^- \ll C}\ \text{LS-}\&_2 \qquad \text{No rule for } \top$$

$$\frac{\Gamma \gg A_1^+ \quad \Gamma; A_2^- \ll C}{\Gamma; A_1^+ \multimap A_2^- \ll C}\ \text{LS-}\multimap \qquad \frac{\Gamma; A(t)^- \ll C}{\Gamma; \forall x.\, A^- \ll C}\ \text{LS-}\forall \qquad \frac{\Gamma; A^+ \Longrightarrow \cdot; C}{\Gamma; \uparrow A^+ \ll C}\ \text{LS-}\uparrow$$

Backward search for a proof of A^- starts with an inversion from $\cdot; \cdot \Longrightarrow A^-; \cdot$. The proof then alternates between focusing and inversion phases. Call a focusing phase followed by an inversion phase a *block*. The boundary between blocks is of particular importance. The sequents at the boundary have the form $\Gamma; \cdot \Longrightarrow \cdot; C$. We call such sequents *stable*. There are two rules that control the phase changes at stable sequents (the block boundaries).

$$\frac{\Gamma \gg A^+}{\Gamma; \cdot \Longrightarrow \cdot; A^+}\ \text{FocusR} \qquad \frac{\Gamma, A^-; A^- \ll C}{\Gamma, A^-; \cdot \Longrightarrow \cdot; C}\ \text{FocusL}$$

An example of a backward derivation highlighting the block structure, is shown in Figure 2. a, b, and c are negative atoms. The elided sections are deterministic application of the above rules. Note that the nondeterminism occurs only at block boundaries.

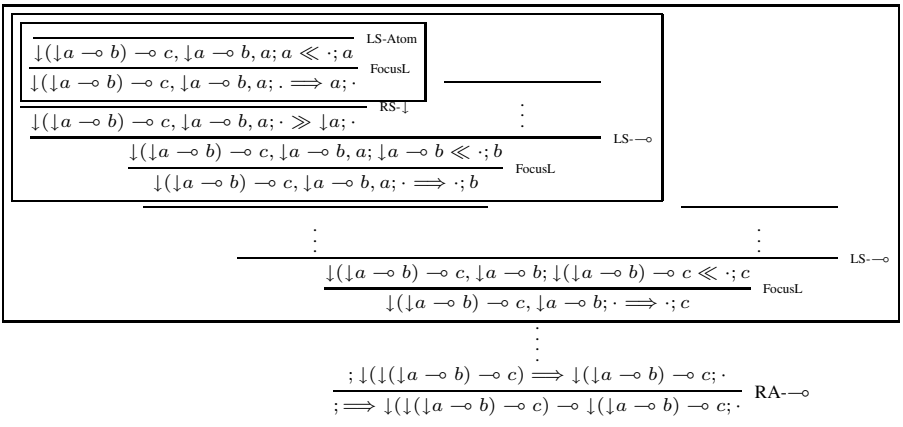


Fig. 2. Backward proof, with blocks

Theorem 1. *[10] If there exists an intuitionistic derivation of A, then for any polarization A^- of A, there exists a focused derivation of $\cdot;\cdot \Longrightarrow A^-;\cdot$.*

2.3 Synthetic Connectives and Derived Rules

We have already observed that backward proofs have the property that the proof is broken into blocks, with stable sequents at the boundary. The only rules applicable to stable sequents are the rules that select a formula on which to focus. It is the formulas occurring in stable sequents that form the primary objects of our further inquiry.

It helps to think of such formulas, abstracted over their free variables, as *synthetic connectives* [2]. Define the synthetic connectives of a formula A as all subformulas of A that could appear in stable sequents in a focused backward proof. In a change of perspective, we can consider each block of a proof as the application of a left or right rule for a synthetic connective. The rules operating on synthetic connectives are derived from the rules for its constituent formulas. We can thus consider a backward proof as a proof using only these synthetic (derived) rules. Each derived rule then corresponds to a block of the original proof.

Since we need only consider stable sequents and synthetic connectives, we can simplify notation, and ignore the (empty) positive left and negative right zones in the derived rules. Write $\Gamma;\cdot \Longrightarrow \cdot;C$ as $\Gamma \Longrightarrow C$. As a further simplification, we can give formulas a predicate label and abstract over its free variables. This labeling technique is described in detail in Degtyarev and Voronkov [6]. For the remainder, we assume this labeling has been carried out. Define an *atomic formula* as either a label or a predicate applied to a (possibly empty) list of terms. After labeling, our sequents consist entirely of atomic formulas.

Example 1. *In Figure 2, frame boxes surround the three blocks of the proof. The synthetic connectives are a, $\downarrow a \multimap b$ and $\downarrow(\downarrow a \multimap b) \multimap c$. There is a single derived rule for each synthetic connective (though this is not the case in general). We implicitly carry the principal formula of a left rule to all of its premises.*

$$\frac{}{\Gamma, a \Longrightarrow a}\ {\scriptstyle Syn_1} \qquad \frac{\Gamma \Longrightarrow a}{\Gamma, \downarrow a \multimap b \Longrightarrow b}\ {\scriptstyle Syn_2} \qquad \frac{\Gamma, a \Longrightarrow b}{\Gamma, \downarrow(\downarrow a \multimap b) \multimap c \Longrightarrow c}\ {\scriptstyle Syn_3}$$

These rules correspond to the blocks shown in Figure 2. Corresponding labeled rules for $L_1 = \downarrow a \multimap b$ *and* $L_2 = \downarrow(\downarrow a \multimap b) \multimap c$ *are*

$$\frac{}{\Gamma, a \Longrightarrow a}\ {\scriptstyle Syn_1} \qquad \frac{\Gamma \Longrightarrow a}{\Gamma, L_1 \Longrightarrow b}\ {\scriptstyle Syn_2} \qquad \frac{\Gamma, a \Longrightarrow b}{\Gamma, L_2 \Longrightarrow c}\ {\scriptstyle Syn_3}$$

Then the blocks of the proof from Figure 2 can be compressed to the succinct

$$\cfrac{\cfrac{\cfrac{}{L_1, L_2, a \Longrightarrow a}\ {\scriptstyle Syn_1}}{L_1, L_2, a \Longrightarrow b}\ {\scriptstyle Syn_2}}{L_1, L_2 \Longrightarrow c}\ {\scriptstyle Syn_3}$$

3 The Polarized Inverse Method

In the previous section we developed a system for focused backwards proofs. We first described backward focused proofs because the inference rules are simpler and the relation to the semantics, e.g., natural deduction, is more direct. We will now invert the backward calculus, allowing us to understand synthetic inference rules in the forward direction. The inverse method has a number of advantages over backward methods. The most important is that derived sequents in the inverse method are independent entities. That is, their free variables are quantified locally outside the sequent. In contrast, backward sequents are only valid in a global context: Variables are quantified over the entire proof object. This difference makes determining non-theoremhood (via saturation) and redundancy elimination (via subsumption) easier for the inverse method.

3.1 Forward Rules

Recall the following rules from the backward (unfocused) intuitionistic sequent calculus:

$$\frac{}{\Gamma, a \Longrightarrow a}\ \text{Init} \qquad \frac{}{\Gamma, \bot \Longrightarrow A}\ \bot\text{-L} \qquad \frac{\Gamma \Longrightarrow A \quad \Gamma \Longrightarrow B}{\Gamma \Longrightarrow A \wedge B}\ \wedge\text{-R}$$

Interpreting these rules for forward search shows some difficulties. In the forward direction we want to guess neither the context Γ, nor the formula A in the $\bot$-L rule. We therefore allow the succedent to be empty, and ignore Γ in such initial sequents. The analogous forward sequents have the form

$$\frac{}{a \longrightarrow a}\ \text{Init} \qquad \frac{}{\bot \longrightarrow \cdot}\ \bot\text{-L} \qquad \frac{\Gamma_1 \longrightarrow A \quad \Gamma_2 \longrightarrow B}{\Gamma_1 \cup \Gamma_2 \longrightarrow A \wedge B}\ \wedge\text{-R}$$

A forward sequent stands for all of its weakening and substitution instances. This new form of sequent requires a more complicated notion of *matching* (or *applying*) inference rules. We now define the operations necessary for proof search in the forward polarized sequent calculus *lifted* [6] to free variables. We assume the reader is familiar with the usual notions of substitutions and most general unifiers.

Definition 1 (Forward Sequents). *A* forward sequent *has the form* $\Gamma \longrightarrow C$ *where* Γ *is a set of atomic formulas and* C *is either the empty set or a set containing a single atomic formula. It is written* $\Gamma \longrightarrow A$ *in the case that* $C = \{A\}$*, or* $\Gamma \longrightarrow \cdot$ *in case* $C = \emptyset$*. The set* Γ *consists of the* antecedents*, and* C *is called the* succedent*.*

The variables of a sequent are implicitly universally quantified outside the sequent. This means every time a sequent is used, we have to rename its variables to be fresh. We will apply such renamings tacitly.

Definition 2 (Inference rules). *An* inference rule *with name* id*,*

$$\frac{H_1 \quad \ldots \quad H_n}{Q}\ {id}^{\Pi}$$

has premises $H_1, \ldots, H_n$ *and conclusion* Q*, all of which are sequents.* Π *is a set of parameters (the* fixed parameters*) that are introduced during the inversion phase. Inference rules are schematic in their variables, which can range over formulas or terms.*

Matching. Given some known sequents, we wish to derive new sequents using the inference rules. The process of inferring new sequents from known sequents using inference rules is called *matching*. First note that we must take into account the parameters introduced by the rule to ensure the eigenvariable condition. Consider the rule

$$\frac{\longrightarrow\ p(x, a)}{\longrightarrow\ \exists x.\, \forall y.\, p(x, y)}\ R^a$$

We must ensure when matching rule R to sequent $\Delta \longrightarrow \delta$ that parameter a does not occur in Δ. Moreover, x must not be unified with any term containing a. We will define matching by cases depending on whether the consequent of the conclusion is empty or not. Let $\mathtt{vars}(t)$ denote the free variables of term t.

Definition 3 (Rule Matching 1). *Sequents* $\Delta_1 \longrightarrow \delta_1, \ldots, \Delta_n \longrightarrow \delta_n$ *match rule*

$$\frac{\Gamma_1 \longrightarrow A_1 \quad \cdots \quad \Gamma_n \longrightarrow A_n}{\Gamma \longrightarrow A}\ R^{\Pi}$$

with substitution θ *if the following conditions hold for all* $1 \leq i \leq n$*.*

1. *Either* $\delta_i\theta = A_i\theta$ *or* $\delta_i = \cdot$*.*
2. *The parameters* $\Pi\theta$ *do not occur in* $\Delta_i\theta \setminus \Gamma_i\theta$*.*
3. *The parameters* $\Pi\theta$ *do not occur in* $\mathtt{vars}(\Gamma_i, A_i)\theta$
4. *For any two parameters* $a, b \in \Pi$*,* $a\theta \neq b\theta$

In that case, the resulting sequent is

$$\Gamma\theta \cup (\Delta_1\theta \setminus \Gamma_1\theta) \cup \ldots \cup (\Delta_n\theta \setminus \Gamma_n\theta) \longrightarrow A\theta$$

If there is a premise with an empty succedent in the rule, then the conclusion also has an empty succedent. In this case, we can rearrange the premises so that the first k premises have an empty antecedent. Then we can use the following definition of matching.

Definition 4 (Rule Matching 2). *Sequents* $\Delta_1 \longrightarrow \delta_1, \ldots, \Delta_n \longrightarrow \delta_n$ *match rule*

$$\frac{\Gamma_1 \longrightarrow \cdot \quad \cdots \quad \Gamma_k \longrightarrow \cdot \quad \Gamma_{k+1} \longrightarrow A_{k+1} \quad \cdots \quad \Gamma_n \longrightarrow A_n}{\Gamma \longrightarrow \cdot}\ R^{\Pi}$$

if there exists a substitution θ *such that*

1. *The parameters* $\Pi\theta$ *do not occur in* $\Delta_i\theta \setminus \Gamma_i\theta$.
2. *The parameters* $\Pi\theta$ *do not occur in* $\mathtt{vars}(\Gamma_i, A_i)\theta$
3. *For any two parameters* $a, b \in \Pi$, $a\theta \neq b\theta$

and one of the following conditions holds:

1. – *For all* $1 \leq i \leq k$, $\delta_i = \cdot$.
   – *For all* $k+1 \leq i \leq n$, $\delta_i = \cdot$ *or* $\delta_i\theta = A_i\theta$.
   In this case the resulting sequent is

$$\Gamma\theta \cup (\Delta_1\theta \setminus \Gamma_1\theta) \cup \ldots \cup (\Delta_n\theta \setminus \Gamma_n\theta) \longrightarrow \cdot$$

2. – *There exists* $1 \leq i \leq k$, $\delta_i\theta = A\theta$.
   – *For all* $1 \leq i \leq k$, $\delta_i = \cdot$ *or* $\delta_i\theta = A\theta$.
   – *For all* $k+1 \leq i \leq n$, $\delta_i = \cdot$ *or* $\delta_i\theta = A_i\theta$.
   In this case the resulting sequent is

$$\Gamma\theta \cup (\Delta_1\theta \setminus \Gamma_1\theta) \cup \ldots \cup (\Delta_n\theta \setminus \Gamma_n\theta) \longrightarrow A\theta$$

The definition of matching assures that the forward application simulates a backward rule application. Since we always combine unused premises in the same way, in the rest of the paper we omit the contexts Γ in forward inference rules.

Example 2. *If the synthetic connective is* $L_1 = \downarrow((\exists y.\, \downarrow p(y)) \multimap \forall x.\, (p(x)\ \&\ q(x)))$ *on the right, then the backward and forward synthetic rules are*

$$\frac{\Gamma, p(a) \Longrightarrow p(b) \quad \Gamma, p(a) \Longrightarrow q(b)}{\Gamma \Longrightarrow L_1}\ \mathit{Syn}^{a,b}$$

$$\frac{p(a) \longrightarrow p(b) \quad p(a) \longrightarrow q(b)}{\longrightarrow L_1}\ \mathit{Syn}^{a,b}$$

3.2 Proof Search

Before we can turn our observations into a method for proof search, we need two more crucial definitions. First, the inverse method cannot in general prove a given sequent exactly, but sometimes only a stronger form of it. This is captured by the *subsumption* relation.

Definition 5 (Subsumption). *A sequent* $\Gamma_1 \longrightarrow C_1$ subsumes *a sequent* $\Gamma_2 \longrightarrow C_2$ *if there exists a substitution* θ *such that* $|\Gamma_1\theta| = |\Gamma_1|$ *(i.e.,* θ *does not contract* Γ_1*) and* $\Gamma_1\theta \subseteq \Gamma_2$ *and* $C_1\theta \subseteq C_2$. *Write* $Q \preceq Q'$ *if* Q *subsumes* Q'.

Suppose $Q \preceq Q'$ and we are trying to prove Q'. Since weakening is an admissible rule in the backward calculus, given a backward proof $\mathcal{D}$ of Q, we could modify $\mathcal{D}$ by weakening, yielding a proof of Q'.

The second definition comes from the following observation. It is not the case that $(p(X,Y), p(Y,X) \longrightarrow g) \preceq (p(Z,Z) \longrightarrow g)$, even though $(p(Z,Z) \longrightarrow g)$ is identical to $(p(X,Y), p(Y,X) \longrightarrow g)$ under substitution $\{X \mapsto Z, Y \mapsto Z\}$. (Remember that we maintain the premises as a set.) Since a sequent stands for its substitution instances, we should be able to infer the latter sequent. This consideration motivates the definition of *contraction*:

Definition 6 (Contraction). *$\Gamma\theta, A_1\theta \longrightarrow C\theta$ is a* contraction instance *of a sequent $\Gamma, A_1, A_2 \longrightarrow C$ if θ is the most general unifier of A_1, A_2.*

Now we have the basic operations necessary to define forward search using the polarized inverse method. We begin with a negative polarized input formula A^-. We first decompose the problem into stable sequents by applying the backward rules, inverting the sequent $\cdot;\cdot \Longrightarrow A^-;\cdot$. The leaves of the backward inversion are stable sequents. Each stable sequent is solved independently. (This is why the bottom portion of Figure 2 is not contained in a block.) For each stable sequent, we determine the sequent's synthetic formulas, and generate the corresponding derived rules. We begin with a sequent database containing the *initial sequents*, those synthetic rules with no premises. We repeatedly match the synthetic rules to known sequents in the forward direction. The resulting matches, along with all of their contraction instances, are added to the database. We continue in this way until we either generate a sequent that subsumes the goal, or until the database is saturated, that is, any further inference would only add sequents subsumed by something already in the database. Due to the undecidability of the problem, if the goal is not provable, it is possible that the database will never saturate.

Theorem 2 (Completeness). *If there exists a (ground) backward focused derivation of a polarized formula A, then such a derivation can be constructed using the polarized inverse method.*

Proof. Analogous to the corresponding proof in Chaudhuri [3]. □

4 An Implementation Framework

We turn now to our implementation, called *Imogen*. The implementation is designed as two distinct modules, referred to respectively as the *front end* and the *back end*. The front end deals with the specifics of a particular logic and focusing strategy. It takes a formula as input and returns the initial stable sequents, and for each sequent a complete set of synthetic inference rules and initial sequents. The back end maintains a database of known sequents, and applies the rules to the database using a *fair* strategy, generating new sequents. It stops when it finds a sequent that subsumes the goal, or when the database is saturated.

This design makes it possible to use the same back end for different logics. While Imogen now only supports two front ends, intuitionistic first-order logic and an optimized front end for the propositional fragment, it would be straightforward to extend

to other logics. We are currently in the process of adding front ends for first-order logic with constraints, and first-order logic with induction.

4.1 The Front End: Rule Generation and Matching

The front end has two distinct tasks. The first is to generate the initial rules and sequents given an input formula and a focusing strategy. This is achieved by, for each synthetic connective, evaluating the inference rules of Section 2 in the backward direction. Each block of a potential backward proof becomes an inference rule.

The second is to define the main functions outlined in the last section: subsumption, contraction, and rule matching. Subsumption can be an expensive operation, but is straightforward to implement. Contraction can be problematic because if a sequent has many antecedents with the same label or predicate symbol, there can be an exponential number of contraction instances. In such cases, it is not uncommon for Imogen to generate tens of thousands of contraction instances of a single sequent.

To implement the function `match` of Definition 3, we use the technique of *partial rule application*. Instead of having a fixed rule set and matching all the hypotheses simultaneously, we have an expanding rule set, and match one premise at a time. The match of a rule with n premises yields a new residual rule with $n - 1$ premises.

Example 3. *Matching rule and sequent*

$$\frac{p(X,Y) \longrightarrow q(X,Y) \quad q(X,Y) \longrightarrow \cdot}{r(X,Y) \longrightarrow \cdot} \qquad q(Z,c), p(c,Z) \longrightarrow q(c,Z)$$

yields the new inference rule

$$\frac{q(c,Y) \longrightarrow \cdot}{q(Y,c), r(c,Y) \longrightarrow \cdot}$$

Matching the new rule against $q(c,d) \longrightarrow q(d,d)$ *yields the new sequent* $q(d,c), r(c,d) \longrightarrow q(d,d)$.

Similar to contraction, if both a rule and sequent have multiple instances of the same label or predicate, matching can produce an inordinate number of new rules or sequents.

4.2 The Back End: Rule Application and Subsumption

The back end takes the initial sequents and rules from the front end, along with the definitions of matching, subsumption and contraction. Then it uses a modified form of the *Otter loop* to search for proofs.

The Otter Loop. The "Otter loop" is a general strategy for automated reasoning using forward inference. In our version, there are two databases of sequents, called *kept* and *active*, and two databases of rules (because of our partial rule matching strategy) also so named. The *active sequents* (AS), consist of all the sequents that have already been

matched to rules in *active rules* (AR). Symmetrically, the active rules are the rules that have been matched to the active sequents. The other databases, the *kept* rules (KR) and sequents (KS), have not yet been considered for matching. A step of the loop proceeds as follows, as shown in the figure below. Imogen chooses either a kept sequent or a kept rule according to some fair strategy. Suppose it chooses a sequent. Then we are in the situation in diagram. The sequent is added to the active sequents, and then is matched to all active rules. This matching will generate new sequents (when the matched rule has a single premise), and new rules (when the matched rule has multiple premises). The new rules and sequents are added to the respective kept databases. A symmetric process occurs when choosing a kept rule.

4.3 Subsumption

Redundancy elimination is an important part of an efficient implementation of the polarized inverse method. Imogen performs subsumption in a variety of ways. The first is *forward subsumption*: New sequents generated during the matching process that are subsumed by existing sequents are never added to the kept database. Another form of subsumption occurs when a new sequent subsumes an existing active or kept sequent. There are two forms of backward subsumption in Imogen. The first, simply called *backward subsumption* is where we delete the subsumed sequent from the database. In *recursive backward subsumption* we delete not only the subsumed sequent, but all of that sequent's descendents except those justifying the subsuming sequent. The idea is that Imogen, with the new, stronger sequent, will eventually recreate equal or stronger forms of the rules and sequents that were deleted. A final version of subsumption is called *rule subsumption*. Rule subsumption occurs when a new sequent subsumes the conclusion of an inference rule. In this case, whatever the content of the premises, the resulting conclusion would be forward subsumed, as matching only instantiates variables and adds to the antecedents. Thus, such a rule can be safely deleted.

Theorem 3. *If there exists a derivation of A, then there exists a derivation that respects forward, backward, and rule subsumption.*

Proof. For forward and backward subsumption, the proof is analogous to the one given by Degtyarev and Voronkov [6]. Since each sequent that could be generated by a subsumed rule would itself be subsumed, their argument extends easily to our framework. □

For recursive backward subsumption, while the soundness is clear, it is not as easy to see that our strategy is still complete.

Theorem 4. *Recursive backward subsumption is nondeterministically complete. That is, if the database saturates without subsuming the goal, then the goal can not be derived.*

Proof. For every recursively deleted sequent we either retain a stronger sequent, or we retain the possibility to recreate a stronger sequent. For the database to be saturated, we must have subsumed or recreated all the deleted sequents. □

This is enough to obtain the correctness of our prover. By soundness (and, in addition, through an independently verifiable natural deduction proof object) we have a proof

when the goal is subsumed. If the database is saturated without subsuming the goal, there cannot be a proof. We conjecture that recursive backward subsumption is also complete in the stronger sense that if there is a proof we could in principle always find it (since rule and sequent selection are fair), but we do not at present have a rigorous proof.

Besides fairness, the proofs of completeness under the various forms of redundancy elimination rely mainly on the following property of the (derived) rules used in the forward direction: If the premise of a rule is subsumed, either the conclusion is already subsumed or we can re-apply the rule and obtain a new conclusion which subsumes the old one.

4.4 Other Features

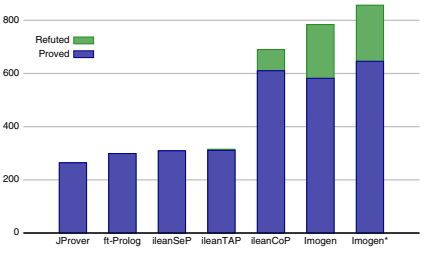
The back end implements a few other notable features. In a backward proof the antecedents of the goal sequent will occur in every sequent in the proof after the initial stabilization phase. We can globalize these antecedents [5], which reduces the space required to store sequents and avoids unnecessary operations on them. Imogen implements a variety of term indexing algorithms [7], including path and substitution tree indexing to quickly retrieve elements of the databases. Experimental results show that in our case path indexing is more efficient than substitution tree indexing. The back end also maintains a descendent graph of the rules and sequents. This graph is used by the front end to reconstruct a natural deduction proof term that can be checked by an external tool.

5 Performance Evaluation

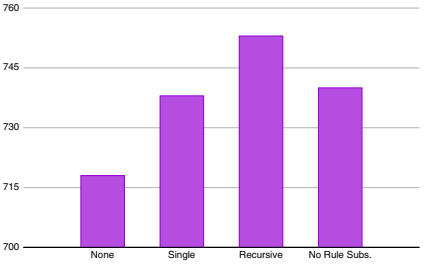
We now give some performance statistics and internal comparisons of the effects of different optimizations. All of the Imogen statistics from this paper are from a 2.4 Ghz Intel Macintosh, Darwin 9.6.0, with 2Gb of memory. Imogen is written in Standard ML, and is compiled with MLton.

ILTP. We evaluated Imogen on ILTP, the Intuitionistic Logic Theorem Proving library [16]. The statistics from the ILTP website [15] are shown below. Currently the library gives detailed results for 6 intuitionistic theorem provers on 2550 problems, with a time limit of 10 minutes. The other provers from ILTP use various optimizations of backward search. The non-Imogen statistics were run on a Xeon 3.4 GHz Linux, Mandrake 10.2. The amount of memory is not given on the website. The first Imogen statistic assign negative polarity to all atoms, which is the default behavior. The final bar, marked Imogen*, assigns negative polarity to all atoms, tries to prove it for 60 seconds and reverts to the default assignment if neither proof nor refutation has been found.

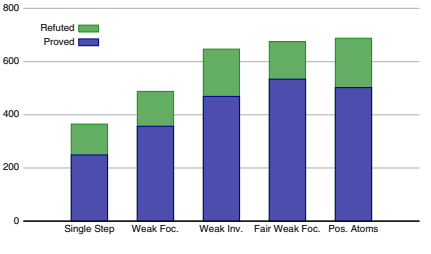
Note that, as usual, there are many theorems backwards methods can solve instantly that Imogen can not solve, and vice versa. We only display total numbers due to space constraints. The ILTP authors will include Imogen statistics in the next release of their library. Besides solving the most total problems, Imogen does much better than other provers at disproving non-theorems. This is a similar result to Imogen's intuitionistic propositional theorem prover described in McLaughlin and Pfenning [12]. Overall, iLeanCoP solved 690 problems, while Imogen solved 784 and Imogen* 857.



Subsumption. The following table shows the performance of Imogen with different settings for subsumption. The first three columns are for backward subsumption settings. The last column does no rule subsumption.



Polarization. One benefit of the polarized inverse method is that it is simple to simulate different focusing strategies using appropriate placement of double shifts. For instance, if we wish to measure the effect of the inversion phase without the focusing phase, or vice versa, we can strategically insert double shifts $\downarrow\uparrow$ or $\uparrow\downarrow$ at the locations where focusing (or inversion) would take place. The double shifts will break the current phase and generate a block boundary. The following table gives the performance of some of these strategies, again using 10 second timeouts. *Single Step* simulates the unfocused inverse method. *Weak Focusing* makes all focusing phases complete, but breaks the inversion phases into single steps. *Weak Inversion* makes the inversion phase complete, but breaks the focusing phase into single steps. *Fair Weak Focusing* is like weak focusing but allows the initial stabilization phase to run unchecked. In all of these experiments, we assigned negative polarity to all atoms. *Positive Atoms* makes all atoms positive, but otherwise does no unnecessary shifting.



6 Conclusion

In this paper we presented a basis for forward reasoning using the polarized inverse method, and demonstrated its practical effectiveness in the case of intuitionistic logic. In related work, Pientka et. al. [14] describe an experimental implementation of a focused inverse method for LF. Chaudhuri [3] describes a focused inverse method prover for linear logic. Earlier work is by Tammet [18] who describes an implementation of a forward intuitionistic theorem prover. We did not compare Imogen to his system because it is not part of ILTP. According to the ILTP website [15], the existing implementation, called Gandalf, is unsound.

Our work is by no means complete. While the current implementation is flexible with polarities, finding an optimal assignment of polarities needs to be studied. We now have only simple-minded heuristics for selecting the polarity of atoms, conjunctions, and inserting shifts. It is known, for instance [5], that using positive atoms simulates backward chaining in the inverse method. In our experiments however, we find that Imogen performs poorly on some problems that backchaining solves quickly. Given the dramatic effect of such choices in propositional logic [12], this promises significant potential for improvement.

Another optimization to consider would be to determine a subordination relation on propositions [13]. This would prune the search space by deleting or strengthening sequents of the form $\Gamma, p \longrightarrow q$ if no proof of q could depend on a proof of p as determined by the subordination relation.

Acknowledgments. This work has been partially supported by the Air Force Research Laboratory grant FA87500720028 *Accountable Information Flow via Explicit Formal Proof* and the National Science Foundation grant NSF-0716469 *Manifest Security*. The authors would like to thank Kaustuv Chaudhuri for sharing his experience with the focused inverse method for linear logic, Jens Otten for his help with the ILTP library, Geoff Sutcliffe for help with the TPTP library, and finally the anonymous reviewers that gave many constructive comments on a previous draft of this paper.

References

1. Andreoli, J.-M.: Logic programming with focusing proofs in linear logic. Journal of Logic and Computation 2(3), 297–347 (1992)
2. Andreoli, J.-M.: Focussing and proof construction. Annals of Pure and Applied Logic 107(1-3), 131–163 (2001)
3. Chaudhuri, K.: The Focused Inverse Method for Linear Logic. PhD thesis, Carnegie Mellon University. Technical report CMU-CS-06-162 (December 2006)
4. Chaudhuri, K., Pfenning, F.: Focusing the inverse method for linear logic. In: Ong, L. (ed.) CSL 2005. LNCS, vol. 3634, pp. 200–215. Springer, Heidelberg (2005)
5. Chaudhuri, K., Pfenning, F., Price, G.: A logical characterization of forward and backward chaining in the inverse method. Journal of Automated Reasoning 40(2-3), 133–177 (2008)
6. Degtyarev, A., Voronkov, A.: The inverse method. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 4, vol. I, pp. 179–272. Elsevier Science, Amsterdam (2001)

7. Graf, P.: Term Indexing. LNCS, vol. 1053. Springer, Heidelberg (1996)
8. Howe, J.M.: Proof Search Issues in Some Non-Classical Logics. PhD thesis, University of St. Andrews, Scotland (1998)
9. Lamarche, F.: Games semantics for full propositional linear logic. In: Logic in Computer Science, pp. 464–473. IEEE Computer Society Press, Los Alamitos (1995)
10. Liang, C., Miller, D.: Focusing and polarization in intuitionistic logic. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 451–465. Springer, Heidelberg (2007)
11. Maslov, S.Y.: An inverse method for establishing deducibility in classical predicate calculus. Soviet Mathematical Doklady 5, 1420–1424 (1964)
12. McLaughlin, S., Pfenning, F.: Imogen: Focusing the polarized inverse method for intuitionistic propositional logic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS, vol. 5330, pp. 174–181. Springer, Heidelberg (2008)
13. Pfenning, F., Schürmann, C.: System description: Twelf — A meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)
14. Pientka, B., Li, D.X., Pompigne, F.: Focusing the inverse method for LF: a preliminary report. In: International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (2007)
15. Raths, T., Otten, J.: The ILTP Library, `http://www.iltp.de`
16. Raths, T., Otten, J., Kreitz, C.: The ILTP problem library for intuitionistic logic. Journal of Automated Reasoning 38(1-3), 261–271 (2007)
17. Suttner, C., Sutcliffe, G.: The TPTP problem library: TPTP v2.0.0. Technical Report AR-97-01, Institut für Informatik, TU München (1997)
18. Tammet, T.: A resolution theorem prover for intuitionistic logic. In: McRobbie, M.A., Slaney, J.K. (eds.) CADE 1996. LNCS (LNAI), vol. 1104, pp. 2–16. Springer, Heidelberg (1996)
19. Zeilberger, N.: Focusing and higher-order abstract syntax. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 359–369. ACM, New York (2008)

A Refined Resolution Calculus for CTL

Lan Zhang, Ullrich Hustadt, and Clare Dixon*

Department of Computer Science, University of Liverpool
Liverpool, L69 3BX, UK
{Lan.Zhang, U.Hustadt, CLDixon}@liverpool.ac.uk

Abstract. In this paper, we present a refined resolution-based calculus for Computation Tree Logic (CTL). The calculus requires a polynomial time computable transformation of an arbitrary CTL formula to an equisatisfiable clausal normal form formulated in an extension of CTL with indexed existential path quantifiers. The calculus itself consists of a set of resolution rules which can be used as the basis for an EXPTIME decision procedure for the satisfiability problem of CTL. We prove soundness and completeness of the calculus. In addition, we introduce CTL-RP, our implementation of the calculus as well as some experimental results.

1 Introduction

Computation Tree Logic (CTL) [7] is a propositional branching-time temporal logic whose underlying model of time is a choice of possibilities branching into future. There are many important applications that can be represented and verified in CTL such as digital circuit verification [8], and the analysis of real time and concurrent systems [15].

The calculus $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$ for CTL introduced in this paper is a refinement of an earlier resolution calculus [5] for CTL. It involves transformation to a normal form, called Separated Normal Form with Global Clauses for CTL, $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ for short, and the application of *step* and *eventuality* resolution rules dealing with constraints on next states and on future states, respectively. We have improved the earlier calculus [5] in the following aspects. A technique introduced in [5] is the use of indices as part of a CTL normal form. We give a formal interpretation of indices and formal semantics for the indexed normal form, $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$, which is missing from [5]. An ordering and a selection function are introduced into the calculus which allow us to prune the search space of our prover. We show that our calculus $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$ is sound, complete and terminating. Furthermore, using our completeness proof we can show that two eventuality resolution rules in [5] are redundant. A detailed complexity analysis of the calculus is provided, which is absent for the earlier calculus. Finally, we have implemented $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$ in our theorem prover CTL-RP whereas no implementation was provided for the earlier calculus in [5]. We also present some experimental results for CTL-RP in this paper.

The rest of this paper is organised as follows. We first present the syntax and semantics of CTL in Section 2 and then introduce a normal form for CTL,

* This work was supported by EPSRC grant EP/D060451/1.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 245–260, 2009.

$\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$, in Section 3. In Section 4 the calculus $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$ is presented. We provide proofs for soundness and completeness of $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$ in Section 5. Section 6 discusses our theorem prover CTL-RP and the comparison between CTL-RP and a tableau theorem prover for CTL. Finally, related work is discussed and conclusions are drawn in Section 7.

2 Syntax and Semantics of CTL

In this paper, we use the syntax and semantics of CTL introduced by Clarke and Emerson in [7]. The language of CTL is based on a set of atomic propositions P_{PL}; propositional constants, **true** and **false**, and boolean operators, $\wedge, \vee, \Rightarrow$, and $\neg$ ($\wedge$ and $\vee$ are associative and commutative); temporal operators $\Box$ (always in the future), $\bigcirc$ (at the next moment in time), $\Diamond$ (eventually in the future), $\mathcal{U}$ (until), and $\mathcal{W}$ (unless); and path quantifiers **A** (for all future paths) and **E** (for some future path).

The set of *(well-formed) formulae of CTL* is inductively defined as follows: **true** and **false** as well as all atomic propositions in P_{PL} are CTL formulae; if φ and ψ are CTL formulae, then so are $\neg\varphi$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \Rightarrow \psi)$, $\mathbf{A}\Box\varphi$, $\mathbf{A}\Diamond\varphi$, $\mathbf{A}\bigcirc\varphi$, $\mathbf{A}(\varphi\,\mathcal{U}\,\psi)$, $\mathbf{A}(\varphi\,\mathcal{W}\,\psi)$, $\mathbf{E}\Box\varphi$, $\mathbf{E}\Diamond\varphi$, $\mathbf{E}\bigcirc\varphi$, $\mathbf{E}(\varphi\,\mathcal{U}\,\psi)$, and $\mathbf{E}(\varphi\,\mathcal{W}\,\psi)$.

Formulae of CTL over P_{PL} are interpreted in *model structures*, $M = \langle S, R, L\rangle$, where S is a set of *states*; R is a total binary *accessibility relation* over S; and $L : S \to 2^{\mathsf{P}_{\mathsf{PL}}}$ is an *interpretation function* mapping each state to the set of atomic propositions true at that state. An infinite path χ_{s_i} is an infinite sequence of states $s_i, s_{i+1}, s_{i+2}, \ldots$ such that for every $j \geq i, (s_j, s_{j+1}) \in R$.

The satisfaction relation $\models$ between a pair consisting of a model structure M and a state $s_i \in S$, and a CTL formula is inductively defined as follows:

$\langle M, s_i\rangle \models \mathbf{true}$ $\qquad$ $\langle M, s_i\rangle \not\models \mathbf{false}$

$\langle M, s_i\rangle \models p$ iff $p \in L(s_i)$ for an atomic proposition $p \in \mathsf{P}_{\mathsf{PL}}$

$\langle M, s_i\rangle \models \neg\varphi$ iff $\langle M, s_i\rangle \not\models \varphi$

$\langle M, s_i\rangle \models (\varphi \vee \psi)$ iff $\langle M, s_i\rangle \models \varphi$ or $\langle M, s_i\rangle \models \psi$

$\langle M, s_i\rangle \models \mathbf{E}\bigcirc\psi$ iff there exists a path χ_{s_i} such that $\langle M, s_{i+1}\rangle \models \psi$

$\langle M, s_i\rangle \models \mathbf{A}(\varphi\,\mathcal{U}\,\psi)$ iff for every path χ_{s_i} there exists $s_j \in \chi_{s_i}$ such that $\langle M, s_j\rangle \models \psi$ and for every $s_k \in \chi_{s_i}$, if $i \leq k < j$ then $\langle M, s_k\rangle \models \varphi$

$\langle M, s_i\rangle \models \mathbf{E}(\varphi\,\mathcal{U}\,\psi)$ iff there exists a path χ_{s_i} and there exists $s_j \in \chi_{s_i}$ such that $\langle M, s_j\rangle \models \psi$ and for every $s_k \in \chi_{s_i}$, if $i \leq k < j$ then $\langle M, s_k\rangle \models \varphi$

In addition, we use the usual equivalences to define the semantics of $\wedge$, $\Rightarrow$ and other boolean operators, and the following equivalences to define the remaining operators of CTL.

$$\mathbf{A}\Diamond\varphi \equiv \mathbf{A}(\mathbf{true}\,\mathcal{U}\,\varphi) \qquad \mathbf{E}\Diamond\varphi \equiv \mathbf{E}(\mathbf{true}\,\mathcal{U}\,\varphi)$$
$$\mathbf{A}\Box\varphi \equiv \neg\mathbf{E}\Diamond\neg\varphi \qquad \mathbf{E}\Box\varphi \equiv \neg\mathbf{A}\Diamond\neg\varphi$$
$$\mathbf{A}(\varphi\,\mathcal{W}\,\psi) \equiv \neg\mathbf{E}(\neg\psi\,\mathcal{U}\,(\neg\varphi \wedge \neg\psi)) \qquad \mathbf{E}(\varphi\,\mathcal{W}\,\psi) \equiv \neg\mathbf{A}(\neg\psi\,\mathcal{U}\,(\neg\varphi \wedge \neg\psi))$$
$$\mathbf{A}\bigcirc\varphi \equiv \neg\mathbf{E}\bigcirc\neg\varphi$$

3 Normal Form

Our calculus $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$ operates on formulae in a clausal normal form, called Separated Normal Form with Global Clauses for CTL, denoted by $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$. The language of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses is defined over an extension of CTL in which we label existential path quantifiers with an index *ind* taken from a countably infinite index set Ind and it consists of formulae of the following form.

$$\mathbf{A}\Box(\mathbf{start} \Rightarrow \bigvee_{j=1}^{k} m_j) \qquad \text{(initial clause)}$$

$$\mathbf{A}\Box(\mathbf{true} \Rightarrow \bigvee_{j=1}^{k} m_j) \qquad \text{(global clause)}$$

$$\mathbf{A}\Box(\bigwedge_{i=1}^{n} l_i \Rightarrow \mathbf{A}\bigcirc \bigvee_{j=1}^{k} m_j) \qquad (\mathbf{A}\text{-step clause})$$

$$\mathbf{A}\Box(\bigwedge_{i=1}^{n} l_i \Rightarrow \mathbf{E}_{\langle ind\rangle}\bigcirc \bigvee_{j=1}^{k} m_j) \qquad (\mathbf{E}\text{-step clause})$$

$$\mathbf{A}\Box(\bigwedge_{i=1}^{n} l_i \Rightarrow \mathbf{A}\Diamond l) \qquad (\mathbf{A}\text{-sometime clause})$$

$$\mathbf{A}\Box(\bigwedge_{i=1}^{n} l_i \Rightarrow \mathbf{E}_{\langle ind\rangle}\Diamond l) \qquad (\mathbf{E}\text{-sometime clause})$$

where $k \geq 0$, $n > 0$, **start** is a propositional constant, l_i $(1 \leq i \leq n)$, m_j $(1 \leq j \leq k)$ and l are literals, that is, atomic propositions or their negation, and *ind* is an element of Ind. As all clauses are of the form $\mathbf{A}\Box(P \Rightarrow D)$ we often simply write $P \Rightarrow D$ instead. We call a clause which is either an initial, a global, an **A**-step, or an **E**-step clause a *determinate clause*. The formula $\mathbf{A}\Diamond(\neg)l$ is called an **A**-*eventuality* and the formula $\mathbf{E}_{\langle ind\rangle}\Diamond(\neg)l$ is called an **E**-*eventuality*.

To provide a semantics for $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$, we extend model structures $\langle S, R, L\rangle$ to $\langle S, R, L, [\_], s_0\rangle$ where s_0 is an element of S and $[\_] : \mathsf{Ind} \to 2^{(S\times S)}$ maps every index $ind \in \mathsf{Ind}$ to a *successor function* $[ind]$ which is a total functional relation on S and a subset of R, that is, for every $s \in S$, there exists only one state $s' \in S, (s, s') \in [ind]$ and $(s, s') \in R$. An infinite path $\chi_{s_i}^{\langle ind\rangle}$ is an infinite sequence of states $s_i, s_{i+1}, s_{i+2}, \ldots$ such that for every $j \geq i, (s_j, s_{j+1}) \in [ind]$. The semantics of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ is then defined as shown below as an extension of the semantics of CTL defined in Section 2. Although the operators $\mathbf{E}_{\langle ind\rangle}\Box$, $\mathbf{E}_{\langle ind\rangle}\mathcal{U}$ and $\mathbf{E}_{\langle ind\rangle}\mathcal{W}$ do not appear in the normal form, we state their semantics, because they occur in the normal form transformation. (The semantics of the remaining operators is analogous to that given previously but in the extended model structure $\langle S, R, L, [\_], s_0\rangle$.)

$\langle M, s_i\rangle \models \mathbf{start}$ iff $s_i = s_0$

$\langle M, s_i\rangle \models \mathbf{E}_{\langle ind\rangle}\bigcirc\psi$ iff there exists a path $\chi_{s_i}^{\langle ind\rangle}$ such that $\langle M, s_{i+1}\rangle \models \psi$

$\langle M, s_i\rangle \models \mathbf{E}_{\langle ind\rangle}\Diamond\psi$ iff $\langle M, s_i\rangle \models \mathbf{E}_{\langle ind\rangle}(\mathbf{true}\,\mathcal{U}\,\psi)$

$\langle M, s_i\rangle \models \mathbf{E}_{\langle ind\rangle}\Box\psi$ iff there exists a path $\chi_{s_i}^{\langle ind\rangle}$ and for every $s_j \in \chi_{s_i}^{\langle ind\rangle}$ if $i \leq j$, then $\langle M, s_j\rangle \models \psi$

$\langle M, s_i\rangle \models \mathbf{E}_{\langle ind\rangle}(\varphi\,\mathcal{U}\,\psi)$ iff there exists a path $\chi_{s_i}^{\langle ind\rangle}$ and there exists $s_j \in \chi_{s_i}^{\langle ind\rangle}$ such that $\langle M, s_j\rangle \models \psi$ and for every $s_k \in \chi_{s_i}^{\langle ind\rangle}$, if $i \leq k < j$, then $\langle M, s_k\rangle \models \varphi$

$\langle M, s_i\rangle \models \mathbf{E}_{\langle ind\rangle}(\varphi\,\mathcal{W}\,\psi)$ iff $\langle M, s_i\rangle \models \mathbf{E}_{\langle ind\rangle}\Box\varphi$ or $\langle M, s_i\rangle \models \mathbf{E}_{\langle ind\rangle}(\varphi\,\mathcal{U}\,\psi)$

A $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ formula φ is *satisfiable in a model structure* $M = \langle S, R, L, [\_], s_0 \rangle$, iff $M, s_0 \models \varphi$ and φ is called *satisfiable* iff there exists a model structure M such that φ is satisfied in M.

We have defined a set of transformation rules which allows us to transform an arbitrary CTL formula into an equi-satisfiable set of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses [19]. The transformation rules are similar to those in [5,12], but modified to allow for global clauses. Basically, we use the following techniques to transform CTL formulae into $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$: introduce new indices into $\mathbf{E}$ path quantifiers; rename complex subformulae by new atomic propositions and link the truth of the new atomic proposition to the truth of the subformula it is renaming; remove combinations of temporal operators which are not allowed to appear in $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses using their semantic definition in terms of other operators. Two examples of transformation rules are the following: (i) $\mathbf{A}\Box(q \Rightarrow \mathbf{E}\bigcirc\varphi)$ is transformed into $\mathbf{A}\Box(q \Rightarrow \mathbf{E}_{\langle ind \rangle}\bigcirc\varphi)$, where *ind* is a new index, thus assigning an index to an $\mathbf{E}\bigcirc$ operator; (ii) $\mathbf{A}\Box(q_1 \Rightarrow \mathbf{E}_{\langle ind \rangle}(q_2 \,\mathcal{U}\, q_3))$ is transformed into $\mathbf{A}\Box(q_1 \Rightarrow q_3 \vee (q_2 \wedge p))$, $\mathbf{A}\Box(p \Rightarrow \mathbf{E}_{\langle ind \rangle}\bigcirc(q_3 \vee (q_2 \wedge p)))$ and $\mathbf{A}\Box(q_1 \Rightarrow \mathbf{E}_{\langle ind \rangle}\Diamond q_3)$, thereby eliminating an occurrence of the $\mathbf{E}_{\langle ind \rangle}\,\mathcal{U}$ operator. The two former formulae are not in $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ yet and require further transformations.

Theorem 1 (Normal form). *Every CTL formula φ can be transformed into an equi-satisfiable set T of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses with at most a linear increase in the size of the problem.*

4 The Clausal Resolution Calculus $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$

The resolution calculus $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$ consists of two types of resolution rules, *step* resolution rules, SRES1 to SRES8, and *eventuality* resolution rules, ERES1 and ERES2, as well as two *rewrite* rules, RW1 and RW2.

Motivated by refinements of propositional and first-order resolution [4], we restrict the applicability of step resolution rules by means of an atom ordering and a selection function. An *atom ordering* for $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$ is a well-founded and total ordering $\succ$ on the set P_{PL}. The ordering $\succ$ is extended to literals by identifying each positive literal p with the singleton multiset $\{p\}$ and each negative literal $\neg p$ with the multiset $\{p, p\}$ and comparing such multisets of atoms by using the multiset extension of $\succ$. Doing so, $\neg p$ is greater than p, but smaller than any literal q or $\neg q$ with $q \succ p$.

A literal l is (*strictly*) *maximal* with respect to a propositional disjunction C iff for every literal l' in C, $l' \not\succ l$ ($l' \not\succeq l$).

A *selection function* is a function S mapping every propositional disjunction C to a possibly empty subset $S(C)$ of the negative literals occurring in C. If $l \in S(C)$ for a disjunction C, then we say that l is *selected* in C.

In the following presentation of the rules of $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$, *ind* is an index in Ind, P and Q are conjunctions of literals, C and D are disjunctions of literals, neither of which contain duplicate literals, and l is a literal.

SRES1

$$\frac{P \Rightarrow \mathbf{A}\bigcirc(C \vee l) \qquad Q \Rightarrow \mathbf{A}\bigcirc(D \vee \neg l)}{P \wedge Q \Rightarrow \mathbf{A}\bigcirc(C \vee D)}$$

SRES2

$$\frac{P \Rightarrow \mathbf{E}_{\langle ind\rangle}\bigcirc(C \vee l) \qquad Q \Rightarrow \mathbf{A}\bigcirc(D \vee \neg l)}{P \wedge Q \Rightarrow \mathbf{E}_{\langle ind\rangle}\bigcirc(C \vee D)}$$

SRES3

$$\frac{P \Rightarrow \mathbf{E}_{\langle ind\rangle}\bigcirc(C \vee l) \qquad Q \Rightarrow \mathbf{E}_{\langle ind\rangle}\bigcirc(D \vee \neg l)}{P \wedge Q \Rightarrow \mathbf{E}_{\langle ind\rangle}\bigcirc(C \vee D)}$$

SRES4

$$\frac{\mathbf{start} \Rightarrow C \vee l \qquad \mathbf{start} \Rightarrow D \vee \neg l}{\mathbf{start} \Rightarrow C \vee D}$$

SRES5

$$\frac{\mathbf{true} \Rightarrow C \vee l \qquad \mathbf{start} \Rightarrow D \vee \neg l}{\mathbf{start} \Rightarrow C \vee D}$$

SRES6

$$\frac{\mathbf{true} \Rightarrow C \vee l \qquad Q \Rightarrow \mathbf{A}\bigcirc(D \vee \neg l)}{Q \Rightarrow \mathbf{A}\bigcirc(C \vee D)}$$

SRES7

$$\frac{\mathbf{true} \Rightarrow C \vee l \qquad Q \Rightarrow \mathbf{E}_{\langle ind\rangle}\bigcirc(D \vee \neg l)}{Q \Rightarrow \mathbf{E}_{\langle ind\rangle}\bigcirc(C \vee D)}$$

SRES8

$$\frac{\mathbf{true} \Rightarrow C \vee l \qquad \mathbf{true} \Rightarrow D \vee \neg l}{\mathbf{true} \Rightarrow C \vee D}$$

A step resolution rule, SRES1 to SRES8, is only applicable if one of the following two conditions is satisfied:

(C1) if l is a positive literal, then (i) l must be strictly maximal with respect to C and no literal is selected in $C \vee l$, and (ii) $\neg l$ must be selected in $D \vee \neg l$ or no literal is selected in $D \vee \neg l$ and $\neg l$ is maximal with respect to D; or

(C2) if l is a negative literal, then (i) l must be selected in $C \vee l$ or no literal is selected in $C \vee l$ and l is maximal with respect to C, and (ii) $\neg l$ must be strictly maximal with respect to D and no literal is selected in $D \vee \neg l$.

Note that these two conditions are identical modulo the polarity of l. If l in $C \vee l$ and $\neg l$ in $D \vee \neg l$ satisfy condition (C1) or condition (C2), then we say that l is *eligible* in $C \vee l$ and $\neg l$ is *eligible* in $D \vee \neg l$.

The rewrite rules RW1 and RW2 are defined as follows:

$$\textbf{RW1} \qquad \bigwedge_{i=1}^{n} m_i \Rightarrow \mathbf{A}\bigcirc\mathbf{false} \longrightarrow \mathbf{true} \Rightarrow \bigvee_{i=1}^{n} \neg m_i$$

$$\textbf{RW2} \qquad \bigwedge_{i=1}^{n} m_i \Rightarrow \mathbf{E}_{\langle ind\rangle}\bigcirc\mathbf{false} \longrightarrow \mathbf{true} \Rightarrow \bigvee_{i=1}^{n} \neg m_i$$

where $n \geq 1$ and each m_i, $1 \leq i \leq n$, is a literal.

The intuition of the eventuality resolution rule ERES1 below is to resolve an eventuality $\mathbf{A}\Diamond\neg l$, which states that $\Diamond\neg l$ is true on all paths, with a set of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses which together, provided that their combined left-hand sides were satisfied, imply that $\Box l$ holds on (at least) one path.

ERES1

$$\frac{P^{\dagger} \Rightarrow \mathbf{E}\bigcirc\mathbf{E}\Box l \qquad Q \Rightarrow \mathbf{A}\Diamond\neg l}{Q \Rightarrow \mathbf{A}(\neg(P^{\dagger})\,\mathcal{W}\,\neg l)}$$

where $P^\dagger \Rightarrow \mathbf{E}\bigcirc\mathbf{E}\Box l$ represents a set, $\Lambda_{\mathbf{E}\Box}$, of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses

$$\begin{array}{cc} P^1_1 \Rightarrow *\, C^1_1 & P^n_1 \Rightarrow *\, C^n_1 \\ \vdots & \vdots \\ P^1_{m_1} \Rightarrow *\, C^1_{m_1} & \cdots \quad P^n_{m_n} \Rightarrow *\, C^n_{m_n} \end{array}$$

with each $*$ either being empty or being an operator in $\{\mathbf{A}\bigcirc\} \cup \{\mathbf{E}_{\langle ind\rangle}\bigcirc \mid ind \in \mathsf{Ind}\}$ and for every i, $1 \leq i \leq n$, (i) $(\bigwedge_{j=1}^{m_i} C^i_j) \Rightarrow l$ and (ii) $(\bigwedge_{j=1}^{m_i} C^i_j) \Rightarrow (\bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} P^i_j)$ are provable. Furthermore, $P^\dagger = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} P^i_j$. Conditions (i) and (ii) ensure that the set $\Lambda_{\mathbf{E}\Box}$ of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses implies $P^\dagger \Rightarrow \mathbf{E}\bigcirc\mathbf{E}\Box l$.

Note that the conclusion of ERES1 is not stated in normal form. To present the conclusion of ERES1 in normal form, we use a new atomic proposition $w^{\mathbf{A}}_{\neg l}$ uniquely associated with the eventuality $\mathbf{A}\Diamond\neg l$. Then the conclusion of ERES1 can be represented by the following set of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses:

$$\begin{array}{rl} & \{w^{\mathbf{A}}_{\neg l} \Rightarrow \mathbf{A}\bigcirc(\neg l \vee \bigvee_{j=1}^{m_i} \neg P^i_j) \mid 1 \leq i \leq n\} \\ \cup & \{\mathbf{true} \Rightarrow \neg Q \vee \neg l \vee \bigvee_{j=1}^{m_i} \neg P^i_j \mid 1 \leq i \leq n\} \\ \cup & \{\mathbf{true} \Rightarrow \neg Q \vee \neg l \vee w^{\mathbf{A}}_{\neg l},\ w^{\mathbf{A}}_{\neg l} \Rightarrow \mathbf{A}\bigcirc(\neg l \vee w^{\mathbf{A}}_{\neg l})\}. \end{array}$$

The use of a proposition $w^{\mathbf{A}}_{\neg l}$ uniquely associated with the eventuality $\mathbf{A}\Diamond\neg l$ is important for the termination of our procedure. It allows us to represent all resolvents by ERES1 using a fixed set of propositions depending only on the initial set of clauses, i.e., n different $\mathbf{A}$-eventualities in the initial set of clauses require at most n new atomic propositions to represent resolvents by ERES1.

Similar to ERES1, the intuition underlying the ERES2 rule is to resolve an eventuality $\mathbf{E}_{\langle ind\rangle}\Diamond\neg l$, which states that $\Diamond\neg l$ is true on the path $\chi^{\langle ind\rangle}_{s_i}$, with a set of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses which together, provided that their combined left-hand sides were true, imply that $\Box l$ holds on the path $\chi^{\langle ind\rangle}_{s_{i+1}}$.

ERES2

$$\frac{\begin{array}{l} P^\dagger \Rightarrow \mathbf{E}_{\langle ind\rangle}\bigcirc(\mathbf{E}_{\langle ind\rangle}\Box l) \\ Q \Rightarrow \mathbf{E}_{\langle ind\rangle}\Diamond\neg l \end{array}}{Q \Rightarrow \mathbf{E}_{\langle ind\rangle}(\neg(P^\dagger)\,\mathcal{W}\,\neg l)}$$

where $P^\dagger \Rightarrow \mathbf{E}_{\langle ind\rangle}\bigcirc(\mathbf{E}_{\langle ind\rangle}\Box l)$ represents a set, $\Lambda^{ind}_{\mathbf{E}\Box}$, of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses which is analogous to the set $\Lambda_{\mathbf{E}\Box}$ but each $*$ is either empty or an operator in $\{\mathbf{A}\bigcirc, \mathbf{E}_{\langle ind\rangle}\bigcirc\}$ and for every i, $1 \leq i \leq n$, (i) $(\bigwedge_{j=1}^{m_i} C^i_j) \Rightarrow l$ and (ii) $(\bigwedge_{j=1}^{m_i} C^i_j) \Rightarrow (\bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} P^i_j)$ are provable. Furthermore, $P^\dagger = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} P^i_j$. Again, conditions (i) and (ii) ensure that the set $\Lambda^{ind}_{\mathbf{E}\Box}$ of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses implies the formula $P^\dagger \Rightarrow \mathbf{E}_{\langle ind\rangle}\bigcirc(\mathbf{E}_{\langle ind\rangle}\Box l)$.

Similarly, we use an atomic proposition $w^{ind}_{\neg l}$ uniquely associated with $\mathbf{E}_{\langle ind\rangle}\Diamond\neg l$ to represent the resolvent of ERES2 as the following set of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses:

$$\begin{array}{rl} & \{w^{ind}_{\neg l} \Rightarrow \mathbf{E}_{\langle ind\rangle}\bigcirc(\neg l \vee \bigvee_{j=1}^{m_i} \neg P^i_j) \mid 1 \leq i \leq n\} \\ \cup & \{\mathbf{true} \Rightarrow \neg Q \vee \neg l \vee \bigvee_{j=1}^{m_i} \neg P^i_j \mid 1 \leq i \leq n\} \\ \cup & \{\mathbf{true} \Rightarrow \neg Q \vee \neg l \vee w^{ind}_{\neg l},\ w^{ind}_{\neg l} \Rightarrow \mathbf{E}_{\langle ind\rangle}\bigcirc(\neg l \vee w^{ind}_{\neg l})\}. \end{array}$$

As for ERES1, the use of atomic propositions uniquely associated with **E**-eventualities allows us to represent all resolvents by ERES2 using a fixed set of atomic propositions depending only on the initial set of clauses.

The expensive part of applying ERES1 and ERES2 is finding sets of **A**-step, **E**-step and global clauses which can serve as premises for these rules, that is, for a given literal l stemming from some eventuality, to find sets of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses $\Lambda_{\mathbf{E}\Box}$, satisfying conditions (i) and (ii), or $\Lambda^{ind}_{\mathbf{E}\Box}$, satisfying conditions (i) and (ii). Such sets of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses are also called **E**-*loops in* l and the formula $\bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} P_j^i$ is called a *loop formula*. Algorithms to find such loops are described in [6].

A *derivation* from a finite set T of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses by $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$ is a sequence $T_0, T_1, T_2, \ldots$ of sets of clauses such that $T = T_0$ and $T_{i+1} = T_i \cup R_i$ where R_i is a set of clauses obtained as the conclusion of the application of a resolution rule to premises in T_i. A *refutation* of T (by $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$) is a derivation from T such that for some $i \geq 0$, T_i contains a contradiction, where a contradiction is either the formula **true** $\Rightarrow$ **false** or **start** $\Rightarrow$ **false**. A derivation *terminates* iff either a contradiction is derived or if no new clauses can be derived by further application of resolution rules.

5 Properties of $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$

The calculus $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$ is sound, complete and terminating. First, we show that $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$ is sound.

Theorem 2 (Soundness of $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$). *Let T be a set of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses. If there is a refutation of T by $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$, then T is unsatisfiable.*

Proof. The soundness of SRES1 to SRES4, ERES1 and ERES2 has been established in [5]. So we only need to prove the soundness of SRES5 to SRES8, RW1 and RW2.

Let $T_0, T_1, \ldots, T_n$ be a derivation from a set of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses $T = T_0$ by the calculus $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$. We will show by induction over the length of the derivation that if T_0 is satisfiable, then so is T_n.

For $T_0 = T$, the claim obviously holds. Now, consider the step of the derivation in which we derive T_{i+1} from T_i for some $i \geq 0$. Assume T_i is satisfiable and $M = \langle S, R, L, [\_], s_0 \rangle$ is a model structure satisfying T_i.

Assume $\mathbf{A}\Box(\mathbf{true} \Rightarrow C \vee l)$ and $\mathbf{A}\Box(\mathbf{start} \Rightarrow D \vee \neg l)$ are in T_i. Let T_{i+1} be obtained by an application of SRES5 to $\mathbf{A}\Box(\mathbf{true} \Rightarrow C \vee l)$ and $\mathbf{A}\Box(\mathbf{start} \Rightarrow D \vee \neg l)$, i.e., $T_{i+1} = T_i \cup \{\mathbf{A}\Box(\mathbf{start} \Rightarrow C \vee D)\}$. We show that M also satisfies T_{i+1}. Consider an arbitrary state $s \in S$. If s is not s_0, then obviously $\langle M, s \rangle \models \mathbf{start} \Rightarrow C \vee D$. Assume the state s is s_0. From $\langle M, s \rangle \models \mathbf{A}\Box(\mathbf{true} \Rightarrow C \vee l)$ and $\langle M, s \rangle \models \mathbf{A}\Box(\mathbf{start} \Rightarrow D \vee \neg l)$ and the semantics of $\mathbf{A}\Box$, **true**, $\Rightarrow$ and **start**, we obtain $\langle M, s \rangle \models C \vee l$ and $\langle M, s \rangle \models D \vee \neg l$. Then we conclude $\langle M, s \rangle \models C \vee D$. As s is s_0, then from the semantics of **start** we have $\langle M, s \rangle \models \mathbf{start} \Rightarrow C \vee D$. Since **start** $\Rightarrow C \vee D$ holds in s_0 and all other states, from the semantics of

$\mathbf{A}\Box$ we conclude $\langle M, s\rangle \models \mathbf{A}\Box(\mathbf{start} \Rightarrow C \vee D)$. Thus the model structure M satisfies T_{i+1}, T_{i+1} is satisfiable and SRES5 is sound. For rules SRES6 to SRES8, the proofs are analogous to that for SRES5.

Regarding RW1, from the semantics of $\mathbf{A}\bigcirc$ and **false** we obtain that the formula $\mathbf{A}\Box(\wedge_{i=1}^{n} Q_i \Rightarrow \mathbf{A}\bigcirc\mathbf{false})$ is true iff $\mathbf{A}\Box(\wedge_{i=1}^{n} Q_i \Rightarrow \mathbf{false})$ is true. This formula is propositionally equivalent to $\mathbf{A}\Box(\vee_{i=1}^{n} \neg Q_i)$ which in turn, by the semantics of $\Rightarrow$ and **true**, is equivalent to $\mathbf{A}\Box(\mathbf{true} \Rightarrow \vee_{i=1}^{n} \neg Q_i)$. The proof for RW2 is similar. □

Our proof of the completeness of $\mathsf{R}_{\mathrm{CTL}}^{\succ,S}$ makes use of (reduced) labelled behaviour graphs, which will be defined below. These graphs can be seen as finite representations of the set of all models of a set of $\mathrm{SNF}_{\mathrm{CTL}}^{\mathrm{g}}$ clauses.

Definition 1 (Labelled behaviour graph). *Let T be a set of* $\mathrm{SNF}_{\mathrm{CTL}}^{\mathrm{g}}$ *clauses and* $\mathrm{Ind}(T)$ *be the set of indices occurring in T. If* $\mathrm{Ind}(T)$ *is empty, then let* $\mathrm{Ind}(T) = \{ind\}$*, where ind is an arbitrary index in* Ind*. Given T and* $\mathrm{Ind}(T)$*, we construct a finite directed graph $H = (N, E)$, called a* labelled behaviour graph *for T.*

A node $n = (V, E_A, E_E)$ in H is a triple, where V, E_A, E_E are constructed as follows. Let V be a valuation of propositions occurring in T. Let E_A be a subset of $\{l \mid Q \Rightarrow \mathbf{A}\Diamond l \in T\}$ and E_E be a subset of $\{l_{\langle ind\rangle} \mid Q \Rightarrow \mathbf{E}_{\langle ind\rangle}\Diamond l \in T\}$. Informally E_A and E_E contain eventualities that need to be satisfied either in the current node or some node reachable from the current node.

To define the set of edges E of H we use the following auxiliary definitions. Let $n = (V, E_A, E_E)$ be a node in N. Let $\mathsf{R}_{\mathbf{A}}(n, T) = \{D \mid Q \Rightarrow \mathbf{A}\bigcirc D \in T$, and $V \models Q\}$. Note if V does not satisfy the left-hand side of any $\mathbf{A}$-step clause (i.e. $\mathsf{R}_{\mathbf{A}}(n, T) = \emptyset$), then there are no constraints from $\mathbf{A}$-step clauses on the next node of the node n and any valuation satisfies $\mathsf{R}_{\mathbf{A}}(n, T)$. Let $\mathsf{R}_{ind}(n, T) = \{D \mid Q \Rightarrow \mathbf{E}_{\langle ind\rangle}\bigcirc D \in T$ and $V \models Q\}$. Let $\mathsf{R}_g(T) = \{D \mid \mathbf{true} \Rightarrow D \in T\}$.

Let functions $\mathsf{Ev}_{\mathbf{A}}(V, T)$ and $\mathsf{Ev}_{\mathbf{E}}(V, T)$ be defined as $\mathsf{Ev}_{\mathbf{A}}(V, T) = \{l \mid Q \Rightarrow \mathbf{A}\Diamond l \in T$ and $V \models Q\}$ and $\mathsf{Ev}_{\mathbf{E}}(V, T) = \{l_{\langle ind\rangle} \mid Q \Rightarrow \mathbf{E}_{\langle ind\rangle}\Diamond l \in T$, and $V \models Q\}$, respectively.

Let functions $\mathsf{Unsat}_{\mathbf{A}}(E_A, V)$ and $\mathsf{Unsat}_{ind}(E_E, V)$ be defined as $\mathsf{Unsat}_{\mathbf{A}}(E_A, V) = \{l \mid l \in E_A$ and $V \not\models l\}$ and $\mathsf{Unsat}_{ind}(E_E, V) = \{l_{\langle ind\rangle} \mid l_{\langle ind\rangle} \in E_E$ and $V \not\models l\}$, respectively.

Then E contains an edge labelled by ind from a node (V, E_A, E_E) to a node (V', E'_A, E'_E) iff V' satisfies the set $\mathsf{R}_{\mathbf{A}}(n, T) \cup \mathsf{R}_{ind}(n, T) \cup \mathsf{R}_g(T)$, $E'_A = \mathsf{Unsat}_{\mathbf{A}}(E_A, V) \cup \mathsf{Ev}_{\mathbf{A}}(V', T)$ and $E'_E = \mathsf{Unsat}_{ind}(E_E, V) \cup \mathsf{Ev}_{\mathbf{E}}(V', T)$.

Let $\mathsf{R}_0(T) = \{D \mid \mathbf{start} \Rightarrow D \in T\}$. Then any node (V, E_A, E_E), where V satisfies the set $\mathsf{R}_0(T) \cup \mathsf{R}_g(T)$, $E_A = \mathsf{Ev}_{\mathbf{A}}(V, T)$ and $E_E = \mathsf{Ev}_{\mathbf{E}}(V, T)$, is an initial node of H. The labelled behaviour graph *for a set of* $\mathrm{SNF}_{\mathrm{CTL}}^{\mathrm{g}}$ *clauses T is the set of nodes and edges reachable from the initial nodes.*

To determine whether T is satisfiable and to be able to construct a CTL model structure from a labelled behaviour graph H for T, some nodes and subgraphs of H may have to be deleted. To this end we need to define one type of nodes

and two types of subgraphs which cannot contribute to the construction of a CTL model structure from H.

Definition 2 (Terminal node). *A node n in a labelled behaviour graph for T is a terminal node iff there exists an index $ind \in \mathrm{Ind}(T)$ such that no edges labelled with ind depart from n.*

Definition 3 (ind-labelled terminal subgraph for $l_{\langle ind \rangle}$). *For a labelled behaviour graph (N, E) for T, a subgraph (N', E') is an* ind-labelled terminal subgraph for $l_{\langle ind \rangle}$ *of (N, E) iff*

(ITS1) *$N' \subseteq N$ and $E' \subseteq E$;*
(ITS2) *for all nodes n, $n' \in N$ and edges $(n, ind', n') \in E$, $n' \in N'$ and $(n, ind', n') \in E'$ iff $n \in N'$ and $ind = ind'$; and*
(ITS3) *for every node $n = (V, E_A, E_E) \in N'$, $l_{\langle ind \rangle} \in E_E$ and $V \models \neg l$.*

Definition 4 (Terminal subgraph for l). *For a labelled behaviour graph (N, E) for T, a subgraph (N', E') is a* terminal subgraph for l *of (N, E) iff*

(TS1) *$N' \subseteq N$ and $E' \subseteq E$;*
(TS2) *for every node $n \in N'$ there exists some index $ind \in \mathrm{Ind}(T)$ such that for all edges $(n, ind, n') \in E$, $n' \in N'$ and $(n, ind, n') \in E'$; and*
(TS3) *for every node $n = (V, E_A, E_E) \in N'$, $l \in E_A$ and $V \models \neg l$.*

Figure 1 shows an example of an ind-labelled terminal subgraph for $q_{\langle ind \rangle}$ consisting of n_1, n_2, n_3, where $ind = 2$, and an example of a terminal subgraph for p consisting of n_1, n_3, n_4. (We assume the set of indices in the clause set T for this labelled behaviour graph is $\mathrm{Ind}(T) = \{1, 2\}$.)

Using the three definitions above, we are able to define a new type of behaviour graphs, namely *reduced labelled behaviour graphs*.

Definition 5 (Reduced labelled behaviour graph). *Given a labelled behaviour graph $H = (N, E)$ for a set of* $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ *clauses T, then the* reduced labelled behaviour graph *for T is the result of exhaustively applying the following* deletion rules *to H.*

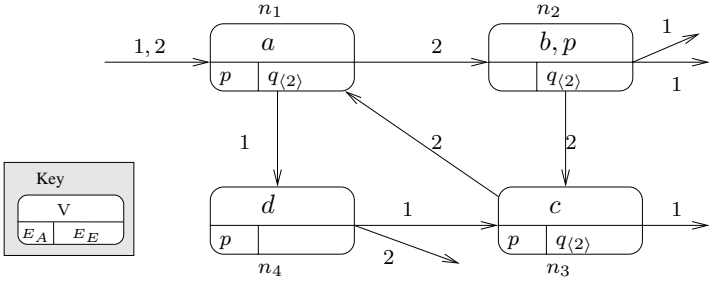


Fig. 1. A 2-labelled terminal subgraph for $q_{\langle 2 \rangle}$ consisting of n_1, n_2, n_3 and a terminal subgraph for p consisting of n_1, n_3, n_4

1. *If $n \in N$ is a terminal node with respect to an index in* $\mathrm{Ind}(T)$*, then delete n and every edge into or out of n.*
2. *If there is an ind-labelled terminal subgraph (N', E') for $l_{\langle ind \rangle}$ of H such that $ind \in \mathrm{Ind}(T)$ and $Q \Rightarrow \mathbf{E}_{\langle ind \rangle} \Diamond l \in T$, then delete every node $n \in N'$ and every edge into or out of nodes in N'.*
3. *If there is a terminal subgraph (N', E') for l of H such that $Q \Rightarrow \mathbf{A} \Diamond l \in T$, then delete every node $n \in N'$ and every edge into or out of nodes in N'.*

Lemma 1. *A set of* $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ *clauses T is unsatisfiable if and only if its reduced labelled behaviour graph H is empty.*

Proof (Sketch). We start by showing the 'if' part. If T is satisfiable, then there is a model structure $M = \langle S, R, L, [\_], s_0 \rangle$ satisfying T. We construct a labelled behaviour graph $H = (N, E)$ for T and inductively define a mapping h from M to H. Let P_T be the set of atomic propositions occurring in T. As the model structure M satisfies the clause set T, $L(s_0)$ must satisfy $\mathsf{R}_0(T) \cup \mathsf{R}_g(T)$, which means there must be an initial node $n_0 = (V_0, E_{A_0}, E_{E_0})$ in H, where $V_0 = L(s_0) \cap P_T$, $E_{A_0} = \mathsf{Ev}_{\mathbf{A}}(V_0, T)$ and $E_{E_0} = \mathsf{Ev}_{\mathbf{E}}(V_0, T)$, and we define $h(s_0) = n_0$.

Next, we assume that $h(s_i) = n_i = (V_i, E_{A_i}, E_{E_i})$ is in H and $(s_i, s_{i+1}) \in [ind]$. As the model structure M satisfies T, $L(s_{i+1})$ must satisfy $\mathsf{R}_{\mathbf{A}}(n_i, T) \cup \mathsf{R}_{ind}(n_i, T) \cup \mathsf{R}_g(T)$, which means there must be a node $n_{i+1}{=}(V_{i+1}, E_{A_{i+1}}, E_{E_{i+1}})$ in H, where $V_{i+1} = L(s_{i+1}) \cap P_T$, $E_{A_{i+1}} = \mathsf{Ev}_{\mathbf{A}}(V_{i+1}, T) \cup \mathsf{Unsat}_{\mathbf{A}}(E_{A_i}, V_i)$, $E_{E_{i+1}} = \mathsf{Ev}_{\mathbf{E}}(V_{i+1}, T) \cup \mathsf{Unsat}_{ind}(E_{E_i}, V_i)$, and we define $h(s_{i+1}) = n_{i+1}$. By the construction of the behaviour graph, the edge $(h(s_i), ind, h(s_{i+1}))$ is in H. Therefore, for every state $s \in S$, the node $h(s)$ is in H and for every pair $(s_i, s_{i+1}) \in R, i \geq 0$, the edge $(h(s_i), h(s_{i+1}))$ is in H. So, the graph $H^M = (N^M, E^M)$ such that $N^M = \{h(s) \mid s \in S\}$ and $E^M = \{(h(s_i), ind, h(s_{i+1})) \mid (s_i, s_{i+1}) \in [ind], ind \in \mathrm{Ind}(T), s_i, s_{i+1} \in S\}$ is a subgraph of H. Then we are able to show that H^M is finite and no deletion rules are applicable to H^M, so the labelled behaviour graph H for T cannot be reduced to an empty graph.

For the 'only if' part of the proof we need some additional definitions. By $RP(s_n)$ we denote a reverse path consisting of a finite sequence $s_n, s_{n-1}, \ldots, s_0$ of states in S and for every i, $0 \leq i \leq n-1, (s_i, s_{i+1}) \in R$. If $n = (V, E_A, E_E) \in N$, let the function $cs(n)$ return a state s of a CTL model structure such that $L(s) = V$.

Assume that the reduced labelled behaviour graph $H = (N, E)$ of T is non-empty, then we inductively construct a CTL model structure $M = \langle S, R, L, [\_], s_0 \rangle$ from H and a mapping h from M to H. The state s_0 of M is given by $s_0 = cs(n_0)$, where n_0 is an arbitrary initial node in H, and we define $h(s_0) = n_0$.

Suppose we have constructed the state s_i for M and $RP(s_i) = s_i, s_{i-1}, \ldots, s_0$. Then our task is to choose for each index $ind \in \mathrm{Ind}(T)$ a pair $(s_i, s_{i+1}) \in [ind]$ for M. Assume $h(s_i) = n$ and n has k ind-successors $\{n_1, n_2, \ldots, n_k\} (k > 0$ as otherwise n would be a terminal node in $H)$. Let S^{RP} be the set $\{s_j \mid s_{j-1}, s_j \in RP(s_i), h(s_{j-1}) = n, h(s_j) \in \{n_1, n_2, \ldots, n_k\}$ and $(s_{j-1}, s_j) \in [ind]\}$. We construct s_{i+1} as follows:

- if the set S^{RP} is empty, then $s_{i+1} = cs(n_1)$;
- else, let $s \in S^{SP}$ be the state such that the distance between s_i and s is the shortest among all the distances between s_i and a state in S^{RP} and $h(s) = n_m \in \{n_1, n_2, \ldots, n_k\}, 1 \leq m \leq k$, then
  - $s_{i+1} = cs(n_{m+1})$, if $m \neq k$;
  - $s_{i+1} = cs(n_1)$, if $m = k$.

By this algorithm, for an arbitrary path χ_{s_0}, if a node n is infinitely often used to construct states $s \in \chi_{s_0}$ and the index *ind* is infinitely often used to construct the next states of s on χ_{s_0}, then *ind*-successors of the node n are fairly chosen. This construction ensures that all eventualities will be satisfied in M.

Following the instructions we provided and using a breadth-first order for the construction, from the state s_0, a CTL model structure M is constructed from H, which satisfies T. □

Theorem 3 (Completeness of $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$). *If a set T of SNF$^{\mathrm{g}}_{\mathrm{CTL}}$ clauses is unsatisfiable, then T has a refutation using the resolution rules SRES1 to SRES8, ERES1 and ERES2 and the rewrite rules RW1 and RW2.*

Proof (Sketch). Let T be an unsatisfiable set of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses. The proof proceeds by induction on the sequence of applications of the deletion rules to the labelled behaviour graph of T. If the unreduced labelled behaviour graph is empty then we can obtain a refutation by applying step resolution rules SRES4, SRES5 and SRES8. Now suppose the labelled behaviour graph H is non-empty. The reduced labelled behaviour graph must be empty by Lemma 1, so there must be a node that can be deleted from H.

We show that for every application of a deletion rule to a behaviour graph H of T resulting in a smaller graph H'', there is a derivation from T by $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$ resulting in a set T' such that the behaviour graph H' for T' is a strict subgraph of H''. In particular, we can prove that the first deletion rule corresponds to a series of step resolution inferences by SRES1 to SRES8 deriving a clause of the form $\mathbf{true} \Rightarrow \mathbf{false}$, $P \Rightarrow \mathbf{A}\bigcirc\mathbf{false}$ or $Q \Rightarrow \mathbf{E}_{\langle ind \rangle}\bigcirc\mathbf{false}$. The rewrite rules RW1 and RW2 will replace $P \Rightarrow \mathbf{A}\bigcirc\mathbf{false}$ and $Q \Rightarrow \mathbf{E}_{\langle ind \rangle}\bigcirc\mathbf{false}$ by the simpler clauses $\mathbf{true} \Rightarrow \neg P$ and $\mathbf{true} \Rightarrow \neg Q$, respectively. We can show that removal of *ind*-labelled terminal subgraphs, i.e., the second deletion rule, and terminal subgraphs, i.e., the third deletion rule, corresponds to applications of ERES2 and ERES1, respectively.

Therefore, if T is unsatisfiable, then the reduced labelled behaviour graph H_{red} for T is empty and the sequence of applications of the deletion rules which reduces the labelled behaviour graph H for T to an empty H_{red} can be used to construct a refutation in $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$. □

The full completeness proof can be found in [19]. In addition to our completeness proof, there is also a proof to show that two eventuality resolution rules TRES1 and TRES2 of the earlier resolution calculus for CTL [5], are redundant. These rules are similar to ERES1 and ERES2 except their first premise uses **A** rather than **E** operators. A detailed explanation can be found in [19].

Theorem 4. *Any derivation from a set T of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses by the calculus $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$ terminates.*

Proof. Let T be constructed from a set P of n atomic propositions and a set *Ind* of m indices. Then the number of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses constructed from P and *Ind* is finite. We can have at most 2^{2n} initial clauses, 2^{2n} global clauses, 2^{4n} **A**-step clauses, $m \cdot 2^{4n}$ **E**-step clauses, $n \cdot 2^{2n+1}$ **A**-sometime clauses, and $m \cdot n \cdot 2^{2n+1}$ **E**-sometime clauses. In total, there could be at most $(m+1)2^{4n}+(m\cdot n+n+1)2^{2n+1}$ different $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses. Any derivation from a set of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses by the calculus $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$ will terminate when either no more new clauses can be derived or a contradiction is obtained. Since there is only a finitely bounded number of different $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses, one of these two conditions will eventually be true. □

Next we consider the complexity of $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$.

Theorem 5. *The complexity of a $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$-based decision procedure is in EXPTIME.*

Proof. Assume a set of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses is constructed from a set P of n propositions and a set *Ind* of m indices. The cost of deciding whether a step resolution rule can be applied to two determinate clauses is $\mathsf{A} = 4n + 1$ in the worst case, provided we can compute $S(C)$ in linear time, compare literals in constant time and check the identity of indices in constant time. From the proof of Theorem 4, we know the number of determinate clauses is at most $\mathsf{B} = 2^{2n}+2^{2n}+2^{4n}+m\cdot 2^{4n}$. Therefore, to naively compute a new clause from an application of some step resolution rule, we might need to look at $\mathsf{C} = \frac{\mathsf{B}(\mathsf{B}-1)}{2}$ combinations of two clauses and the associated cost is $(\mathsf{C} \cdot \mathsf{A})$. Moreover, to decide whether the resolvent is a new clause or not, we need to compare the resolvent with at most B clauses and the cost is $\mathsf{D} = \mathsf{B} \cdot (\mathsf{A} + 4n^2)$. In the worst case, where each pair of clauses generates a resolvent but the resolvent already exists and only the last pair of clauses gives a new clause, to gain a new clause from an application of some step resolution rule, the complexity is of the order $(\mathsf{C} \cdot \mathsf{A} \cdot \mathsf{D})$, that is, EXPTIME.

To compute a new clause from an application of some eventuality resolution rule, the complexity depends on the complexity of the so-called CTL loop search algorithm which computes premises for the eventuality resolution rules [6]. The CTL loop search algorithm is a variation of the PLTL loop search algorithm [9] which has been shown to be in EXPTIME and we can show that the complexity of the CTL loop search algorithm from [6] is also in EXPTIME. Generally speaking, each iteration of the CTL loop search algorithm is a saturation of the clause set, which is in EXPTIME, and there may be an exponential number of iterations required. Since a new clause is produced by an application of either step resolution or eventuality resolution, the complexity of generating a new clause is of the order EXPTIME. According to the proof of Theorem 4, there can be at most $(m + 1)2^{4n} + (m \cdot n + n + 1)2^{2n+1}$ different $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses. Therefore, the complexity of saturating a set of $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses and thereby deciding its satisfiability is in EXPTIME. □

6 Implementation and Experiments

We have implemented the transformation from CTL to $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ and the calculus $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$ in the prover CTL-RP. The implementation of $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$ follows the approach used in [13] to implement a resolution calculus for PLTL. In particular, we transform all $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$ clauses except **A**- and **E**-sometime clauses into first-order clauses. Then we are able to use first-order ordered resolution with selection to emulate step resolution. With respect to the eventuality resolution rules we have implemented the **E**-loop search algorithm in [19]. We again compute the results of applications of the eventuality resolution rules in the form of first-order clauses. To implement first-order ordered resolution with selection, we reuse the implementation provided by the first-order theorem prover SPASS 3.0 [16,18]. Regarding redundancy elimination, we adopt the following techniques from SPASS: tautology deletion, forward subsumption, backward subsumption and a modified version of matching replacement resolution. A formal description of the approach and related proofs are presented in detail in [19].

Besides CTL-RP, there is only one other CTL theorem prover that we know of, namely a CTL module for the Tableau Workbench (TWB) [1]. The Tableau Workbench is a general framework for building automated theorem provers for arbitrary propositional logics. It provides a number of pre-defined provers for a wide range of logics, for example, propositional logic, linear-time temporal logic and CTL. Regarding CTL, it implements a so-called one-pass tableau calculus for this logic which results in double-EXPTIME decision procedure [2]. Therefore the complexity of this CTL decision procedure is higher than the complexity of CTL-RP, which is EXPTIME. It should be noted that the prime aim of TWB is not efficiency.

There is no established way to evaluate the performance of CTL decision procedures nor is there a repository or random generator of CTL formulae that one might use for such an evaluation. We have therefore created three sets of benchmark formulae that we have used to compare CTL-RP version 00.09 with TWB version 3.4. The comparison was performed on a Linux PC with an Intel Core 2 E6400 CPU@2.13 GHz and 3GB main memory, using the Fedora 9 operating system.

Table 1. Performance of CTL-RP and TWB on sample CTL equivalences

	CTL equivalence	CTL-RP	TWB
1.	$\mathbf{A}\Box p \equiv \neg\mathbf{E}\Diamond\neg p$	0.008s	0.005s
2.	$\mathbf{E}\Box p \equiv \neg\mathbf{A}\Diamond\neg p$	0.008s	0.004s
3.	$\mathbf{E}\bigcirc(p \vee q) \equiv \mathbf{E}\bigcirc p \vee \mathbf{E}\bigcirc q$	0.005s	0.005s
4.	$\mathbf{A}\bigcirc p \equiv \neg\mathbf{E}\bigcirc\neg p$	0.004s	0.006s
5.	$\mathbf{E}(p\,\mathcal{U}\,q) \equiv q \vee (p \wedge \mathbf{E}\bigcirc\mathbf{E}(p\,\mathcal{U}\,q))$	0.049s	0.005s
6.	$\mathbf{A}(p\,\mathcal{U}\,q) \equiv q \vee (p \wedge \mathbf{A}\bigcirc\mathbf{A}(p\,\mathcal{U}\,q))$	0.068s	0.005s
7.	$\mathbf{E}\Diamond p \equiv \mathbf{E}(\mathbf{true}\ \mathcal{U}\,p)$	0.010s	0.008s
8.	$\mathbf{A}\Diamond p \equiv \mathbf{A}(\mathbf{true}\ \mathcal{U}\,p)$	0.010s	0.008s

The first set of benchmark formulae, CTL-BF1, consists of eight well-known equivalences between temporal formulae taken from [10]. The CTL equivalences themselves and the CPU time required by TWB and CTL-RP to prove each of them is shown in Table 1.

Both systems easily prove each of the formulae in less then 0.1 seconds, however, with TWB being significantly faster on two of the formulae.

We obtain the second set of benchmarks by randomly generating formulae consisting of a specification of a state transition system and properties of the system to be verified. In particular, let a *state specification* be a conjunction of literals l_i, $1 \leq i \leq 4$, with each l_i being an element of $\{a_i, \neg a_i\}$. Let a *transition specification* be a CTL formula in the form $\mathbf{A}\Box(s \Rightarrow \mathbf{A}\bigcirc(\bigvee_{i=1}^{n} s_i))$ or $\mathbf{A}\Box(s \Rightarrow \mathbf{E}\bigcirc(\bigvee_{i=1}^{n} s_i))$, where n is a randomly generated number between 1 and 3, and s and each s_i, $1 \leq i \leq n$ is a randomly generated state specification. Furthermore, let a *property specification* be a CTL formula of the form $*(\bigvee_{i=1}^{n} s_i)$, where $*$ is a randomly chosen element of $\{\mathbf{A}\bigcirc, \mathbf{E}\bigcirc, \mathbf{A}\Box, \mathbf{E}\Box, \mathbf{A}\Diamond, \mathbf{E}\Diamond\}$ or $(\bigvee_{i=1}^{n} s_i) * (\bigvee_{i=1}^{m} s_j)$, where $*$ is a randomly chosen element of $\{\mathbf{A}\,\mathcal{U}\,, \mathbf{E}\,\mathcal{U}\,\}$, n and m are randomly generated numbers between 1 and 2, and each s_i and s_j, $1 \leq i \leq n, 1 \leq j \leq m$, is a randomly generated state specification. CTL-BF2 consists of one hundred formulae with each formula being a conjunction (set) of 30 transition specifications and 5 property specifications. All one hundred formula are unsatisfiable. Figure 2 shows a graph indicating the CPU time in seconds required by TWB and CTL-RP to establish the unsatisfiability of each benchmark formula in CTL-BF2. For CTL-RP, each of the 100 benchmark formulae was solved in less than one CPU second. TWB, on the other hand, required more time for most of benchmark formulae and was not able to solve 21 of the benchmark formulae in less than 200 CPU seconds each, which was the time limit imposed for both provers. The results on CTL-BF2 show that CTL-RP can provide a proof for each benchmark

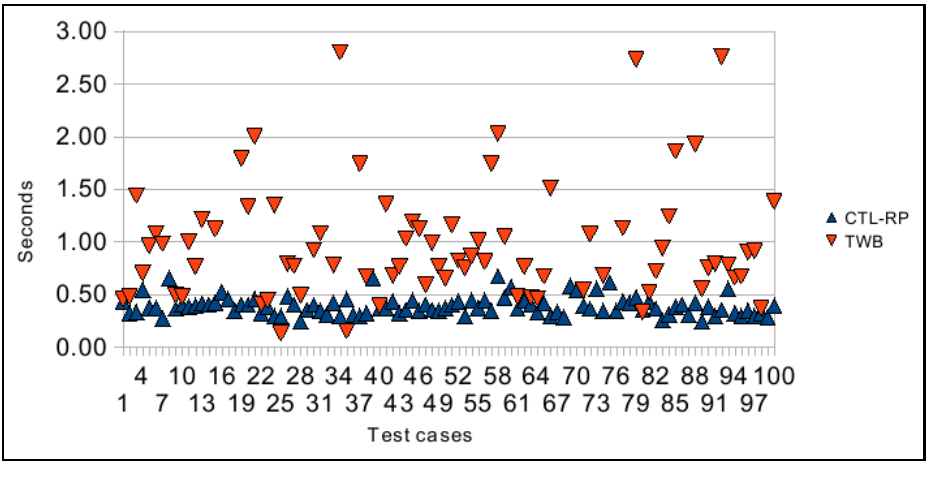


Fig. 2. Performance on the second set of benchmark formulae CTL-BF2

Table 2. Performance of CTL-RP and TWB on CTL-BF3

Property	CTL-RP	TWB
1	1.39s	-
2	192.86s	-
3	326.02s	-

formula in a reasonable time with the vast majority of formulae being solved in less than 0.50 seconds. In contrast, the performance of TWB is much more variable, with a high percentage of formulae not being solved.

The last set of benchmarks, CTL-BF3, is based on a protocol specification. Following the description of a network protocol, the Alternating Bit Protocol, in [14], we have specified this protocol in CTL and specified and verified the following three of its properties by CTL-RP and TWB.

$$1.\ \mathbf{A}(i\,\mathcal{U}\,a_0) \qquad 2.\ \mathbf{A}\Box(a_0 \Rightarrow \mathbf{A}(a_0\,\mathcal{U}\,a_1)) \qquad 3.\ \mathbf{A}\Box(a_1 \Rightarrow \mathbf{A}(a_1\,\mathcal{U}\,a_0))$$

where i represents that the receiver is in the starting state of the finite state transition system that represents the receiver's behaviour according to the protocol; a_0 represents that the receiver sends an acknowledgement with control bit 0; and a_1 represents that the receiver sends an acknowledgement with control bit 1.

While CTL-RP was able to establish the validity of each of the three benchmark formulae, as indicated in Table 2, TWB did not terminate within 20 hours of CPU time.

7 Related Work and Conclusions

CTL [10] was introduced by Emerson et al in the 1980s and now is a well-known branching-time temporal logic for the specification and verification of computational systems. Approaches to the satisfiability problem in CTL include automata techniques [17], tableau calculi [2,11] and a resolution calculus [5], developed by Bolotov.

Bolotov's calculus for CTL is based on the ideas underlying a resolution calculus for PLTL [12], which is implemented in the theorem prover TRP++ [13] using first-order techniques. Here, we have provided a refined clausal resolution calculus $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$ for CTL, based on [5]. Compared with [5], we provide a formal semantics for indices and $\mathrm{SNF}^{\mathrm{g}}_{\mathrm{CTL}}$. Moreover, we use an ordering and a selection function to restrict the applicability of step resolution rules and we have fewer eventuality resolution rules. We present a new completeness proof based on behaviour graphs. Our completeness proof demonstrates a closer relationship between applications of resolution rules and deletions on behaviour graphs. The proof shows that the additional eventuality resolution rules in [5], which are the most costly rules, are redundant. In addition, we prove that the complexity of a $\mathsf{R}^{\succ,S}_{\mathrm{CTL}}$-based decision procedure for CTL is EXPTIME. We have compared our implementation CTL-RP with another theorem prover, TWB, and the experimental results show good performance of CTL-RP. In future work, we plan to

create a scalable family of benchmarks and investigate more complicated specification and verification problems. Also, we intend to extend our clausal resolution calculus $\mathsf{R}_{\mathrm{CTL}}^{\succ,S}$ to other logics close to CTL, for example, ATL [3].

References

1. Abate, P., Goré, R.: The Tableaux Workbench. In: Cialdea Mayer, M., Pirri, F. (eds.) TABLEAUX 2003. LNCS, vol. 2796, pp. 230–236. Springer, Heidelberg (2003)
2. Abate, P., Goré, R., Widmann, F.: One-Pass Tableaux for Computation Tree Logic. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS, vol. 4790, pp. 32–46. Springer, Heidelberg (2007)
3. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. J. ACM 49(5), 672–713 (2002)
4. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Handbook of Automated Reasoning, vol. 1, pp. 19–99. Elsevier, Amsterdam (2001)
5. Bolotov, A.: Clausal Resolution for Branching-Time Temporal Logic. PhD thesis, Manchester Metropolitan University (2000)
6. Bolotov, A., Dixon, C.: Resolution for Branching Time Temporal Logics: Applying the Temporal Resolution Rule. In: Proc. TIME 2000, pp. 163–172. IEEE, Los Alamitos (2000)
7. Clarke, E.M., Emerson, E.A.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
9. Dixon, C.: Temporal Resolution Using a Breadth-First Search Algorithm. Annals of Mathematics and Artificial Intelligence 22(1-2), 87–115 (1998)
10. Emerson, E.A.: Temporal and Modal Logic. In: Handbook of Theoretical Computer Science, pp. 996–1072. Elsevier, Amsterdam (1990)
11. Emerson, E.A., Halpern, J.Y.: Decision Procedures and Expressiveness in the Temporal Logic of Branching Time. J. Comput. Syst. Sci. 30(1), 1–24 (1985)
12. Fisher, M., Dixon, C., Peim, M.: Clausal Temporal Resolution. ACM Transactions on Computational Logic 2(1), 12–56 (2001)
13. Hustadt, U., Konev, B.: TRP++: A Temporal Resolution Prover. In: Collegium Logicum, pp. 65–79. Kurt Gödel Society (2004)
14. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press, Cambridge (2004)
15. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, Heidelberg (1992)
16. Max-Planck-Institut Informatik: SPASS: An Automated Theorem Prover for First-Order Logic with Equality, `http://www.spass-prover.org/`
17. Vardi, M.Y., Wolper, P.: Automata-Theoretic Techniques for Modal Logics of Programs. J. Comput. Syst. Sci. 32(2), 183–221 (1986)
18. Weidenbach, C., Schmidt, R.A., Hillenbrand, T., Rusev, R., Topic, D.: System description: Spass version 3.0. In: Pfenning, F. (ed.) CADE 2007. LNCS, vol. 4603, pp. 514–520. Springer, Heidelberg (2007)
19. Zhang, L., Hustadt, U., Dixon, C.: First-order Resolution for CTL. Technical Report ULCS-08-010, Department of Computer Science, University of Liverpool (2008)

Fair Derivations in Monodic Temporal Reasoning

Michel Ludwig[1] and Ullrich Hustadt[1,⋆]

Department of Computer Science, University of Liverpool, United Kingdom
{Michel.Ludwig,U.Hustadt}@liverpool.ac.uk

Abstract. Ordered fine-grained resolution with selection is a sound and complete resolution-based calculus for monodic first-order temporal logic. The inference rules of the calculus are based on standard resolution between different types of temporal clauses on one hand and inference rules with semi-decidable applicability conditions that handle eventualities on the other hand. In this paper we illustrate how the combination of these two different types of inference rules can lead to unfair derivations in practice. We also present an inference procedure that allows us to construct fair derivations and prove its refutational completeness. We conclude with some experimental results obtained with the implementation of the new procedure in the theorem prover TSPASS.

1 Introduction

Temporal logics have long been considered as appropriate formal languages for specifying important computational properties in computer science and artificial intelligence [5]. However, while various propositional temporal logics have been used very successfully in this context, the same has not been true to the same extent for first-order temporal logic. One reason is that first-order temporal logic is not even semi-decidable, which limits its usefulness for verification purposes. A prominent fragment of first-order temporal logic which has the completeness property is the monodic fragment [8]. Besides completeness, monodic first-order temporal logic enjoys a number of other beneficial properties, e.g. the existence of non-trivial decidable subclasses, complete reasoning procedures, etc.

A first resolution-based calculus for monodic first-order temporal logic was introduced in [3]. A more machine-oriented version, the fine-grained first-order temporal resolution calculus, was described in [12]. Subsequently, a refinement of fine-grained temporal resolution, the ordered fine-grained temporal resolution with selection calculus, was presented in [10].

Essentially, the inference rules of ordered fine-grained resolution with selection can be classified into two different categories. The majority of the rules are based on standard first-order resolution between different types of temporal clauses. The remaining inference rules reflect the induction principle that holds for monodic temporal logic over a flow of time isomorphic to the natural numbers.

⋆ The work of the second author was supported by EPSRC grant EP/D060451/1.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 261–276, 2009.

The applicability of the rules in this second category is only semi-decidable. Consequently, fair derivations, i.e. derivations in which every non-redundant clause that is derivable from a given clause set is eventually derived, cannot be guaranteed in practice as the applicability check for an inference rule of the second category might not terminate.

In this paper we present an inference procedure that can construct fair derivations for reasoning in monodic first-order temporal logic and we prove its refutational completeness. The paper is organized as follows. In Section 2 we briefly recall the syntax and semantics of monodic first-order temporal logic. The ordered fine-grained resolution with selection calculus is presented in Section 3, and the new inference procedure is introduced in Section 4. Then, the refutational completeness of the new procedure is shown in Section 5. Finally, in Section 6 we briefly discuss the implementation of the new inference procedure in the theorem prover TSPASS and present some experimental results.

2 First-Order Temporal Logic

We assume the reader to be familiar with first-order logic and associated notions, including, for example, terms and substitutions.

Then, the language of First-Order (Linear Time) Temporal Logic, FOTL, is an extension of classical first-order logic by temporal operators for a discrete linear model of time (i.e. isomorphic to $\mathbb{N}$). The signature of FOTL (without equality and function symbols) is composed of a countably infinite set X of *variables* x_0, x_1, ..., a countably infinite set of *constants* c_0, c_1, ..., a non-empty set of *predicate symbols* P, P_0, ..., each with a fixed arity ≥ 0, the *propositional operators* $\top$ (**true**), $\neg$, $\vee$, the *quantifiers* $\exists x_i$ and $\forall x_i$, and the *temporal operators* $\Box$ ('always in the future'), $\Diamond$ ('eventually in the future'), $\bigcirc$ ('at the next moment'), U ('until') and W ('weak until') (see e.g. [5]). We also use $\bot$ (**false**), $\wedge$, and $\Rightarrow$ as additional operators, defined using $\top$, $\neg$, and $\vee$ in the usual way. The set of FOTL formulae is defined as follows: $\top$ is a FOTL formula; if P is an n-ary predicate symbol and $t_1, \ldots, t_n$ are variables or constants, then $P(t_1, \ldots, t_n)$ is an *atomic* FOTL formula; if φ and ψ are FOTL formulae, then so are $\neg\varphi$, $\varphi \vee \psi$, $\exists x\varphi$, $\forall x\varphi$, $\Box\varphi$, $\Diamond\varphi$, $\bigcirc\varphi$, $\varphi\,\mathsf{U}\,\psi$, and $\varphi\,\mathsf{W}\,\psi$. Free and bound variables of a formula are defined in the standard way, as well as the notions of open and closed formulae. For a given formula φ, we write $\varphi(x_1, \ldots, x_n)$ to indicate that all the free variables of φ are among $x_1, \ldots, x_n$. As usual, a *literal* is either an atomic formula or its negation, and a *proposition* is a predicate of arity 0.

Formulae of this logic are interpreted over structures $\mathfrak{M} = (D_n, I_n)_{n\in\mathbb{N}}$ that associate with each element n of $\mathbb{N}$, representing a moment in time, a first-order structure $\mathfrak{M}_n = (D_n, I_n)$ with its own non-empty domain D_n and interpretation I_n. An *assignment* $\mathfrak{a}$ is a function from the set of variables to $\bigcup_{n\in\mathbb{N}} D_n$. The application of an assignment to formulae, predicates, constants and variables is defined in the standard way, in particular, $\mathfrak{a}(c) = c$ for every constant c. The definition of the *truth relation* $\mathfrak{M}_n \models^{\mathfrak{a}} \varphi$ (only for those $\mathfrak{a}$ such that $\mathfrak{a}(x) \in D_n$ for every variable x) is given in Fig. 1.

$\mathfrak{M}_n \models^{\mathfrak{a}} \top$	
$\mathfrak{M}_n \models^{\mathfrak{a}} P(t_1, \ldots, t_n)$	iff $(I_n(\mathfrak{a}(t_1)), \ldots, I_n(\mathfrak{a}(t_n))) \in I_n(P)$
$\mathfrak{M}_n \models^{\mathfrak{a}} \neg\varphi$	iff not $\mathfrak{M}_n \models^{\mathfrak{a}} \varphi$
$\mathfrak{M}_n \models^{\mathfrak{a}} \varphi \vee \psi$	iff $\mathfrak{M}_n \models^{\mathfrak{a}} \varphi$ or $\mathfrak{M}_n \models^{\mathfrak{a}} \psi$
$\mathfrak{M}_n \models^{\mathfrak{a}} \exists x\varphi$	iff $\mathfrak{M}_n \models^{\mathfrak{b}} \varphi$ for some assignment $\mathfrak{b}$ that may differ from $\mathfrak{a}$ only in x and such that $\mathfrak{b}(x) \in D_n$
$\mathfrak{M}_n \models^{\mathfrak{a}} \forall x\varphi$	iff $\mathfrak{M}_n \models^{\mathfrak{b}} \varphi$ for every assignment $\mathfrak{b}$ that may differ from $\mathfrak{a}$ only in x and such that $\mathfrak{b}(x) \in D_n$
$\mathfrak{M}_n \models^{\mathfrak{a}} \bigcirc\varphi$	iff $\mathfrak{M}_{n+1} \models^{\mathfrak{a}} \varphi$
$\mathfrak{M}_n \models^{\mathfrak{a}} \Diamond\varphi$	iff there exists $m \geq n$ such that $\mathfrak{M}_m \models^{\mathfrak{a}} \varphi$
$\mathfrak{M}_n \models^{\mathfrak{a}} \Box\varphi$	iff for all $m \geq n$, $\mathfrak{M}_m \models^{\mathfrak{a}} \varphi$
$\mathfrak{M}_n \models^{\mathfrak{a}} \varphi \,\mathsf{U}\, \psi$	iff there exists $m \geq n$ such that $\mathfrak{M}_m \models^{\mathfrak{a}} \psi$ and $\mathfrak{M}_i \models^{\mathfrak{a}} \varphi$ for every $i, n \leq i < m$
$\mathfrak{M}_n \models^{\mathfrak{a}} \varphi \,\mathsf{W}\, \psi$	iff $\mathfrak{M}_n \models^{\mathfrak{a}} \varphi \,\mathsf{U}\, \psi$ or $\mathfrak{M}_n \models^{\mathfrak{a}} \Box\varphi$

Fig. 1. Truth-relation for first-order temporal logic

In this paper we make the *expanding domain assumption*, that is, $D_n \subseteq D_m$ if $n < m$, and we assume that the interpretation of constants is *rigid*, that is, $I_n(c) = I_m(c)$ for all $n, m \in \mathbb{N}$.

A structure $\mathfrak{M} = (D_n, I_n)_{n\in\mathbb{N}}$ is said to be a *model* for a formula φ if and only if for every assignment $\mathfrak{a}$ with $\mathfrak{a}(x) \in D_0$ for every variable x it holds that $\mathfrak{M}_0 \models^{\mathfrak{a}} \varphi$. A formula is *satisfiable* if and only there exists a model for φ. A formula φ is *valid* if and only if every temporal structure $\mathfrak{M} = (D_n, I_n)_{n\in\mathbb{N}}$ is a model for φ.

The set of valid formulae of this logic is not recursively enumerable. However, the set of valid *monodic* formulae is known to be finitely axiomatisable [15]. A formula φ of FOTL is called *monodic* if any subformula of φ of the form $\bigcirc\psi$, $\Box\psi$, $\Diamond\psi$, $\psi_1 \,\mathsf{U}\, \psi_2$, or $\psi_1 \,\mathsf{W}\, \psi_2$ contains at most one free variable. For example, the formulae $\exists x \Box \forall y P(x, y)$ and $\forall x \Box P(c, x)$ are monodic, whereas the formula $\forall x \exists y (Q(x, y) \Rightarrow \Box Q(x, y))$ is not monodic.

Every monodic temporal formula can be transformed into an equi-satisfiable normal form, called *divided separated normal form (DSNF)* [12].

Definition 1. *A* monodic temporal problem P in divided separated normal form (DSNF) *is a quadruple $\langle \mathcal{U}, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle$, where the universal part $\mathcal{U}$ and the initial part $\mathcal{I}$ are finite sets of first-order formulae; the step part $\mathcal{S}$ is a finite set of clauses of the form $p \Rightarrow \bigcirc q$, where p and q are propositions, and $P(x) \Rightarrow \bigcirc Q(x)$, where P and Q are unary predicate symbols and x is a variable; and the eventuality part $\mathcal{E}$ is a finite set of formulae of the form $\Diamond L(x)$ (a* non-ground *eventuality clause) and $\Diamond l$ (a* ground eventuality *clause), where l is an at most unary ground literal and $L(x)$ is a unary non-ground literal with the variable x as its only argument.*

We associate with each monodic temporal problem $\mathsf{P} = \langle \mathcal{U}, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle$ the monodic FOTL formula $\mathcal{I} \wedge \Box\mathcal{U} \wedge \Box\forall x \mathcal{S} \wedge \Box\forall x \mathcal{E}$. When we talk about particular properties of a temporal problem (e.g., satisfiability, validity, logical consequences, etc) we refer to properties of this associated formula.

The transformation to DSNF is based on a renaming and unwinding technique which substitutes non-atomic subformulae by atomic formulae with new predicate symbols and replaces temporal operators by their fixed point definitions as described, for example, in [6].

Theorem 1 (see [4], Theorem 3.4). *Any monodic formula in first-order temporal logic can be transformed into an equi-satisfiable monodic temporal problem in DSNF with at most a linear increase in the size of the problem.*

The main purpose of the divided separated normal form is to cleanly separate different temporal aspects of a FOTL formula from each other. For the resolution calculus in this paper we will need to go one step further by transforming the universal and initial part of a monodic temporal problem into clause normal form.

Definition 2. *Let* $\mathsf{P} = \langle \mathcal{U}, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle$ *be a monodic temporal problem. The* clausification $\mathrm{Cls}(\mathsf{P})$ of P *is a quadruple* $\langle \mathcal{U}', \mathcal{I}', \mathcal{S}', \mathcal{E}' \rangle$ *such that* $\mathcal{U}'$ *is a set of clauses, called* universal clauses*, obtained by clausification of* $\mathcal{U}$*;* $\mathcal{I}'$ *is a set of clauses, called* initial clauses*, obtained by clausification of* $\mathcal{I}$*;* $\mathcal{S}'$ *is the smallest set of step clauses such that all step clauses from* $\mathcal{S}$ *are in* $\mathcal{S}'$ *and for every non-ground step clause* $P(x) \Rightarrow \bigcirc L(x)$ *in* $\mathcal{S}$ *and every constant* c *occurring in* P*, the clause* $P(c) \Rightarrow \bigcirc L(c)$ *is in* $\mathcal{S}'$*;* $\mathcal{E}'$ *is the smallest set of eventuality clauses such that all eventuality clauses from* $\mathcal{E}$ *are in* $\mathcal{E}'$ *and for every non-ground eventuality clause* $\Diamond L(x)$ *in* $\mathcal{E}$ *and every constant* c *occurring in* P*, the eventuality clause* $\Diamond L(c)$ *is in* $\mathcal{E}'$*.*

One has to note that new constants and, especially, function symbols of an arbitrary arity can be introduced during the Skolemization process. As a consequence it is not possible in general to instantiate every variable that occurs in the original problem with all the constants and function symbols. On the other hand, the variables occurring in the step and eventuality clauses have to be instantiated with the constants that are present in the *original* problem (before Skolemization) in order to ensure the completeness of the calculus presented in Section 3.

Clause equality up to variable renaming is denoted by the symbol $=_X$; similarly, the symbols $\subseteq_X$ and $=_X$ are used to represent clause set inclusion and clause set equality up to variable renaming (of individual clauses).

A tautology is either an initial or universal clause of the form $C \vee L \vee \neg L$, or a step clause of the form $C \Rightarrow \bigcirc(D \vee L \vee \neg L)$. Step clauses of the latter form can be derived by the calculus defined in the subsequent section. For a set of clauses $\mathcal{N}$ (or a temporal problem) we denote by $\mathrm{taut}(\mathcal{N})$ the set of all the tautological clauses contained in the set $\mathcal{N}$. In what follows $\mathcal{U}$ denotes the (current) universal part of a monodic temporal problem P.

In the next section we recall the ordered fine-grained resolution with selection calculus first presented in [10]. A version of the calculus without ordering restrictions and selection functions was introduced first in [11].

3 Ordered Fine-Grained Resolution with Selection

We assume that we are given an *atom ordering* $\succ$, that is, a strict partial ordering on ground atoms which is well-founded and total, and a *selection function* S which maps any first-order clause C to a (possibly empty) subset of its negative literals and which is instance compatible:

Definition 3. *We say that a selection function S is* instance compatible *if and only if for every clause C, for every substitution σ and for every literal $l \in C\sigma$ it holds that $l \in S(C\sigma) \iff \exists l' \in S(C) \colon l'\sigma = l$.*

The atom ordering $\succ$ is extended to ground literals by $\neg A \succ A$ and $(\neg)A \succ (\neg)B$ if and only if $A \succ B$. The ordering is extended on the non-ground level as follows: for two arbitrary literals L and L', $L \succ L'$ if and only if $L\sigma \succ L'\sigma$ for every grounding substitution σ. A literal L is called (strictly) maximal w.r.t. a clause C if and only if there is no literal $L' \in C$ with $L' \succ L$ ($L' \succeq L$). A literal L is *eligible* in a clause $L \vee C$ for a substitution σ if either it is selected in $L \vee C$, or otherwise no literal is selected in C and $L\sigma$ is maximal w.r.t. $C\sigma$.

The atom ordering $\succ$ and the selection function S are used to restrict the applicability of the deduction rules of fine-grained resolution as follows. We also assume that the clauses used as premises for the different resolution-based inference rules are made variable disjoint beforehand.

(1) *First-order ordered resolution with selection between two universal clauses*

$$\frac{C_1 \vee A \quad \neg B \vee C_2}{(C_1 \vee C_2)\sigma}$$

if σ is the most general unifier of A and B, A *is eligible in* $(C_1 \vee A)$ *for* σ, *and* $\neg B$ *is eligible in* $(\neg B \vee C_2)$ *for* σ. The result is a universal clause.

(2) *First-order ordered positive factoring with selection*

$$\frac{C_1 \vee A \vee B}{(C_1 \vee A)\sigma}$$

if σ is the most general unifier of A and B, and A *is eligible in* $(C_1 \vee A \vee B)$ *for* σ. The result is again a universal clause.

(3) *First-order ordered resolution with selection between an initial and a universal clause, between two initial clauses, and ordered positive factoring with selection on an initial clause.* These are defined in analogy to the two deduction rules above with the only difference that the result is an initial clause.

(4) *Ordered fine-grained step resolution with selection.*

$$\frac{C_1 \Rightarrow \bigcirc(D_1 \vee A) \quad C_2 \Rightarrow \bigcirc(D_2 \vee \neg B)}{(C_1 \wedge C_2)\sigma \Rightarrow \bigcirc(D_1 \vee D_2)\sigma}$$

where $C_1 \Rightarrow \bigcirc(D_1 \vee A)$ and $C_2 \Rightarrow \bigcirc(D_2 \vee \neg B)$ are step clauses, σ is a most general unifier of the atoms A and B such that σ does not map variables

from C_1 or C_2 into a constant or a functional term, *A is eligible in $(D_1 \vee A)$ for σ, and $\neg B$ is eligible in $(D_2 \vee \neg B)$ for σ.*

$$\frac{C_1 \Rightarrow \bigcirc(D_1 \vee A) \quad D_2 \vee \neg B}{C_1\sigma \Rightarrow \bigcirc(D_1 \vee D_2)\sigma}$$

where $C_1 \Rightarrow \bigcirc(D_1 \vee A)$ is a step clause, $D_2 \vee \neg B$ is a universal clause, and σ is a most general unifier of the atoms A and B such that σ does not map variables from C_1 into a constant or a functional term, *A is eligible in $(D_1 \vee A)$ for σ, and $\neg B$ is eligible in $(D_2 \vee \neg B)$ for σ.* There also exists a similar rule where the positive literal A is contained in a universal clause and the negative literal $\neg B$ in a step clause.

(5) *Ordered fine-grained positive step factoring with selection.*

$$\frac{C \Rightarrow \bigcirc(D \vee A \vee B)}{C\sigma \Rightarrow \bigcirc(D \vee A)\sigma}$$

where σ is a most general unifier of the atoms A and B such that σ does not map variables from C into a constant or a functional term, and *A is eligible in $(D \vee A \vee B)$ for σ.*

(6) *Clause conversion.* A step clause of the form $C \Rightarrow \bigcirc\bot$ is rewritten to the universal clause $\neg C$.
Step clauses of the form $C \Rightarrow \bigcirc\bot$ will also be called *terminating* or *final* step clauses.

(7) *Duplicate literal elimination in left-hand sides of terminating step clauses.* A clause of the form $(C \wedge A \wedge A) \Rightarrow \bigcirc\bot$ yields the clause $(C \wedge A) \Rightarrow \bigcirc\bot$.

(8) *Eventuality resolution rule w.r.t. $\mathcal{U}$:*

$$\frac{\forall x(\mathcal{A}_1(x) \Rightarrow \bigcirc\mathcal{B}_1(x)) \quad \cdots \quad \forall x(\mathcal{A}_n(x) \Rightarrow \bigcirc\mathcal{B}_n(x)) \qquad \Diamond L(x)}{\forall x \bigwedge_{i=1}^{n} \neg\mathcal{A}_i(x)} \; (\Diamond^{\mathcal{U}}_{res}),$$

where $\forall x(\mathcal{A}_i(x) \Rightarrow \bigcirc\mathcal{B}_i(x))$ are formulae computed from the set of step clauses such that for every i, $1 \leq i \leq n$, the *loop* side conditions $\forall x(\mathcal{U} \wedge \mathcal{B}_i(x) \Rightarrow \neg L(x))$ and $\forall x(\mathcal{U} \wedge \mathcal{B}_i(x) \Rightarrow \bigvee_{j=1}^{n}(\mathcal{A}_j(x)))$ are valid.[1]

The set of full merged step clauses, satisfying the loop side conditions, is called a *loop in $\Diamond L(x)$* and the formula $\bigvee_{j=1}^{n} \mathcal{A}_j(x)$ is called a *loop formula.* More details can be found in [10].

(9) *Ground eventuality resolution rule w.r.t. $\mathcal{U}$:*

$$\frac{\mathcal{A}_1 \Rightarrow \bigcirc\mathcal{B}_1 \quad \cdots \quad \mathcal{A}_n \Rightarrow \bigcirc\mathcal{B}_n \qquad \Diamond l}{\bigwedge_{i=1}^{n} \neg\mathcal{A}_i} \; (\Diamond^{\mathcal{U}}_{res}),$$

where $\mathcal{A}_i \Rightarrow \bigcirc\mathcal{B}_i$ are ground formulae computed from the set of step clauses such that for every i, $1 \leq i \leq n$, the *loop* side conditions $\mathcal{U} \wedge \mathcal{B}_i \models \neg l$ and $\mathcal{U} \wedge \mathcal{B}_i \models \bigvee_{j=1}^{n} \mathcal{A}_j$ are valid. The notions of *ground loop* and *ground loop formula* are defined similarly to the case above.

[1] In the case $\mathcal{U} \models \forall x \neg L(x)$, the *degenerate clause*, $\top \Rightarrow \bigcirc\top$, can be considered as a premise of this rule; the conclusion of the rule is then $\neg\top$.

Function FG-BFS
Input: A set of universal clauses $\mathcal{U}$ and a set of step clauses $\mathcal{S}$, saturated by ordered fine-grained resolution with selection, and an eventuality clause $\Diamond L(x) \in \mathcal{E}$, where $L(x)$ is unary literal.
Output: A formula $H(x)$ with at most one free variable.
Method: (1) Let $H_0(x) = \mathbf{true}$; $\mathcal{M}_0 = \emptyset$; $i = 0$
(2) Let $\mathcal{N}_{i+1} = \mathcal{U} \cup \{ P(c^l) \Rightarrow \bigcirc M(c^l) \mid \text{original } P(x) \Rightarrow \bigcirc M(x) \in \mathcal{S} \} \cup \{\mathbf{true} \Rightarrow \bigcirc(\neg H_i(c^l) \vee L(c^l))\}$. Apply the rules of ordered fine-grained resolution with selection *except the clause conversion rule* to $\mathcal{N}_{i+1}$. If we obtain a contradiction, then return the loop **true** (in this case $\forall x \neg L(x)$ is implied by the universal part).
Otherwise let $\mathcal{M}_{i+1} = \{C_j \Rightarrow \bigcirc\bot\}_{j=1}^{n}$ be the set of all new *terminating* step clauses in the saturation of $\mathcal{N}_{i+1}$.
(3) If $\mathcal{M}_{i+1} = \emptyset$, return **false**; else let $H_{i+1}(x) = \bigvee_{j=1}^{n}(\tilde{\exists} C_j)\{c^l \to x\}$
(4) If $\forall x(H_i(x) \Rightarrow H_{i+1}(x))$, return $H_{i+1}(x)$.
(5) $i = i + 1$; goto 2.
Note: The constant c^l is a fresh constant used for loop search only

Fig. 2. Breadth-first Search Algorithm Using Fine-grained Step Resolution

Rules (1) to (7), also called rules of *fine-grained step resolution*, are either identical or closely related to the deduction rules of ordered first-order resolution with selection; a fact that we exploit in our implementation of the calculus.

Loop formulae, which are required for applications of the rules (8) and (9), can be computed by the fine-grained breadth-first search algorithm (FG-BFS), depicted in Fig. 2. The process of running the FG-BFS algorithm is called *loop search*.

Let *ordered fine-grained resolution with selection* be the calculus consisting of the rules (1) to (7) above, together with the ground and non-ground eventuality resolution rules described above, i.e. rules (8) and (9). We denote this calculus by $\mathfrak{I}_{FG}^{S,\succ}$.

Definition 4 (Derivation). *A (linear)* derivation Δ *(in* $\mathfrak{I}_{FG}^{S,\succ}$*) from the clausification* $\mathrm{Cls}(P)$ *of a monodic temporal problem* P *is a sequence of tuples* $\Delta = \langle \mathcal{U}_1, \mathcal{I}_1, \mathcal{S}_1, \mathcal{E}\rangle, \langle \mathcal{U}_2, \mathcal{I}_2, \mathcal{S}_2, \mathcal{E}\rangle, \ldots$ *such that each tuple at an index* $i+1$ *is obtained from the tuple at the index* i *by adding the conclusion of an application of one of the inference rules of* $\mathfrak{I}_{FG}^{S,\succ}$ *to premises from the sets* $\mathcal{U}_i$, $\mathcal{I}_i$, $\mathcal{S}_i$ *to that set, with the other sets as well as* $\mathcal{E}$ *remaining unchanged*[2]*. The derivation* Δ *will also be denoted by a sequence of clauses* $\mathcal{C}_1, \mathcal{C}_2, \ldots$ *where each clause* $\mathcal{C}_i$ *is either contained in the problem* $\langle \mathcal{U}_1, \mathcal{I}_1, \mathcal{S}_1, \mathcal{E}\rangle$ *or is newly obtained in the inference step that derived the problem* $\langle \mathcal{U}_i, \mathcal{I}_i, \mathcal{S}_i, \mathcal{E}\rangle$.

A derivation Δ *such that the empty clause is an element of a* $\mathcal{U}_i \cup \mathcal{I}_i$ *is called a (*$\mathfrak{I}_{FG}^{S,\succ}$*-)*refutation *of* $\langle \mathcal{U}_1, \mathcal{I}_1, \mathcal{S}_1, \mathcal{E}\rangle$.

[2] In an application of ground eventuality or eventuality resolution rule, the set $\mathcal{U}$ in the definition of the rule refers to $\mathcal{U}_i$.

A derivation Δ is fair *if and only if for each clause $\mathcal{C}$ which can be derived from premises in $\langle \bigcup_{i\geq 1} \mathcal{U}_i, \bigcup_{i\geq 1} \mathcal{I}_i, \bigcup_{i\geq 1} \mathcal{S}_i, \mathcal{E}\rangle$ there exists an index j such that $\mathcal{C}$ occurs in $\langle \mathcal{U}_j, \mathcal{I}_j, \mathcal{S}_j, \mathcal{E}\rangle$.*

A set of temporal clauses $\mathcal{N}$ is said to be saturated *under ordered fine-grained resolution with selection if and only if the resulting clauses from all the possible inferences under the rules of ordered fine-grained resolution with selection are already contained in the set $\mathcal{N}$.*

Ordered fine-grained resolution with selection is sound and complete monodic temporal problems over expanding domains as stated in the following theorem.

Theorem 2 (see [10], Theorem 5). *Let P be a monodic temporal problem. Let $\succ$ be an atom ordering and S an instance compatible selection function. Then P is unsatisfiable iff there exists a $\mathfrak{I}_{FG}^{S,\succ}$-refutation of* $\mathrm{Cls}(\mathsf{P})$*. Moreover, P is unsatisfiable iff any fair $\mathfrak{I}_{FG}^{S,\succ}$-derivation is a refutation of* $\mathrm{Cls}(\mathsf{P})$*.*

4 Constructing Fair Derivations

As stated in Theorem 2, any fair derivation from a clausified monodic temporal problem will eventually include a monodic temporal problem containing the empty clause. However, due to the presence of the ground and non-ground eventuality resolution rules in our calculus, constructing a fair derivation is a non-trivial problem. The validity of the side conditions of loop formulae, i.e. $\forall x(\mathcal{U} \wedge \mathcal{B}_i(x) \Rightarrow \neg L(x))$ and $\forall x(\mathcal{U} \wedge \mathcal{B}_i(x) \Rightarrow \bigvee_{j=1}^{n}(\mathcal{A}_j(x)))$ in the non-ground case, is only semi-decidable. Thus, the construction of a derivation could potentially 'get stuck' while checking these side conditions.

The use of the FG-BFS algorithm to systematically search for (ground) loop formulae does not solve the problem related to the semi-decidability of the side conditions used in the eventuality resolution rules. In step (2) of the algorithm we need to saturate the set of universal and step clauses $\mathcal{N}_{i+1}$ using the rules of fine-grained step resolution except the clause conversion rule. This saturation process may not terminate even if for a given eventuality clause $\Diamond L(x) \in \mathcal{E}$ or $\Diamond l \in \mathcal{E}$ and the set of current universal and step clauses no loop formula exists. Thus, FG-BFS also cannot guarantee fairness.

If one tries to solve the fairness problem by delaying the application of the eventuality resolution rules as long as possible, then one faces the problem that the saturation process under the rules of fine-grained step resolution may not terminate even if the original monodic temporal problem is unsatisfiable. Consequently, the strategy of executing the FG-BFS algorithm only after the original temporal problem has been saturated under fine-grained resolution may still lead to unfairness.

We can thus see that achieving fairness in derivations is not a trivial task and that it can only be accomplished if the two potentially non-terminating types of saturations, which are the regular saturation under ordered fine-grained resolution with selection on one hand, and the saturations required for loop search on the other hand, are not executed sequentially. We hence propose a

Initialization

$\langle \mathcal{U}, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle \Longrightarrow \mathcal{N} \,|\, \emptyset \,|\, \emptyset$ where

$$\mathcal{N} = \mathcal{U} \cup \mathcal{I} \cup \mathcal{S} \cup \{\, P(c^l) \Rightarrow \bigcirc M(c^l) \mid P(x) \Rightarrow \bigcirc M(x) \in \mathcal{S} \,\} \cup \{\, s_0^L \Rightarrow \bigcirc L(c^l) \mid \Diamond L(x) \in \mathcal{E} \,\}$$

Tautology Deletion

$\mathcal{N} \cup \{\mathcal{C}\} \,|\, \mathcal{P} \,|\, \mathcal{O} \Longrightarrow \mathcal{N} \,|\, \mathcal{P} \,|\, \mathcal{O}$ if $\mathcal{C}$ is a tautology

Forward Subsumption

$\mathcal{N} \cup \{\mathcal{C}\} \,|\, \mathcal{P} \,|\, \mathcal{O} \Longrightarrow \mathcal{N} \,|\, \mathcal{P} \,|\, \mathcal{O}$ if some clause in $\mathcal{P} \cup \mathcal{O}$ subsumes $\mathcal{C}$

Backward Subsumption

$\mathcal{N} \,|\, \mathcal{P} \cup \{\mathcal{C}\} \,|\, \mathcal{O} \Longrightarrow \mathcal{N} \,|\, \mathcal{P} \,|\, \mathcal{O}$

$\mathcal{N} \,|\, \mathcal{P} \,|\, \mathcal{O} \cup \{\mathcal{C}\} \Longrightarrow \mathcal{N} \,|\, \mathcal{P} \,|\, \mathcal{O}$ if some clause in $\mathcal{N}$ properly subsumes $\mathcal{C}$

Clause Processing

$\mathcal{N} \cup \{\mathcal{C}\} \,|\, \mathcal{P} \,|\, \mathcal{O} \Longrightarrow \mathcal{N} \,|\, \mathcal{P} \cup \{\mathcal{C}\} \,|\, \mathcal{O}$ if none of the previous rules applies

Loop Search Contradiction

$\emptyset \,|\, \mathcal{P} \cup \{s_i^L \Rightarrow \bigcirc\bot\} \,|\, \mathcal{O} \Longrightarrow \{\bot\} \,|\, \mathcal{P} \,|\, \mathcal{O} \cup \{s_i^L \Rightarrow \bigcirc\bot\}$ for some i, L

Next Loop Search Iteration

$\emptyset \,|\, \mathcal{P} \cup \{s_i^L \wedge C \Rightarrow \bigcirc\bot\} \,|\, \mathcal{O} \Longrightarrow$

$\{s_{i+1}^L \Rightarrow \bigcirc\neg C \vee L(c^l)\} \,|\, \mathcal{P} \,|\, \mathcal{O} \cup \{s_i^L \wedge C \Rightarrow \bigcirc\bot\}$ for some i, L and $C \neq \emptyset$

Clause Conversion

$\emptyset \,|\, \mathcal{P} \cup \{C \Rightarrow \bigcirc\bot\} \,|\, \mathcal{O} \Longrightarrow \{\Box\neg C\} \,|\, \mathcal{P} \,|\, \mathcal{O} \cup \{C \Rightarrow \bigcirc\bot\}$ where no $s_i^L \in C$

Regular Inference Computation

$\emptyset \,|\, \mathcal{P} \cup \{\mathcal{C}\} \,|\, \mathcal{O} \Longrightarrow \mathcal{N} \,|\, \mathcal{P} \,|\, \mathcal{O} \cup \{\mathcal{C}\}$ if none of the previous rule applies and where $\mathcal{N} = \mathrm{Res}_{\mathrm{T}}(\mathcal{C}, \mathcal{O})$

Loop Testing

$\emptyset \,|\, \mathcal{P} \,|\, \mathcal{O} \Longrightarrow \mathcal{N} \,|\, \mathcal{P} \,|\, \mathcal{O}$ where

$$\mathcal{N} = \{\, \Box\forall x \neg H_{i+1}^L(x) \mid \text{for all } i, L \text{ with } \models \forall x (H_i^L(x) \Leftrightarrow H_{i+1}^L(x)) \,\}$$

and $H_i^L(x) := \bigvee \{(\tilde{\exists} C_j)\{c^l \to x\} \mid s_i^L \wedge C_j \Rightarrow \bigcirc\bot \in \mathcal{P} \cup \mathcal{O}\}$ for all i, L

Fig. 3. Fair inference procedure

way of combining these two types of saturations into one 'global' saturation process.

The first step towards a procedure which guarantees the construction of a fair derivation is based on an idea introduced in [7] for improving the efficiency of loop search in propositional linear-time temporal logic. It suggests a minor modification of the loop search algorithm. In step (2) of the algorithm we now add the clauses which result from clausification of the formula $\mathcal{C}_{i+1}^L = s_{i+1}^L \Rightarrow \bigcirc(\neg H_i(c^l) \vee L(c^l))$ to $\mathcal{N}_{i+1}$, where s_{i+1}^L is a proposition uniquely associated with index $i+1$ and the eventuality clause $\Diamond L(x)$ for which we search for a loop. The proposition s_{i+1}^L acts as a marker for these clauses which are generated purely as a means to conduct the search. As there are only negative occurrences of s_{i+1}^L, the application of inference rules to these clauses will 'propagate' the proposition to all clauses we derive from $\mathcal{C}_{i+1}^L$. This also means that $\mathcal{M}_{i+1}$ can

now be defined as the set of all clauses of the form $s^L_{i+1} \wedge C_j \Rightarrow \bigcirc\mathbf{false}$. While this makes the construction of $\mathcal{M}_{i+1}$ operationally easier compared to FG-BFS, it does not fundamentally change the algorithm. However, the FG-BFS algorithm allows us to take advantage of the following observations. Within iterations of the steps (2)–(5) of the algorithm, the clauses in the various sets $\mathcal{N}_{i+1}$ are now separated by the 'marker' s^L_{i+1}. Thus, instead of using different sets $\mathcal{N}_{i+1}$ we can use a single set $\mathcal{T}$ which is simply extended in each iteration of the steps (2)–(5). Furthermore, we can keep the set $\mathcal{T}$ between different calls of the FG-BFS procedure and also between repeated calls of FG-BFS for the same eventuality clause $\Diamond L(x) \in \mathcal{E}$. Finally, if we restrict the clause conversion rule so that it cannot be applied to any clause containing a 'marker' s^L_i, then there is no reason to separate the clauses in $\mathcal{T}$ from those in the current monodic temporal problem in clausified form stored by the prover. Fig. 3 depicts the inference procedure based on these considerations in the presentation style of [1]. The inference procedure operates on states $(\mathcal{N} \,|\, \mathcal{P} \,|\, \mathcal{O})$ that are constructed from an initial temporal problem $\mathsf{P} = \langle \mathcal{U}, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle$.

A state $(\mathcal{N} \,|\, \mathcal{P} \,|\, \mathcal{O})$ consists of three sets of clauses $\mathcal{N}$ (the set of *new* clauses), $\mathcal{P}$ (the set of clauses that still have to be *processed*) and $\mathcal{O}$ (the set of *old* clauses). The set $\mathcal{N}$ collects the newly-derived clauses and the set $\mathcal{O}$ contains all the clauses that have already been used as premises in inference steps (or can never be used as premises). Finally, the set $\mathcal{P}$ contains all the clauses that still need to be considered as premises. In the initial state $(\mathcal{N}_0 \,|\, \emptyset \,|\, \emptyset)$ constructed by the 'initialization' rule, the sets $\mathcal{P}$, $\mathcal{O}$ are empty and the set $\mathcal{N}_0$ contains all the clauses contained in a temporal problem $\mathsf{P} = \langle \mathcal{U}, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle$. Additionally, as motivated above, all the clauses required for loop search are added to $\mathcal{N}_0$. Subsequent states are obtained by applying one of the other inference rules depicted in Fig. 3 on a given state.

The rules 'tautology deletion', 'forward subsumption' and 'backward subsumption' perform reduction on the clause sets. 'Tautology deletion' removes tautological clauses from set of newly-derived clauses $\mathcal{N}$. 'Forward subsumption' and 'backward subsumption' eliminate clauses that have been subsumed by other clauses. Finally, the 'clause processing' rule is responsible for moving a clause that has survived the previous reduction steps to the set $\mathcal{P}$. Once no further reductions are possible, additional clauses can be derived by the following inference rules.

For the 'loop search contradiction' rule note that the presence of $s^L_i \Rightarrow \bigcirc\bot$ in $\mathcal{P}$ indicates that we can apply the non-ground eventuality resolution rule, resulting in the conclusion $\Box\forall x \neg\top$, which is contradictory. The empty clause is then added as set $\mathcal{N}$ and the clause $s^L_i \Rightarrow \bigcirc\bot$ is moved to the set of old clauses $\mathcal{O}$. If the set $\mathcal{P}$ contains a clause $s^L_i \wedge C \Rightarrow \bigcirc\bot$ for some i, L and $C \neq \emptyset$, then such a clause would be part of the set $\mathcal{N}_i$ in the FG-BFS procedure, which is used to define the formula $H_{i+1}(x)$, which in turn is used to define the clauses in $\mathcal{M}_{i+1}$. Here, we directly define the one clause of $\mathcal{M}_{i+1}$ which derives from $s^L_i \wedge C \Rightarrow \bigcirc\bot$ and add it as newly-derived clause set. Finally, if a clause $C \Rightarrow \bigcirc\bot$ (without a marker s^L_i) is contained in the set $\mathcal{P}$, such a clause is a

suitable premise for the application of the clause conversion rule and we add the universal clause(s) $\Box\neg C$ as newly-derived clause set. The clause $C \Rightarrow \bigcirc\bot$ is moved to the set $\mathcal{O}$.

In the case where the set $\mathcal{P}$ contains a clause $\mathcal{C}$ that is not handled by one of the previous rules, we compute the set $\mathrm{Res}_{\mathrm{T}}(\mathcal{C}, \mathcal{O})$, which consists of all the conclusions derivable from the clause $\mathcal{C}$ with itself and with the clauses in $\mathcal{O}$ by the rules (1) to (5) and (7). The computed clauses are then added to the next state as set $\mathcal{N}$ and the clause $\mathcal{C}$ is moved to the set of old clauses $\mathcal{O}$. The remaining rule 'loop testing' is responsible for checking the loop side condition. First, the formulae H_i^L are computed for all eventuality clauses $\Diamond L(x) \in \mathcal{E}$ and all indices i used to create some marker s_i^L in the set $\mathcal{P} \cup \mathcal{O}$. We then check whether the loop condition $\forall x(H_i^L(x) \Leftrightarrow H_{i+1}^L(x))$ holds for every i and every L. If so, an application of the non-ground eventuality resolution rule is possible and we compute the conclusion of the application and add it to the set $\mathcal{N}$. This concludes the description of the fair inference procedure.

In order for the inference procedure shown in Fig. 3 to remain complete two inference rules have to be added to ordered fine-grained resolution with selection: the rule of *arbitrary factoring in left-hand sides of terminating step clauses* and the rule of *arbitrary factoring in (at most) monadic negative universal clauses*. The calculus ordered fine-grained resolution with selection extended by the two rules introduced above will be called *subsumption compatible* ordered fine-grained resolution with selection and will be denoted by $\mathfrak{I}^{S,\succ}_{FG,Sub}$.

Furthermore, the completeness proof of the inference procedure shown in Fig. 3 also depends on special properties of the selection function. We require the selection function to be *subsumption compatible*, as defined below.

Definition 5. *We say that a selection function S is* subsumption compatible *if and only if for every substitution σ and for every two clauses $\mathcal{C}$ and $\mathcal{D}$ with $\mathcal{C}\sigma \subseteq \mathcal{D}$ it holds for every literal $L \in D$ that $L \in S(\mathcal{D}) \iff L\sigma \in S(\mathcal{C})$.*

There are three important observations to be made about the 'loop testing' rule. First, we can observe that $H_i^L(x)$ and $H_{i+1}^L(x)$ are monadic first-order formulae. Thus, the validity of the loop condition is a decidable problem. Second, in FG-BFS, in order to establish whether $\forall x(H_i^L(x) \Leftrightarrow H_{i+1}^L(x))$ is valid we only need to test whether $\forall x(H_i^L(x) \Rightarrow H_{i+1}^L(x))$ holds as the implication $\forall x(H_{i+1}^L(x) \Rightarrow H_i^L(x))$ is always valid by the construction of $H_i^L(x)$ and $H_{i+1}^L(x)$ in these procedures. However, in the context of the inference procedure in Fig. 3 this is no longer the case and we need to test both implications. Finally, whenever the loop condition holds, we have indeed found a loop formula, although it may not be equivalent to a formula returned by FG-BFS. We will see that eventually an equivalent formula will be computed by the procedure in Fig. 3.

We conclude this section by stating the refutational completeness of subsumption compatible ordered fine-grained resolution with selection.

Theorem 3. *Let P be a monodic temporal problem. Let $\succ$ be an atom ordering and S a subsumption compatible selection function. Then P is unsatisfiable iff*

there exists a $\mathfrak{I}^{S,\succ}_{FG,Sub}$*-refutation of* $\mathrm{Cls}(P)$*. Moreover,* P *is unsatisfiable iff any fair* $\mathfrak{I}^{S,\succ}_{FG,Sub}$*-derivation is a refutation of* $\mathrm{Cls}(P)$*.*

5 Refutational Completeness

The proof of refutational completeness for the inference procedure shown in Fig. 3 is based on showing that for every non-tautological clause $\mathcal{D}$ which can be derived by subsumption-complete ordered fine-grained with selection from a temporal problem P there exists a clause $\mathcal{C}$ that is derived by the fair inference procedure in Fig. 3 started on the same temporal problem P such that $\mathcal{C}$ subsumes $\mathcal{D}$, i.e. $\mathcal{C} \leq_s \mathcal{D}$.

For terminating step clauses or universal clauses $\mathcal{C}(\Rightarrow \bigcirc\bot)$ and $\mathcal{D}(\Rightarrow \bigcirc\bot)$ we define $\mathcal{C} \leq_s \mathcal{D}$ iff there exists a substitution σ with $\mathcal{C}\sigma \subseteq \mathcal{D}$. Note that it is possible to extend the subsumption relation on the other types of temporal clauses and retain the compatibility with the inference rules of our calculus.

For the completeness proof we assume that the fair inference procedure actually does not terminate whenever the empty clause has been derived but continues to derive clauses instead.

Definition 6 (Derivation). *A derivation* Δ *produced by the inference procedure shown in Fig. 3 from a temporal problem* $P = \langle \mathcal{U}, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle$ *is a sequence of states* $\langle \mathcal{U}, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle \Longrightarrow \mathcal{N}_0 \,|\, \mathcal{P}_0 \,|\, \mathcal{O}_0 \Longrightarrow \mathcal{N}_1 \,|\, \mathcal{P}_1 \,|\, \mathcal{O}_1 \Longrightarrow \mathcal{N}_2 \,|\, \mathcal{P}_2 \,|\, \mathcal{O}_2 \Longrightarrow \ldots$ *where each state* $(\mathcal{N}_i \,|\, \mathcal{P}_i \,|\, \mathcal{O}_i)$*,* $i \geq 0$*, results from an application of an inference rule shown in Fig. 3 on the state* $(\mathcal{N}_{i-1} \,|\, \mathcal{P}_{i-1} \,|\, \mathcal{O}_{i-1})$ *iff* $i > 0$ *or on* $\langle \mathcal{U}, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle$*.*

If $\mathcal{N}_i = \emptyset$ *and* $\mathcal{P}_i = \emptyset$ *for an index* $i \in \mathbb{N}$*, we define* $(\mathcal{N}_i \,|\, \mathcal{P}_i \,|\, \mathcal{O}_i) = (\mathcal{N}_j \,|\, \mathcal{P}_j \,|\, \mathcal{O}_j)$ *for every* $j \geq i$*.*

A derivation Δ *is said to be* fair *if and only if* $\bigcup_{i=0}^{\infty} \bigcap_{j \geq i}^{\infty} \mathcal{P}_j = \emptyset$ *and, whenever possible, every application of the 'regular inference computation' rule is eventually followed by an application of the 'loop testing' rule.*

Throughout this section we assume that $\mathsf{P} = \langle \mathcal{U}, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle$ is a clausified monodic temporal problem and that $\Delta = (\mathcal{N}_i \,|\, \mathcal{P}_i \,|\, \mathcal{O}_i)_{i \in \mathbb{N}}$ is a fair derivation produced by the inference procedure shown in Fig. 3 from the temporal problem P.

The first lemma shows that non-tautological clauses which do not originate from loop search are subsumed by clauses derived by the fair inference procedure.

Lemma 1. *Let* $\Delta = (\mathcal{N}_i \,|\, \mathcal{P}_i \,|\, \mathcal{O}_i)_{i \in \mathbb{N}}$ *be a fair derivation produced by the inference procedure shown in Fig. 3 from the temporal problem* P*. Let* $\Delta' = \mathcal{D}_1, \ldots, \mathcal{D}_n$ *be a derivation produced by subsumption complete ordered fine-grained resolution with selection from the temporal problem* P *without applying the eventuality resolution rules and such that for every clause* $\mathcal{D} \in \mathcal{U} \cup \mathcal{I} \cup \mathcal{S}$ *with* $\mathcal{D} \notin \mathrm{taut}(P)$ *there exists a clause* $\mathcal{C} \in \bigcup_{i=0}^{\infty} \mathcal{O}_i$ *with* $\mathcal{C} \leq_s \mathcal{D}$*.*

Then it holds for every i *with* $1 \leq i \leq n$ *that*

$$\mathcal{D}_i \notin \mathrm{taut}(P) \Rightarrow \exists\, j \in \mathbb{N}\ \exists \mathcal{C}_i \in \mathcal{O}_j \colon \mathcal{C}_i \leq_s \mathcal{D}_i$$

Proof. Similar to the first-order case, by induction on i and by using the fairness of the derivation Δ. □

The remainder of the proofs are now concerned with showing that non-tautological clauses stemming from loop search are also subsumed by clauses derived by the fair inference procedure. The next two lemmata are crucial for the completeness proof.

For two clauses $\mathcal{C}$ and $\mathcal{D}$ we write $\mathcal{C} \vdash_f^* \mathcal{D}$ to indicate that either $\mathcal{C} = \mathcal{D}$ or that $\mathcal{D}$ results from the clause $\mathcal{C}$ through repetitive applications of the arbitrary factoring rule. By $\mathrm{LT}(\mathcal{S})$ we denote the constant flooding of the step clauses contained in the set $\mathcal{S}$ with the loop search constant c^l, i.e. $\mathrm{LT}(\mathcal{S}) = \mathcal{S} \cup \{\, P(c^l) \Rightarrow \bigcirc Q(c^l) \mid P(x) \Rightarrow \bigcirc Q(x) \in \mathcal{S} \,\}$.

Lemma 2. *Let $C \Rightarrow \bigcirc\bot$ and $D_1 \Rightarrow \bigcirc\bot, \ldots, D_n \Rightarrow \bigcirc\bot$ be terminating step clauses derived by subsumption complete ordered fine-grained resolution with selection from sets $\boldsymbol{P} \cup \mathrm{LT}(\mathcal{S}) \cup \mathrm{Cls}(\boldsymbol{true} \Rightarrow \bigcirc(\neg H_i(c^l) \vee L(c^l)))$ and $\boldsymbol{P} \cup \mathrm{LT}(\mathcal{S}) \cup \mathrm{Cls}(\boldsymbol{true} \Rightarrow \bigcirc(\neg H_j(c^l) \vee L(c^l)))$, respectively, as in the FG-BFS algorithm for (arbitrary) iterations i and j.*

In addition let the formula $\forall x((\tilde{\exists} C)\{c^l \mapsto x\} \Rightarrow \bigvee_{j=1}^n (\tilde{\exists} D_j)\{c^l \mapsto x\})$ be valid. Then there exists an index k with $1 \leq k \leq n$ and a clause D such that $D_k \vdash_f^* D \wedge D \leq_s C$.

Proof. Based on analysing the refutation of the formula $\neg\forall x((\tilde{\exists} C)\{c^l \mapsto x\} \Rightarrow \bigvee_{j=1}^n (\tilde{\exists} D_j)\{c^l \mapsto x\})$ by regular first-order resolution. □

Lemma 3. *Let $\mathcal{M}$ and $\mathcal{M}'$ be sets of terminating step clauses derived by subsumption complete ordered fine-grained resolution with selection from from sets $\boldsymbol{P} \cup \mathrm{LT}(\mathcal{S}) \cup \mathrm{Cls}(\boldsymbol{true} \Rightarrow \bigcirc(\neg H_i(c^l) \vee L(c^l)))$ and $\boldsymbol{P} \cup \mathrm{LT}(\mathcal{S}) \cup \mathrm{Cls}(\boldsymbol{true} \Rightarrow \bigcirc(\neg H_j(c^l) \vee L(c^l)))$, respectively, as in the FG-BFS algorithm for (arbitrary) iterations i and j. We also assume that the sets $\mathcal{M}$ and $\mathcal{M}'$ are closed under the application of the (unordered) factoring rule. Finally, let $\mathcal{N} = \{C_1 \Rightarrow \bigcirc\bot, \ldots, C_m \Rightarrow \bigcirc\bot\} \subseteq \mathcal{M}$ and $\mathcal{N}' = \{D_1 \Rightarrow \bigcirc\bot, \ldots, D_n \Rightarrow \bigcirc\bot\} \subseteq \mathcal{M}'$ be the sets of all the minimal step clauses with respect to the relation $\leq_s$ contained in the sets $\mathcal{M}$ and $\mathcal{M}'$, respectively.*

Then the following statements are equivalent:

(i) $\mathcal{N} =_X \mathcal{N}'$

(ii) the formula $\forall x(\bigvee_{i=1}^m (\tilde{\exists} C_i)\{c^l \mapsto x\} \Leftrightarrow \bigvee_{j=1}^n (\tilde{\exists} D_j)\{c^l \mapsto x\})$ *is valid*

(iii) $\forall i, 1 \leq i \leq m\ \exists j, 1 \leq j \leq n\colon D_j \leq_s C_i$ *and* $\forall j, 1 \leq i \leq n\ \exists i, 1 \leq j \leq m\colon C_i \leq_s D_j$

Proof. The implication $(i) \Rightarrow (ii)$ is obvious. By Lemma 2 and by closedness under factoring inferences the implication $(ii) \Rightarrow (iii)$ holds. For the remaining implication $(iii) \Rightarrow (i)$ let $C \Rightarrow \bigcirc\bot \in \mathcal{N}$. It then follows from the assumptions that there exists a step clause $D \Rightarrow \bigcirc\bot \in \mathcal{N}'$ such that $D \leq_s C$. We obtain again from the assumptions that there exists a step clause $C' \Rightarrow \bigcirc \in \mathcal{N}$ with $C' \leq_s D$. Thus, we can conclude that there exists a variable renaming σ with $C\sigma = D$ and $C\sigma \Rightarrow \bigcirc\bot \in \mathcal{N}'$ as otherwise there would exist a step clause

$C' \Rightarrow \bigcirc\bot \in \mathcal{M}$ with $C' <_s C$, which would contradict the minimality of the step clause $C \Rightarrow \bigcirc\bot$.

The inclusion $\mathcal{N}' \subseteq_X \mathcal{N}$ can be shown analogously. □

Lemma 3 shows that the equivalence test between the monadic formulae $H_i^L(x)$ and $H_{i+1}^L(x)$ in the 'loop testing' rule can replaced by a test for mutual subsumption of terminating step clauses. Lemma 3 can also be used to prove further properties of the loop search algorithm. For example, it can be shown that the sets of minimal clauses with respect to the subsumption relation remain unchanged in subsequent iterations (up to variable renaming) once the loop search condition has been fulfilled and the algorithm has been kept iterating.

It is also possible to show that for any derivation of the fair inference procedure there exists an iteration index which contains all the universal clauses that can ever be derived by either loop search or the clause conversion rule. The rationale behind this observation is the finite number of step clauses that are contained in the original temporal problem. Consequently, for any execution of the loop search algorithm (for an arbitrary eventuality) and for any iteration it can be established that there exists an iteration index for a run of the fair inference procedure in which the same terminating step clauses (up to variable renaming) are derived unless they were subsumed by other clauses. Moreover, due to Lemma 3 the loop condition will eventually be achieved in the fair inference procedure as it is equivalent to the equality of the sets containing the minimal clauses with respect to the subsumption relation $\leq_s$. Note that the terminating loop search clauses might be derived in a different order than with the loop search algorithm.

Based on these observations we can now state the refutational completeness of the inference procedure shown in Fig. 3.

Theorem 4. *Let $\mathsf{P} = \langle \mathcal{U}_0, \mathcal{I}, \mathcal{S}, \mathcal{E} \rangle$ be a clausified monodic temporal problem. Let $\succ$ be an atom ordering and S a subsumption compatible selection function. Additionally, let $\Delta = (\mathcal{N}_i \,|\, \mathcal{P}_i \,|\, \mathcal{O}_i)_{i \in \mathbb{N}}$ be a fair derivation produced by the inference procedure shown in Fig. 3 from the temporal problem P using the atom ordering $\succ$ and selection function S. Then it holds that:*

$$\mathsf{P} \text{ is unsatisfiable iff } \bot \in \bigcup_{j=0}^{\infty} \mathcal{O}_j$$

6 Implementation

The inference procedure depicted in Fig. 3 has been implemented in the automated theorem prover TSPASS[3], which is based on the first-order resolution prover SPASS 3.0 [14]. It has to be noted that TSPASS currently does not allow the use of selection functions. A full description of the implementation can be found in [13].

We have analyzed the practical performance of TSPASS 0.91-0.15 on different temporal problems and found it to be competitive with TeMP [9]. TeMP is an

[3] `http://www.csc.liv.ac.uk/~{}{}michel/software/tspass/`

earlier implementation of ordered fine-grained resolution with selection based on a sequential architecture in which loop searches are performed only after the considered clause sets have been saturated under fine-grained step resolution. TeMP uses the first-order theorem prover Vampire as inference kernel for the saturations under fine-grained step resolution.

The experimental setting was as follows. The experiments were run on a PC equipped with an Intel Core 2 E6400 CPU and 3 GB of main memory. An execution timeout of 12 minutes was imposed on each problem. For TeMP the input problems were first transformed into its clausal input form and then TeMP was started on this clausal input without any additional settings. TSPASS was instructed to perform subsumption-based loop search testing.

Table 1 shows the satisfiability status, the number of clauses generated and the median CPU time in seconds over three different runs of TeMP and TSPASS for the specification of five representative examples (out of 39) on simple foraging robots and some associated properties. The specification results from a novel application of monodic first-order temporal logic in the verification of the behaviour of robot swarms. The use of FOTL allows us to verify properties for a potentially infinite number of robots contained in the swarms. Further details can be found in [2]. Problem 0 represents the specification of the robot transition system, and the remaining problems verify some properties of the transition system. Each of these problems contains at least seven eventualities. TeMP and TSPASS both terminate on the satisfiable problem 0, but TeMP cannot solve the unsatisfiable problem 2 within the given time limit. Problem 2 seems to exhibit the fairness problems of TeMP as it remains 'stuck' whilst saturating a temporal clause set under ordered fine-grained resolution with selection. Due to the way TeMP computes the number of generated clauses, i.e. the clause statistics are only updated whenever Vampire returns from a saturation process, we cannot easily give the number of clauses generated on problem 2 as the total number of clauses is not available in TeMP itself.

Moreover, on average TeMP derives more clauses and requires more execution time than TSPASS, except for problem 12. We attribute this observation to the subsumption-based loop search test in TSPASS and to the fact that inferences in TSPASS which have been computed once for a loop search instance do not have to be computed again for further loop search saturations. Further details and more examples can be found in [13].

Table 1. Results obtained for the robot specification examples

Problem	Clauses Generated		Time		Result
	TeMP	TSPASS	TeMP	TSPASS	
0	19611	5707	0.482s	0.383s	Satisfiable
1	21812	833	0.523s	0.080s	Unsatisfiable
2	-	4834	-	0.371s	Unsatisfiable
12	689	793	0.035s	0.078s	Unsatisfiable
18	32395	5262	0.967s	0.387s	Unsatisfiable

7 Conclusion

We have presented a fair inference procedure for reasoning in monodic first-order temporal logic based on the ordered fine-grained resolution with selection calculus. The calculus originally contains rules with semi-decidable applicability conditions, which are realised by a loop-search algorithm computing saturations under ordered fine-grained step resolution with selection. The design of the new inference procedure is based on integrating the saturation steps related to loop search, which may not terminate, into the main saturation process. Consequently, fair derivations can be obtained as the different saturations required for fine-grained step resolution and loop search are no longer performed sequentially.

References

1. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 1, pp. 19–99. Elsevier, Amsterdam (2001)
2. Behdenna, A., Dixon, C., Fisher, M.: Deductive verification of simple foraging robotic behaviours (under review)
3. Degtyarev, A., Fisher, M., Konev, B.: Monodic temporal resolution. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 397–411. Springer, Heidelberg (2003)
4. Degtyarev, A., Fisher, M., Konev, B.: Monodic temporal resolution. ACM Transactions On Computational Logic 7(1), 108–150 (2006)
5. Emerson, E.A.: Temporal and modal logic. In: Handbook of Theoretical Computer Science. Formal Models and Semantics, vol. B, pp. 995–1072 (1990)
6. Fisher, M., Dixon, C., Peim, M.: Clausal temporal resolution. ACM Transactions on Computational Logic 2(1), 12–56 (2001)
7. Gago, M.C.F., Fisher, M., Dixon, C.: Algorithms for guiding clausal temporal resolution. In: Jarke, M., Koehler, J., Lakemeyer, G. (eds.) KI 2002. LNCS, vol. 2479, pp. 235–252. Springer, Heidelberg (2002)
8. Hodkinson, I., Wolter, F., Zakharyaschev, M.: Decidable fragments of first-order temporal logics. Annals of Pure and Applied Logic 106, 85–134 (2000)
9. Hustadt, U., Konev, B., Riazanov, A., Voronkov, A.: TeMP: A temporal monodic prover. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 326–330. Springer, Heidelberg (2004)
10. Hustadt, U., Konev, B., Schmidt, R.A.: Deciding monodic fragments by temporal resolution. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 204–218. Springer, Heidelberg (2005)
11. Konev, B., Degtyarev, A., Dixon, C., Fisher, M., Hustadt, U.: Towards the implementation of first-order temporal resolution: the expanding domain case. In: Proc. TIME-ICTL 2003, pp. 72–82. IEEE Computer Society, Los Alamitos (2003)
12. Konev, B., Degtyarev, A., Dixon, C., Fisher, M., Hustadt, U.: Mechanising first-order temporal resolution. Information and Computation 199(1-2), 55–86 (2005)
13. Ludwig, M., Hustadt, U.: Implementing a fair monodic temporal logic prover. AI Communications (to appear)
14. Weidenbach, C., Schmidt, R., Hillenbrand, T., Rusev, R., Topic, D.: System description: SPASS version 3.0. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 514–520. Springer, Heidelberg (2007)
15. Wolter, F., Zakharyaschev, M.: Axiomatizing the monodic fragment of first-order temporal logic. Annals of Pure and Applied logic 118, 133–145 (2002)

A Term Rewriting Approach to the Automated Termination Analysis of Imperative Programs*

Stephan Falke and Deepak Kapur

CS Department, University of New Mexico, Albuquerque, NM, USA
{spf,kapur}@cs.unm.edu

Abstract. An approach based on term rewriting techniques for the automated termination analysis of imperative programs operating on integers is presented. An imperative program is transformed into rewrite rules with constraints from quantifier-free Presburger arithmetic. Any computation in the imperative program corresponds to a rewrite sequence, and termination of the rewrite system thus implies termination of the imperative program. Termination of the rewrite system is analyzed using a decision procedure for Presburger arithmetic that identifies possible chains of rewrite rules, and automatically generated polynomial interpretations are used to show finiteness of such chains. An implementation of the approach has been evaluated on a large collection of imperative programs, thus demonstrating its effectiveness and practicality.

1 Introduction

Methods for automatically proving termination of imperative programs operating on integers have received increased attention recently. The most commonly used automatic method for this is based on linear ranking functions which linearly combine the values of the program variables in a given state [2,5,6,27,28]. It was shown in [29] that termination of a simple class of linear programs consisting of a single loop is decidable. More recently, the combination of abstraction refinement and linear ranking functions has been considered [9,10,4]. The tool `Terminator` [11], developed at Microsoft Research and based on this idea, has reportedly been used for showing termination of device drivers.

On the other hand, termination analysis for term rewrite systems (TRSs) has been investigated extensively [30]. In this paper, techniques based on ideas from the term rewriting literature are used in order to show termination of imperative programs operating on integers. This is done by translating imperative programs into constrained term rewrite systems based on Presburger arithmetic ($\mathcal{PA}$-based TRS), where the constraints are relations on program variables expressed as quantifier-free formulas from Presburger arithmetic. This way, computations of the imperative program are simulated by rewrite sequences, and termination of the $\mathcal{PA}$-based TRS implies termination of the imperative program.

* Partially supported by NSF grants CCF-0541315 and CNS-0831462.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 277–293, 2009.

It is then shown that a $\mathcal{PA}$-based TRS is terminating if and only if it does not admit infinite chains built from the rewrite rules. In order to show absence of infinite chains, termination processors are introduced. Here, a termination processor transform a "complex" termination problem into a set of "simpler" termination problems. This general approach for proving termination stems from the dependency pair framework for ordinary TRSs [19]. The present paper introduces several such termination processors for $\mathcal{PA}$-based TRSs that are based on ideas from the term rewriting literature [24,1,14]. The first termination processor uses a decision procedure for Presburger arithmetic to identify possible chains. For this, it is determined which rules from a $\mathcal{PA}$-based TRS may follow each other in a chain. Once possible chains are identified, well-founded relations based on polynomial interpretations are used to show that these chains are finite. A further termination processor can be used to combine two $\mathcal{PA}$-based rewrite rules that may follow each other in a chain into a single $\mathcal{PA}$-based rewrite rule. In particular, the constraints of these two rewrite rules are propagated.

The approach presented in this paper is very intuitive. Indeed, it has been successfully introduced in graduate level classes on the theory of programming languages and static program analysis. The students in these classes do typically not have previous knowledge of term rewriting or termination methods based on linear ranking functions [2,5,6,27,28].

The approach has been implemented in the termination tool `pasta`. An empirical evaluation on a collection of examples taken from recent papers on the termination analysis of imperative programs [2,3,4,5,6,9,10,27,28] shows the practicality of the method. The only non-trivial part of an implementation is the automatic generation of well-founded relations using polynomial interpretations [24] since the constraints of the $\mathcal{PA}$-based rewrite rules need to be taken into consideration. Current methods developed in the term rewriting literature [8,16] do not support constraints, thus requiring the development of a new method.[1]

The paper is organized as follows. Sect. 2 introduces $\mathcal{PA}$-based TRSs. The translation of imperative programs into $\mathcal{PA}$-based TRSs is discussed in Sect. 3. Next, Sect. 4 introduces chains and shows that a $\mathcal{PA}$-based TRS is terminating iff it does not admit infinite chains. Furthermore, a framework for showing termination by transforming a $\mathcal{PA}$-based TRS into a set of simpler $\mathcal{PA}$-based TRSs using termination processors is introduced. Sect. 5 discusses a termination processor that uses a decision procedure for Presburger arithmetic to identify possible chains. Well-founded relations based on polynomial interpretations are introduced in Sect. 6 and a termination processor that combines $\mathcal{PA}$-based rewrite rules and propagates their constraints is given in Sect. 7. Finally, Sect. 8 outlines the implementation of the termination tool `pasta`. This includes a method for the automatic generation of polynomial interpretations as discussed above. Sect. 9 concludes and presents an empirical evaluation of `pasta`. The proofs omitted from this version and the examples used for the empirical evaluation of `pasta` can be found in [15].

[1] The method presented in [17] is similar to the method of this paper. Indeed, the method of [17] was partially derived from the method presented here.

2 $\mathcal{PA}$-Based TRSs

In order to model integers, the function symbols from $\mathcal{F}_{\mathcal{PA}} = \{0, 1, +, -\}$ with types $0, 1 : \texttt{int}$, $+ : \texttt{int} \times \texttt{int} \to \texttt{int}$, and $- : \texttt{int} \to \texttt{int}$ are used. Terms built from these function symbols and a disjoint set $\mathcal{V}$ of variables are called $\mathcal{PA}$-*terms*. This paper uses a simplified, more natural notation for $\mathcal{PA}$-terms, i.e., the $\mathcal{PA}$-term $(x + (-(y + y))) + (1 + (1 + 1))$ will be written as $x - 2y + 3$.

Then, $\mathcal{F}_{\mathcal{PA}}$ is extended by finitely many function symbols f with types $\texttt{int} \times \ldots \times \texttt{int} \to \texttt{univ}$, where $\texttt{univ}$ is a type distinct from $\texttt{int}$. The set containing these function symbols is denoted by $\mathcal{F}$, and $\mathcal{T}(\mathcal{F}, \mathcal{F}_{\mathcal{PA}}, \mathcal{V})$ denotes the set of terms of the form $f(s_1, \ldots, s_n)$ where $f \in \mathcal{F}$ and $s_1, \ldots, s_n$ are $\mathcal{PA}$-terms. Notice that nesting of function symbols from $\mathcal{F}$ is *not permitted*, thus resulting in a simple term structure. This simple structure is nonetheless sufficient for modeling imperative programs. A *substitution* is a mapping from variables to $\mathcal{PA}$-terms.

Next, recall the syntax and semantics of $\mathcal{PA}$-constraints, i.e., quantifier-free formulas from Presburger arithmetic. An *atomic $\mathcal{PA}$-constraint* has the form $s \simeq t$, $s \geq t$, or $s > t$ for $\mathcal{PA}$-terms s, t. The set of $\mathcal{PA}$-*constraints* is the closure of the set of atomic $\mathcal{PA}$-constraints under $\top$ (truth), $\bot$ (falsity), $\neg$ (negation), and $\wedge$ (conjunction). The Boolean connectives $\vee$, $\Rightarrow$, and $\Leftrightarrow$ are defined as usual. $\mathcal{PA}$-constraints of the form $s < t$ and $s \leq t$ are used to stand for $t > s$ and $t \geq s$, respectively. Also, $s \not\simeq t$ stands for $\neg(s \simeq t)$, and similarly for the other predicates. $\mathcal{PA}$-constraints have the expected semantics, where a $\mathcal{PA}$-constraint is $\mathcal{PA}$-*valid* iff it is true for all ground instantiations of its variables and $\mathcal{PA}$-*satisfiable* iff it is true for some ground instantiation of its variables. Notice that $\mathcal{PA}$-validity and $\mathcal{PA}$-satisfiability are decidable. For $\mathcal{PA}$-terms s, t, writing $s \simeq_{\mathcal{PA}} t$ is a shorthand for "$s \simeq t$ is $\mathcal{PA}$-valid". Similarly, for terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{F}_{\mathcal{PA}}, \mathcal{V})$, $s \simeq_{\mathcal{PA}} t$ iff $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$ such that $s_i \simeq_{\mathcal{PA}} t_i$ for all $1 \leq i \leq n$.

Definition 1 ($\mathcal{PA}$-Based Rewrite Rules). *A $\mathcal{PA}$-based rewrite rule has the form $l \to r[\![C]\!]$ where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{F}_{\mathcal{PA}}, \mathcal{V})$ and C is a $\mathcal{PA}$-constraint. Thus, l and r have the form $f(s_1, \ldots, s_n)$ where $f \in \mathcal{F}$ and $s_1, \ldots, s_n$ are $\mathcal{PA}$-terms.*

The constraint $\top$ is omitted in a $\mathcal{PA}$-based rewrite rule $l \to r[\![\top]\!]$. A $\mathcal{PA}$-*based term rewrite system ($\mathcal{PA}$-based TRS)* $\mathcal{R}$ is a finite set of $\mathcal{PA}$-based rewrite rules. The rewrite relation of a $\mathcal{PA}$-based TRS requires that the constraint of the $\mathcal{PA}$-based rewrite rule is $\mathcal{PA}$-valid after being instantiated by the matching substitution. Notice that reductions are only possible at the root position.

Definition 2 (Rewrite Relation). *Let $s \to_{\mathcal{PA}\setminus\mathcal{R}} t$ iff there exist $l \to r[\![C]\!] \in \mathcal{R}$ and a substitution σ such that $s \simeq_{\mathcal{PA}} l\sigma$, $C\sigma$ is $\mathcal{PA}$-valid, and $t = r\sigma$.*[2]

[2] The use of $s \simeq_{\mathcal{PA}} l\sigma$ is due to the representation of integers and is needed to, e.g., identify the terms 0 and $-1 + 1$. This is only needed if l contains the same variable more than once or uses pattern-matching. If l has the form $f(x_1, \ldots, x_n)$ for pairwise distinct variables, then $s \simeq_{\mathcal{PA}} l\sigma$ can be replaced by $s = l\sigma$. The left-hand sides of the $\mathcal{PA}$-based TRSs obtained from imperative programs in Sect. 3 have this form.

Example 3. Using the $\mathcal{PA}$-based rewrite rule $\mathsf{eval}(x, y) \to \mathsf{eval}(x + 1, y)[\![x < y]\!]$, the term $\mathsf{eval}(-1, 1)$ can be reduced using the substitution $\sigma = \{x \mapsto -1, y \mapsto 1\}$ since $\mathsf{eval}(-1, 1) = \mathsf{eval}(x, y)\sigma$ and $(x < y)\sigma = (-1 < 1)$ is $\mathcal{PA}$-valid. Therefore, $\mathsf{eval}(-1, 1) \to_{\mathcal{PA}\backslash\mathcal{R}} \mathsf{eval}(-1+1, 1)$. The term $\mathsf{eval}(-1+1, 1)$ can be reduced once more, resulting in $\mathsf{eval}(-1 + 1 + 1, 1)$, which cannot be reduced anymore. ◊

3 Translating Imperative Programs into $\mathcal{PA}$-Based TRSs

In this paper, a simple imperative programming language where programs are formed according to the following grammar is considered.[3]

```
 prog ::= stmt
 stmt ::= skip | assign | stmt; stmt | if (cond) {stmt} else {stmt}
        | while (cond) {stmt} | break | continue | either {stmt} or {stmt}
 cond ::= "PA-constraints"
assign ::= (var_1, ..., var_k) := (exp_1, ..., exp_k)    for some k ≥ 1
  exp ::= "linear arithmetic expressions"
```

The constructs in this programming language have the standard (operational) semantics, i.e., `skip` is a do-nothing statement, `break` aborts execution of the innermost `while`-loop surrounding it, and `continue` aborts the current iteration of the innermost `while`-loop surrounding it and immediately starts the next iteration. The `either`-statement denotes a nondeterministic choice. For the $\mathcal{PA}$-constraints in `cond`, conjunction is written as `&&`, disjunction is written as `||`, and negation is written as `!`. It is assumed that every parallel assignment statement contains each variable of the program at most once on its left-hand side. A parallel assignment statement `(x1,...,xk) := (`e_1`,...,`e_k`)` with $k = 1$ is also written `x1 :=` e_1, and `x++` abbreviates `x := x + 1`. Similarly, `x--` abbreviates `x := x - 1`.

Translations from imperative programs into functional form (or rewrite rules) is "folklore", going back at least to McCarthy's work in the early 1960s [25]. The main idea for this is to model the state transformations occurring in the imperative program by suitable rewrite rules. Since $\mathcal{PA}$-based TRSs support the conditions of `while`-loops and `if`-statements, a translation is particularly simple for them. It proceeds as follows, where it is assumed that the program uses the variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$ and that the program contains m control points (i.e., program entry, `while`-loops and `if`-statements[4]). Then the i^{th} control point in the program is assigned a function symbol $\mathsf{eval}_i : \mathtt{int} \times \ldots \times \mathtt{int} \to \mathtt{univ}$ with n arguments. W.l.o.g. it can be assumed that each straight-line code segment between control points is a single assignment statement, `skip`, or empty.

For all $1 \leq i, j \leq m$ such that the j^{th} control point can be reached from the i^{th} control point by a straight-line code segment, each such straight-line code segment gives rise to a $\mathcal{PA}$-based rewrite rule of the form $\mathsf{eval}_i(\ldots) \to \mathsf{eval}_j(\ldots)[\![C]\!]$ where the constraint C is determined as follows. If the i^{th} control

[3] Several extension of the programming language are considered in [15].

[4] For termination purposes it is not necessary to consider the program exit.

point is a `while`-loop, then C is the condition of the `while`-loop or the negated condition of the `while`-loop, depending on whether the loop body is entered to reach the j^{th} control point or not. Similarly, if the i^{th} control point is an `if`-statement, then C is the condition or the negated condition of the `if`-statement, depending on whether the `then`- or the `else`-branch is taken.[5]

If the straight-line code segment is a `skip`-statement or empty, then the rewrite rule is $\mathsf{eval}_i(x_1,\ldots,x_n) \to \mathsf{eval}_j(x_1,\ldots,x_n)[\![C]\!]$. If the straight-line code segment is a parallel assignment statement $\texttt{(x}_{i_1}\texttt{,...,x}_{i_k}\texttt{) := (}e_{i_1}\texttt{,...,}e_{i_k}\texttt{)}$, then the rewrite rule is $\mathsf{eval}_i(x_1,\ldots,x_n) \to \mathsf{eval}_j(e_1,\ldots,e_n)[\![C]\!]$ where $e_h = e_{i_l}$ if $x_h = x_{i_l}$ for some i_l and $e_h = x_h$ otherwise. Notice that the arguments to eval_i on the left-hand side are thus pairwise distinct variables.

Example 4. Using this translation, the following is obtained.

```
while (x > 0 && y > 0) {
   if (x > y) {
      while (x > 0) {
        x--;
        y++
      }
   } else {
      while (y > 0) {
         y--;
         x++
      }
   }
}
```

$$\mathsf{eval}_1(x,y) \to \mathsf{eval}_2(x,y)\ [\![x>0 \wedge y>0 \wedge x>y]\!] \quad (1)$$
$$\mathsf{eval}_1(x,y) \to \mathsf{eval}_3(x,y)\ [\![x>0 \wedge y>0 \wedge x \not> y]\!] \quad (2)$$
$$\mathsf{eval}_2(x,y) \to \mathsf{eval}_2(x-1,y+1)\ [\![x>0]\!] \quad (3)$$
$$\mathsf{eval}_2(x,y) \to \mathsf{eval}_1(x,y)\ [\![x \not> 0]\!] \quad (4)$$
$$\mathsf{eval}_3(x,y) \to \mathsf{eval}_3(x+1,y-1)\ [\![y>0]\!] \quad (5)$$
$$\mathsf{eval}_3(x,y) \to \mathsf{eval}_1(x,y)\ [\![y \not> 0]\!] \quad (6)$$

Here, the outer `while`-loop is the first control point and the inner `while`-loop are the second and third control points, i.e., the optimization mentioned in Footnote 5 has been used. This $\mathcal{PA}$-based TRS is used as a running example. ◊

The following theorem is based on the observation that any state transition of the imperative program can be simulated by a rewrite sequence. Thus, termination of the imperative program is implied by termination of the $\mathcal{PA}$-based TRS obtained from it.

Theorem 5. *Let P be an imperative program. Then the above translation produces a $\mathcal{PA}$-based TRS $\mathcal{R}_P$ such that P is terminating if $\mathcal{R}_P$ is terminating.*[6]

[5] It is also possible to combine the control point of an `if`-statement with its (textually) preceding control point. Then C is the conjunction of the constraints obtained from these two control points.

[6] To see that the converse is not true in general, consider the (artificial) imperative program `x := 0; while (x > 0) { x++ }`. Then the $\mathcal{PA}$-based TRS generated from this program consists of $\mathsf{eval}_1(x) \to \mathsf{eval}_2(0)$ and $\mathsf{eval}_2(x) \to \mathsf{eval}_2(x+1)[\![x>0]\!]$ and is thus non-terminating. Here, the observation that the loop is never entered has not been utilized. Using static program analysis, this kind of information can be obtained automatically, cf. Sect. 9. Then, the imperative program would be translated into the $\mathcal{PA}$-based TRS consisting of $\mathsf{eval}_1(x) \to \mathsf{eval}_2(0)$ and $\mathsf{eval}_2(x) \to \mathsf{eval}_2(x+1)[\![x > 1 \wedge \bot]\!]$, which is terminating.

4 Characterizing Termination of $\mathcal{PA}$-Based TRSs

In order to verify termination of $\mathcal{PA}$-based TRSs, the notion of *chains* is used. Intuitively, a chain represents a possible sequence of rule applications in a reduction w.r.t. $\to_{\mathcal{PA}\setminus\mathcal{R}}$. In the following, it is always assumed that different (occurrences of) $\mathcal{PA}$-based rewrite rules are variable-disjoint, and that the domain of a substitution may be infinite. This allows for a single substitution in the following definition. Recall that $\to_{\mathcal{PA}\setminus\mathcal{R}}$ is only applied at the root position of a term, thus resulting in a definition of chains that is simpler than the corresponding notion needed in dependency pair methods [1,14].

Definition 6 ($\mathcal{R}$-Chains). *Let $\mathcal{R}$ be a $\mathcal{PA}$-based TRS. A (possibly infinite) sequence of $\mathcal{PA}$-based rewrite rules $l_1 \to r_1[\![C_1]\!], l_2 \to r_2[\![C_2]\!], \ldots$ from $\mathcal{R}$ is an* $\mathcal{R}$-chain *iff there exists a substitution σ such that $r_i\sigma \simeq_{\mathcal{PA}} l_{i+1}\sigma$ and $C_i\sigma$ is $\mathcal{PA}$-valid for all $i \geq 1$.*

Example 7. Continuing Ex. 4, the $\mathcal{R}$-chain

$$\begin{aligned}
\mathsf{eval}_1(x, y) &\to \mathsf{eval}_2(x, y)\ [\![x > 0 \wedge y > 0 \wedge x > y]\!] \\
\mathsf{eval}_2(x', y') &\to \mathsf{eval}_2(x' - 1, y' + 1)\ [\![x' > 0]\!] \\
\mathsf{eval}_2(x'', y'') &\to \mathsf{eval}_2(x'' - 1, y'' + 1)\ [\![x'' > 0]\!] \\
\mathsf{eval}_2(x''', y''') &\to \mathsf{eval}_1(x''', y''')\ [\![x''' \not> 0]\!]
\end{aligned}$$

can be built by considering the substitution $\sigma = \{x \mapsto 2, x' \mapsto 2, x'' \mapsto 1, x''' \mapsto 0, y \mapsto 1, y' \mapsto 1, y'' \mapsto 2, y''' \mapsto 3\}$. ◊

Using the notion of $\mathcal{R}$-chains, the following characterization of termination of a $\mathcal{PA}$-based TRS $\mathcal{R}$ is immediate. This corresponds to the soundness and completeness theorem for dependency pairs [1,14].

Theorem 8. *Let $\mathcal{R}$ be a $\mathcal{PA}$-based TRS. Then $\mathcal{R}$ is terminating if and only if there are no infinite $\mathcal{R}$-chains.*

In the next sections, various techniques for showing termination of $\mathcal{PA}$-based TRSs are developed. These techniques are stated independently of each other in the form of *termination processors*, following the dependency pair framework for ordinary term rewriting [19] and for term rewriting with built-in numbers [14]. The main motivation for this approach is that it allows to combine different termination techniques in a flexible manner since it typically does not suffice to just use a single technique in a successful termination proof.

Termination processors are used to transform a $\mathcal{PA}$-based TRS into a (finite) set of simpler $\mathcal{PA}$-based TRSs for which termination is (hopefully) easier to show. A termination processor Proc is *sound* iff for all $\mathcal{PA}$-based TRSs $\mathcal{R}$, $\mathcal{R}$ is terminating whenever all $\mathcal{PA}$-based TRSs in $\mathsf{Proc}(\mathcal{R})$ are terminating. Notice that $\mathsf{Proc}(\mathcal{R}) = \{\mathcal{R}\}$ is possible. This can be interpreted as a failure of Proc and indicates that a different termination processor should be applied.

Using sound termination processors, a termination proof of a $\mathcal{PA}$-based TRS $\mathcal{R}$ then consists of the recursive application of these processors. If all $\mathcal{PA}$-based TRSs obtained in this process are transformed into $\emptyset$, then $\mathcal{R}$ is terminating.

5 Termination Graphs

Notice that a $\mathcal{PA}$-based TRS $\mathcal{R}$ may give rise to infinitely many different $\mathcal{R}$-chains. This section introduces a method that represents these infinitely many chains in a finite graph. Then, each $\mathcal{R}$-chain (and thus each computation path in the imperative program) corresponds to a path in this graph. By considering the strongly connected components of this graph, it then becomes possible to decompose a $\mathcal{PA}$-based TRS into several independent $\mathcal{PA}$-based TRSs by determining which $\mathcal{PA}$-based rewrite rules may follow each other in a chain.[7]

The termination processor based on this idea uses *termination graphs*. This notion corresponds to the notion of dependency graphs used [1,14].

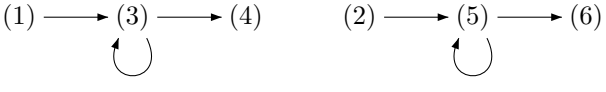
Definition 9 (Termination Graphs). *For a $\mathcal{PA}$-based TRS $\mathcal{R}$, the nodes of the $\mathcal{R}$-*termination graph $\mathsf{TG}(\mathcal{R})$ *are the rules from $\mathcal{R}$ and there is an arc from $l_1 \to r_1[\![C_1]\!]$ to $l_2 \to r_2[\![C_2]\!]$ iff $l_1 \to r_1[\![C_1]\!]$, $l_2 \to r_2[\![C_2]\!]$ is an $\mathcal{R}$-chain.*

In contrast to [1,14], it is decidable whether there is an arc from $l_1 \to r_1[\![C_1]\!]$ to $l_2 \to r_2[\![C_2]\!]$. Let $r_1 = f(s_1, \ldots, s_n)$ and $l_2 = g(t_1, \ldots, t_m)$. If $f \neq g$ then there is no arc. Otherwise, $n = m$ and there is an arc between the $\mathcal{PA}$-based rewrite rules iff there is a substitution σ such that $s_1\sigma \simeq t_1\sigma \wedge \ldots s_n\sigma \simeq t_n\sigma \wedge C_1\sigma \wedge C_2\sigma$ is $\mathcal{PA}$-valid, i.e., iff $s_1 \simeq t_1 \wedge \ldots s_n \simeq t_n \wedge C_1 \wedge C_2$ is $\mathcal{PA}$-satisfiable.

A set $\mathcal{R}' \subseteq \mathcal{R}$ of $\mathcal{PA}$-based rewrite rules is a *strongly connected subgraph (SCS)* of $\mathsf{TG}(\mathcal{R})$ iff for all $\mathcal{PA}$-based rewrite rules $l_1 \to r_1[\![C_1]\!]$ and $l_2 \to r_2[\![C_2]\!]$ from $\mathcal{R}'$ there exists a non-empty path from $l_1 \to r_1[\![C_1]\!]$ to $l_2 \to r_2[\![C_2]\!]$ that only traverses $\mathcal{PA}$-based rewrite rules from $\mathcal{R}'$. An SCS is a *strongly connected component (SCC)* if it is not a proper subset of any other SCS. Since every infinite $\mathcal{R}$-chain contains an infinite tail that stays within one SCS of $\mathsf{TG}(\mathcal{R})$, it is sufficient to prove the absence of infinite chains for each SCC separately.

Theorem 10 (Processor Based on Termination Graphs). *The termination processor with $\mathsf{Proc}(\mathcal{R}) = \{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$, where $\mathcal{R}_1, \ldots, \mathcal{R}_n$ are the SCCs of $\mathsf{TG}(\mathcal{R})$, is sound.*[8]

Example 11. Continuing Ex. 4, the $\mathcal{PA}$-based TRS $\{(1) - (6)\}$ gives rise to the following termination graph.



The termination graph contains two SCC and the termination processor of Thm. 10 returns the $\mathcal{PA}$-based TRSs $\{(3)\}$ and $\{(5)\}$, which can be handled independently of each other. ◊

[7] Notice that this fundamentally differs from control flow graphs as used in, e.g., [5,6]. The edges in the control flow graph correspond to the rewrite rules in $\mathcal{R}$. Thus, the termination graph determines which edges in the control flow graph may follow each other in an execution of the program.

[8] Notice, in particular, that $\mathsf{Proc}(\emptyset) = \emptyset$. Also, notice that $\mathcal{PA}$-based rewrite rules with unsatisfiable constraints are not connected to any $\mathcal{PA}$-based rewrite rule and do thus not occur in any SCC.

6 $\mathcal{PA}$-Polynomial Interpretations

In this section, well-founded relations on terms are considered and it is shown that $\mathcal{PA}$-based rewrite rules may be deleted from a $\mathcal{PA}$-based TRS if their left-hand side is strictly "bigger" than their right-hand side. This corresponds to the use of reduction pairs in dependency pair methods [1,14]. A promising way for the generation of well-founded relations is the use of polynomial interpretations [24]. In contrast to [24], $\mathcal{PA}$-based TRSs allow for the use of polynomial interpretations with coefficients from $\mathbb{Z}$. In the term rewriting literature, polynomial interpretations with coefficients from $\mathbb{Z}$ have been used in [21,20,14,17].

A *$\mathcal{PA}$-polynomial interpretation* maps each symbol $f \in \mathcal{F}$ to a polynomial over $\mathbb{Z}$ such that $\mathcal{P}ol(f) \in \mathbb{Z}[x_1, \ldots, x_n]$ if f has n arguments. The mapping $\mathcal{P}ol$ is then extended to terms from $\mathcal{T}(\mathcal{F}, \mathcal{F}_{\mathcal{PA}}, \mathcal{V})$ by letting $[f(t_1, \ldots, t_n)]_{\mathcal{P}ol} = \mathcal{P}ol(f)(t_1, \ldots, t_n)$ for all $f \in \mathcal{F}$. For this, notice that $\mathcal{PA}$-terms can already be interpreted as (linear) polynomials. Now $\mathcal{PA}$-polynomial interpretations generate relations on terms as follows. Here, $[s]_{\mathcal{P}ol} \geq 0$ is needed for well-foundedness.

Definition 12 ($\succ_{\mathcal{P}ol}$ and $\gtrsim_{\mathcal{P}ol}$). *Let $\mathcal{P}ol$ be a $\mathcal{PA}$-polynomial interpretation, let $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{F}_{\mathcal{PA}}, \mathcal{V})$, and let C be a $\mathcal{PA}$-constraint. Then*

- *$s \succ^C_{\mathcal{P}ol} t$ iff $\forall^*.\ C \Rightarrow [s]_{\mathcal{P}ol} \geq 0$ and $\forall^*.\ C \Rightarrow [s]_{\mathcal{P}ol} > [t]_{\mathcal{P}ol}$ are true in the integers, where, $\forall^*$ denotes the universal closure of the formula.*
- *$s \gtrsim^C_{\mathcal{P}ol} t$ iff $\forall^*.\ C \Rightarrow [s]_{\mathcal{P}ol} \geq [t]_{\mathcal{P}ol}$ is true in the integers.*

Using $\mathcal{PA}$-polynomial interpretations, $\mathcal{PA}$-based rewrite rules $l \to r[\![C]\!]$ with $l \succ^C_{\mathcal{P}ol} r$ can be removed from a $\mathcal{PA}$-based TRS if all remaining $\mathcal{PA}$-based rewrite rules $l' \to r'[\![C']\!]$ satisfy $l' \gtrsim^{C'}_{\mathcal{P}ol} r'$.

Theorem 13 (Processor Based on $\mathcal{PA}$-Polynomial Interpretations). *Let $\mathcal{P}ol$ be a $\mathcal{PA}$-polynomial interpretation and let* Proc *be the termination processor with* $\mathsf{Proc}(\mathcal{R}) =$

- *$\{\mathcal{R} - \mathcal{R}'\}$, if $\mathcal{R}' \subseteq \mathcal{R}$ such that*
  - *$l \succ^C_{\mathcal{P}ol} r$ for all $l \to r[\![C]\!] \in \mathcal{R}'$, and*
  - *$l \gtrsim^C_{\mathcal{P}ol} r$ for all $l \to r[\![C]\!] \in \mathcal{R} - \mathcal{R}'$.*
- *$\{\mathcal{R}\}$, otherwise.*

Then Proc *is sound.*

Example 14. Continuing Ex. 11, recall the $\mathcal{PA}$-based TRSs $\{(3)\}$ and $\{(5)\}$ that can be handled independently of each other. For the $\mathcal{PA}$-based TRS $\{(3)\}$, a $\mathcal{PA}$-polynomial interpretation with $\mathcal{P}ol(\mathsf{eval}_2) = x_1$ can be used. With this interpretation, $\mathsf{eval}_2(x, y) \succ^{[\![x>0]\!]}_{\mathcal{P}ol} \mathsf{eval}_2(x - 1, y + 1)$ since $\forall x.\ x > 0 \Rightarrow x \geq 0$ and $\forall x.\ x > 0 \Rightarrow x > x - 1$ are true in the integers. Applying the termination processor of Thm. 13, the first $\mathcal{PA}$-based TRS is thus transformed into the trivial $\mathcal{PA}$-based TRS $\emptyset$. The second $\mathcal{PA}$-based TRS can be handled similarly using a $\mathcal{PA}$-polynomial interpretation with $\mathcal{P}ol(\mathsf{eval}_3) = x_2$. ◊

If $\mathcal{P}ol(f)$ is a linear polynomial for all $f \in \mathcal{F}$, then it is decidable whether $l \succ^C_{\mathcal{P}ol} r$ or $l \gtrsim^C_{\mathcal{P}ol} r$. A method for the automatic generation of suitable $\mathcal{PA}$-polynomial interpretations is presented in Sect. 8.

7 Chaining

It is possible to replace a $\mathcal{PA}$-based rewrite rule $l \to r[\![C]\!]$ by a set of new $\mathcal{PA}$-based rewrite rules that are formed by chaining $l \to r[\![C]\!]$ to the $\mathcal{PA}$-based rewrite rules that may follow it in an infinite chain.[9] This way, further information about the possible substitutions used in a chain can be obtained. Chaining of $\mathcal{PA}$-based rewrite rules corresponds to executing bigger parts of the imperative program at once, spanning several control points. Within this section, it is assumed that all $\mathcal{PA}$-based rewrite rules have the form $f(x_1, \ldots, x_n) \to r[\![C]\!]$, where $x_1, \ldots, x_n$ are pairwise distinct variables. Recall that the $\mathcal{PA}$-based rewrite rules generated by the translation from Sect. 3 satisfy this requirement.

Example 15. Consider the following imperative program and the $\mathcal{PA}$-based TRS generated from it.

```
while (x > z) {
   while (y > z) {
      y--
   }
   x--
}
```

$$\mathsf{eval}_1(x,y,z) \to \mathsf{eval}_2(x,y,z)\ [\![x > z]\!] \quad (7)$$
$$\mathsf{eval}_2(x,y,z) \to \mathsf{eval}_2(x,y-1,z)\ [\![y > z]\!] \quad (8)$$
$$\mathsf{eval}_2(x,y,z) \to \mathsf{eval}_1(x-1,y,z)\ [\![y \not> z]\!] \quad (9)$$

The $\mathcal{PA}$-based TRS $\{(7),(8),(9)\}$ is transformed into $\{(7),(9)\}$ using a $\mathcal{PA}$-polynomial interpretation with $\mathcal{P}ol(\mathsf{eval}_1) = \mathcal{P}ol(\mathsf{eval}_2) = x_2 - x_3$. The $\mathcal{PA}$-based TRS $\{(7),(9)\}$ cannot be handled by the techniques presented so far. Notice that in any chain, each occurrence of the $\mathcal{PA}$-based rewrite rule (7) is followed by an occurrence of the $\mathcal{PA}$-based rewrite rule (9). Thus, (7) may be replaced by a new $\mathcal{PA}$-based rewrite rule that simulates an application of (7) followed by an application of (9). This new $\mathcal{PA}$-based rewrite rule is

$$\mathsf{eval}_1(x,y,z) \to \mathsf{eval}_1(x-1,y,z)\ [\![x > z \wedge y \not> z]\!] \quad (7.9)$$

The $\mathcal{PA}$-based TRS $\{(7.9),(9)\}$ is first transformed into the $\mathcal{PA}$-based TRS $\{(7.9)\}$ using the termination graph. Then, the $\mathcal{PA}$-based TRS $\{(7.9)\}$ can be handled using a $\mathcal{PA}$-polynomial interpretation with $\mathcal{P}ol(\mathsf{eval}_1) = x_1 - x_3$. ◊

Formally, this idea can be stated as the following termination processor. In practice, its main use is the propagation of $\mathcal{PA}$-constraints. Notice that chaining of $\mathcal{PA}$-based rewrite rules is easily possible if the left-hand sides have the form $f(x_1, \ldots, x_n)$. Also, notice that the rule $l \to f(s_1, \ldots, s_n)[\![C]\!]$ is *replaced* by the rules that are obtained by chaining.

Theorem 16 (Processor Based on Chaining). *The termination processor with* $\mathsf{Proc}(\mathcal{R}) = \{\mathcal{R}_1 \cup \mathcal{R}_2\}$ *is sound, where, for a rule* $l \to f(s_1, \ldots, s_n)[\![C]\!] \in \mathcal{R}$, $\mathcal{R}_1 = \mathcal{R} - \{l \to f(s_1, \ldots, s_n)[\![C]\!]\}$ *and* $\mathcal{R}_2 = \{l \to r'\mu[\![C \wedge C'\mu]\!] \mid f(x_1, \ldots, x_n) \to r'[\![C']\!] \in \mathcal{R},\ \mu = \{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}\}$.

[9] Dually, it is possible to consider the $\mathcal{PA}$-based rewrite rules that may *precede* it. Notice that chaining conceptually differs from the use of conditional constraints in [20,17]. There, conditional constraints are solely used for the generation of reduction pairs. Chaining is independent of reduction pairs and transforms a $\mathcal{PA}$-based TRS.

8 Implementation

In order to show the practicality of the proposed approach, the termination analysis method for $\mathcal{PA}$-based TRSs has been implemented in the tool `pasta` ($\mathcal{PA}$-based Term Rewrite System Termination Analyzer). While an implementation in an existing termination tool for TRSs such as AProVE [18] or T$_{\text{T}}$T$_2$ [23] would also be possible, an implementation from scratch more clearly demonstrates that the approach can easily be implemented. `pasta` has been written in OCaml and consists of about 1500 lines of code. The whole implementation took less than 30 person-hours to complete. An empirical evaluation shows that `pasta` is very successful and efficient.

8.1 General Overview

The first decision that has to be made is the order in which the termination processors from Sections 5–7 are applied. For this, the loop given below is used.

```
todo := SCC(R)
while todo ≠ ∅ do
  P := pick-and-remove(todo)
  P' := polo(P)
  if P = P' then
    P' := chain(P)
    if P = P' then
      return "Failure"
    end if
  end if
  todo := todo ∪ SCC(P')
end while
return "Termination shown"
```

Here, SCC is the termination processor of Thm. 10 that returns the SCCs of the termination graph, polo is the termination processor of Thm. 13 using linear $\mathcal{PA}$-polynomial interpretations that deletes $\mathcal{PA}$-based rewrite rules which are decreasing w.r.t. $\succ_{\mathcal{P}ol}$, and chain is the termination processor of Thm. 16 that combines $\mathcal{PA}$-based rewrite rules.

SCC builds the termination graph using a decision procedure for $\mathcal{PA}$-satisfiability. Then, the standard graph algorithm is used to compute the SCCs. In `pasta`, the library ocamlgraph[10] [7] is used for graph manipulations, and the SMT solver `yices`[11] [13] is used as a decision procedure for $\mathcal{PA}$-satisfiability. The most non-trivial part of the implementation is the function polo for the automatic generation of (linear) $\mathcal{PA}$-polynomial interpretations.

8.2 Generation of $\mathcal{PA}$-Polynomial Interpretations

For the automatic generation, a linear *parametric* $\mathcal{PA}$-polynomial interpretation is used, i.e., an interpretation where the coefficients of the polynomials are not integers but parameters that have to be determined. Thus, $\mathcal{P}ol(f) = a_1x_1 + \ldots + a_nx_n + c$ for each function symbol f with n arguments, where the a_i and c are parameters that have to be determined.

In this section, it is assumed that the constraints of all $\mathcal{PA}$-based rewrite rules are conjunctions of atomic $\mathcal{PA}$-constraints. This can be achieved by a conversion into disjunctive normal form (DNF) and the introduction of one rewrite rule

[10] Freely available from `http://ocamlgraph.lri.fr/`

[11] Available from `http://yices.csl.sri.com/`

for each dual clause in this DNF.[12] Recall that the termination processor of Thm. 13 operating on a $\mathcal{PA}$-based TRS $\mathcal{R}$ aims at generating a $\mathcal{PA}$-polynomial interpretation $\mathcal{P}ol$ such that

- $\forall^*.\ C \Rightarrow [l]_{\mathcal{P}ol} - [r]_{\mathcal{P}ol} > 0$ and $\forall^*.\ C \Rightarrow [l]_{\mathcal{P}ol} \geq 0$ are true in the integers for all $l \to r[\![C]\!] \in \mathcal{R}'$ for some non-empty $\mathcal{R}' \subseteq \mathcal{R}$ and
- $\forall^*.\ C \Rightarrow [l]_{\mathcal{P}ol} - [r]_{\mathcal{P}ol} \geq 0$ is true in the integers for all $l \to r[\![C]\!] \in \mathcal{R} - \mathcal{R}'$

Notice that $[l]_{\mathcal{P}ol}$ and $[l]_{\mathcal{P}ol} - [r]_{\mathcal{P}ol}$ are linear parametric polynomials, i.e., polynomials over the variables whose coefficients are linear polynomials over the parameters. Here, linearity of the coefficients is due to the fact that function symbols from $\mathcal{F}$ do not occur nested in terms. For instance, if $[l] = \mathsf{eval}(x, x)$ and $\mathcal{P}ol(\mathsf{eval}) = ax_1 + bx_2 + c$, then $[l]_{\mathcal{P}ol} = (a + b)x + c$.

In order to determine the parameters such that $\forall^*.\ C \Rightarrow [l]_{\mathcal{P}ol} - [r]_{\mathcal{P}ol} \geq 0$ is true in the integers for all $l \to r[\![C]\!] \in \mathcal{R}$, sufficient conditions on the parameters are derived and it is checked whether these conditions are satisfiable. The derivation of conditions is done independently for each of the $\mathcal{PA}$-based rewrite rules, but the check for satisfiability of the conditions has to consider all $\mathcal{PA}$-based rewrite rules since they need to be oriented using the same $\mathcal{PA}$-polynomial interpretation.

For a single $\mathcal{PA}$-based rewrite rule $l \to r[\![C]\!]$, the conditions on the parameters are obtained as follows, where $p = [l]_{\mathcal{P}ol} - [r]_{\mathcal{P}ol}$:

1. C is transformed into a conjunction of atomic $\mathcal{PA}$-constraints of the form $\sum_{i=1}^{n} a_i x_i + c \geq 0$ where $a_1, \ldots, a_n, c \in \mathbb{Z}$.
2. Use the $\mathcal{PA}$-constraints from step 1 to derive upper and/or lower bounds on the variables in p.
3. Use the bounds from step 2 to derive conditions on the parameters.

A similar method for the generation of polynomial interpretations in the presence on conditions is presented in [17]. Indeed, the main inference rule A used in [17] has been derived from the rules $\mathsf{Express}^+$ and $\mathsf{Express}^-$ used in step 2 of the method presented here.

Step 1: Transformation of C. This is straightforward: $s \simeq t$ is transformed into $s - t \geq 0$ and $t - s \geq 0$, $s \geq t$ is transformed into $s - t \geq 0$, and $s > t$ is transformed into $s - t - 1 \geq 0$.

Step 2: Deriving upper and/or lower bounds. The $\mathcal{PA}$-constraints obtained after step 1 might already contain upper and/or lower bounds on the variables, where a lower bound has the form $x + c \geq 0$ and an upper bound has the from $-x + c \geq 0$ for some $c \in \mathbb{Z}$. Otherwise, it might be possible to obtain such bounds as follows.

An atomic constraint of the form $ax + c \geq 0$ with $a \neq 0, 1, -1$ that contains only one variable gives a bound on that variable that can be obtained by dividing

[12] Recall that a formula in DNF is a disjunction of conjunctions. The conjunctions are called *dual clauses*.

by $|a|$ and rounding. For example, the $\mathcal{PA}$-constraint $2x + 3 \geq 0$ is transformed into $x + 1 \geq 0$, and $-3x - 2 \geq 0$ is transformed into $-x - 1 \geq 0$.

An atomic $\mathcal{PA}$-constraint with more than one variable can be used to express a variable x occurring with coefficient 1 in terms of the other variables and a fresh slack variable w with $w \geq 0$. This allows to eliminate x from the polynomial p and at the same time gives the lower bound 0 on the slack variable w. For example, $x - 2y \geq 0$ can be used to eliminate the variable x by replacing it with $2y + w$. Similar reasoning applies if the variable x occurs with coefficient -1.

These ideas are formalized in the transformation rules given below that operate on triples $\langle C_1, C_2, q\rangle$ where C_1 and C_2 are sets of atomic $\mathcal{PA}$-constraints and q is a linear parametric polynomial. Here, C_1 only contains $\mathcal{PA}$-constraints of the form $\pm x_i + c \geq 0$ giving upper and/or lower bounds on the variable x_i and C_2 contains arbitrary atomic $\mathcal{PA}$-constraints. The initial triple is $\langle \emptyset, C, p\rangle$.

$$\mathsf{Strengthen}\ \frac{\langle C_1,\ C_2 \uplus \{a_i x_i + c \geq 0\},\ q\rangle}{\langle C_1 \cup \left\{ \frac{a_i}{|a_i|} x_i + \lfloor \frac{c}{|a_i|} \rfloor \geq 0 \right\},\ C_2,\ q\rangle} \quad \text{if } a_i \neq 0$$

$$\mathsf{Express}^{+}\ \frac{\langle C_1,\ C_2 \uplus \left\{\sum_{i=1}^{n} a_i x_i + c \geq 0\right\},\ q\rangle}{\langle C_1 \cup \{w \geq 0\},\ C_2\sigma,\ q\sigma\rangle} \quad \begin{array}{l}\text{if } a_j = 1 \text{ and } \sigma \text{ substitutes}\\ x_j \text{ by } -\sum_{i\neq j} a_i x_i - c + w\\ \text{for a fresh slack variable } w\end{array}$$

$$\mathsf{Express}^{-}\ \frac{\langle C_1,\ C_2 \uplus \left\{\sum_{i=1}^{n} a_i x_i + c \geq 0\right\},\ q\rangle}{\langle C_1 \cup \{w \geq 0\},\ C_2\sigma,\ q\sigma\rangle} \quad \begin{array}{l}\text{if } a_j = -1 \text{ and } \sigma \text{ substitutes}\\ x_j \text{ by } \sum_{i\neq j} a_i x_i + c - w\\ \text{for a fresh slack variable } w\end{array}$$

Step 3: Deriving conditions on the parameters. After finishing step 2, a final triple $\langle C_1, C_2, q\rangle$ is obtained. If C_1 contains more than one bound on a variable x_i, then it suffices to consider the maximal lower bound and the minimal upper bound. The bounds in C_1 are used in combination with absolute positiveness [22] in order to obtain conditions on the parameters that make $q = \sum_{i=1}^{n} p_i x_i + p_0$ non-negative for all instantiations satisfying $C_1 \cup C_2$.

If C_1 contains a lower bound of the form $x_j + c \geq 0$ for the variable x_j, then notice that $q = \sum_{i=1}^{n} p_i x_i + p_0$ can also be written as $q = \sum_{i\neq j} p_i x_i + p_j(x_j + c) + p_0 - p_j c$. Since $x_j + c \geq 0$ is assumed, the absolute positiveness test requires $p_j \geq 0$ as a condition on p_j.[13] Similarly, if $-x_j + c \geq 0$ occurs in C_1, then q can be written as $q = \sum_{i\neq j} p_i x_i - p_j(-x_j + c) + p_0 + p_j c$ and $-p_j \geq 0$ is obtained as a condition on p_j. If C_1 does not contain any upper or lower bound on a variable x_j, then $p_j = 0$ is obtained by the absolute positiveness test. After all variables of q have been processed in this fashion, it additionally needs to be required that the constant term of the final polynomial is non-negative as well.

[13] Alternatively, reasoning similar to rules $\mathsf{Express}^{+}$ can be used, i.e., if C_1 contains $x_j + c \geq 0$, then x_j could be replaced by $-c + w$, where $w \geq 0$. Both methods produce the same conditions on the parameters.

For example, if $C_1 = \{x+1 \geq 0,\ -y-1 \geq 0\}$ and $q = (a+b)x+by+c$, then q can also be written as $q = (a+b)(x+1) - b(-y-1) + c - (a+b) - b$ and the absolute positiveness test requires $a+b \geq 0$, $-b \geq 0$, and $c-a-2b \geq 0$.

Summarizing this method, the following algorithm is used in order to obtain conditions D on the parameters. Here, $\mathsf{sign}(C)$ is 1 if C is of the form $x_i + c \geq 0$ and -1 if C is of the form $-x_i + c \geq 0$.

$D := \mathsf{true}$
$r := p_0$
for $1 \leq i \leq n$ **do**
  take constraint C of the form $\pm x_i + c \geq 0$ from C_1
  if none such C exists **then**
    $D := D \wedge p_i = 0$
  else
    $D := D \wedge \mathsf{sign}(C) \cdot p_i \geq 0$
    $r := r - \mathsf{sign}(C) \cdot c \cdot p_i$
  end if
end for
$D := D \wedge r \geq 0$

Automatically finding strictly decreasing rules. For the termination processor of Thm. 13, it also has to be ensured that $\mathcal{R}'$ is non-empty, i.e., that at least one $\mathcal{PA}$-based rewrite rule is decreasing w.r.t. $\succ_{\mathcal{P}ol}$. Let $l \to r[\![C]\!]$ be a $\mathcal{PA}$-based rewrite rule that should satisfy $l \succ^{C}_{\mathcal{P}ol} r$. Then, $\forall^*.\ C \Rightarrow [l]_{\mathcal{P}ol} \geq 0$ gives rise to conditions D_1 on the parameters as above. The second condition, i.e., $\forall^*.\ C \Rightarrow [l]_{\mathcal{P}ol} - [r]_{\mathcal{P}ol} > 0$, gives rise to conditions D_2 just as above, with the only difference that the last line of the algorithm now requires $r > 0$.

Given a set of rules $\{l_1 \to r_1[\![C_1]\!], \ldots, l_n \to r_n[\![C_n]\!]\}$, the final constraint on the parameters is then $\bigwedge_{i=1}^{n} D^i \wedge \bigvee_{i=1}^{n} (D_1^i \wedge D_2^i)$ where the D^i are obtained from $\forall^*.\ C_i \Rightarrow [l_i]_{\mathcal{P}ol} - [r_i]_{\mathcal{P}ol} \geq 0$, the D_1^i are obtained from $\forall^*.\ C_i \Rightarrow [l_i]_{\mathcal{P}ol} \geq 0$, and the D_2^i are obtained from $\forall^*.\ C_i \Rightarrow [l_i]_{\mathcal{P}ol} - [r_i]_{\mathcal{P}ol} > 0$. This constraint can be given to a witness-producing decision procedure for $\mathcal{PA}$-satisfiability in order to obtain values for the parameters. $\mathcal{R}'$ can then be obtained.

Example 17. The method is illustrated on the termination problem $\{(8)\}$ from Ex. 15 consisting of the rewrite rule $\mathsf{eval}_2(x,y,z) \to \mathsf{eval}_2(x,y-1,z)\ [\![y > z]\!]$. For this, a parametric $\mathcal{PA}$-polynomial interpretation with $\mathcal{P}ol(\mathsf{eval}_2) = ax_1 + bx_2 + cx_3 + d$ is used, where a, b, c, d are parameters that need to be determined. Thus, the goal is to instantiate the parameters in such a way that $\mathsf{eval}_2(x,y,z) \succ^{[\![y>z]\!]}_{\mathcal{P}ol} \mathsf{eval}_2(x,y-1,z)$, i.e., such that $\forall x,y,z.\ y > z \Rightarrow [\mathsf{eval}_2(x,y,z)]_{\mathcal{P}ol} \geq 0$ and $\forall x,y,z.\ y > z \Rightarrow [\mathsf{eval}_2(x,y,z)]_{\mathcal{P}ol} - [\mathsf{eval}_2(x,y-1,z)]_{\mathcal{P}ol} > 0$ are true in the integers. Notice that $[\mathsf{eval}_2(x,y,z)]_{\mathcal{P}ol} = ax+by+cz+d$ and $[\mathsf{eval}_2(x,y-1,z)]_{\mathcal{P}ol} = ax + by + cz - b + d$. Therefore, $[\mathsf{eval}_2(x,y,z)]_{\mathcal{P}ol} - [\mathsf{eval}_2(x,y-1,z)]_{\mathcal{P}ol} = b$.

For the first formula, the constraint $y > z$ is transformed into $y - z - 1 \geq 0$ in step 1. In step 2, the transformation rules from above are applied to the triple $\langle \emptyset, \{y - z - 1 \geq 0\}, ax + by + cz + d \rangle$. Using the $\mathsf{Express}^+$-rule with $\sigma = \{y \mapsto z + w + 1\}$, the triple $\langle \{w \geq 0\},\ \emptyset,\ ax + (b+c)z + bw + b + d \rangle$ is obtained and step 3 gives $a = 0 \wedge b + c = 0 \wedge b \geq 0 \wedge b + d \geq 0$ as conditions on the parameters.

For the second formula from above, $b > 0$ is immediately obtained as a condition. The final constraint on the parameters is thus $a = 0 \wedge b + c = 0 \wedge b \geq 0 \wedge b + d \geq 0 \wedge b > 0$. This constraint is satisfiable and `yices` returns the witness values $a = 0, b = 1, c = -1, d = 0$, giving rise to the $\mathcal{PA}$-polynomial interpretation $\mathcal{P}ol(\mathsf{eval}_2) = x_2 - x_3$ already considered in Ex. 15. $\Diamond$

A potential relationship between the method presented in this section and methods for the generation of linear ranking functions based on Farka's Lemma [5,6] has been pointed out to us. While time constraints did not permit us to investigate this further, we would like to point out that the proposed method for the generation of linear $\mathcal{PA}$-polynomial interpretations extends to the generation of non-linear $\mathcal{PA}$-polyonial interpretations, while it is unclear whether the methods of [5,6] can be extended to the generation of non-linear ranking functions since Farka's Lemma does not extend to the non-linear case.

9 Conclusions and Future Work

This paper has proposed a method for showing termination of imperative programs operating on integers that is based on ideas from the term rewriting literature. For this, a translation from imperative programs into constrained term rewrite systems operating on integers has been introduced. Then, techniques for showing termination of such $\mathcal{PA}$-based TRSs have been developed.

An implementation of this approach has been evaluated on a collection of 40 examples that were taken from various places, including several recent papers on the termination of imperative programs [2,3,4,5,6,9,10,27,28]. The collection of examples includes "classical" algorithms such as binary search, bubblesort, heapsort, and the computation of the greatest common divisor. Twelve out of these 40 examples (e.g., the heapsort example from [12]) require simple invariants on the program variables (such as "a variable is always non-negative") or simple reasoning of the kind "if variables do not change between control points, then relations that are true for them at the first control point are still true at the second control point" for a successful termination proof. This kind of information can be obtained automatically using static program analysis tools such as `Interproc`[14] [26]. The translation into $\mathcal{PA}$-based TRSs from Sect. 3 can immediately use this information by adding it to the constraints of the generated rewrite rules.

The tool `pasta` has been able to show termination of all examples fully automatically, on average taking less than 0.05 seconds[15] for each example, with the longest time being a little less than a third of a second. Thus, `pasta` shows the practicality and effectiveness of the proposed approach on a collection of "typical" examples. For comparison, the tool AProVE Integer [17] was run on the same collection of examples with a timeout of 60 seconds for each example. A summary of the results in given below, including the run-times for the classical

[14] Freely available from `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/`

[15] All times were obtained on a 2.2 GHz AMD Athlon$^{\text{TM}}$ with 2 GB main memory.

algorithms mentioned above. Notice that `pasta` is orders of magnitude faster and slightly more powerful than AProVE Integer.

	`pasta`	AProVE Integer
Successful proofs	40	38
Binary Search	0.284s	10.76s
Bubblesort	0.042s	1.72s
Heapsort	0.318s	50.12s
Greatest Common Divisor	0.032s	8.45s
Total time for all 40 examples	1.910s	323.09s
Average time	0.048s	8.08s

Notice that an empirical comparison with the methods of [2,3,4,5,6,9,10,27,28] is not possible since implementations of those methods are not publicly available. The examples, detailed results, the termination proofs generated by `pasta`, and the tool `pasta` itself are available at `http://www.cs.unm.edu/~spf/pasta/`.

There are several directions for extending the work presented in this paper. Since $\mathcal{PA}$-based TRSs are limited to Presburger arithmetic, an extension to non-linear arithmetic should be considered. Then, the constraint language becomes undecidable, but SMT-solvers such as `yices` [13] still provide incomplete methods. The only termination processor that relies on the decidability of the constraints is the termination graph, but incomplete methods would still result in a sound method since an over-approximation of the real termination graph would be computed. The method for the automatic generation of polynomial interpretations requires some extension to non-linear constraints, or alternatively the method proposed in [17] could be used. Furthermore, the imperative language should be extended to support functions. This can be done at the price of loosing the simple term structure in $\mathcal{PA}$-based rewrite rules, thus requiring the unrestricted dependency pair method [1,14] for proving termination.

Acknowledgements. We thank the anonymous reviewers for valuable comments.

References

1. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. TCS 236, 133–178 (2000)
2. Bradley, A., Manna, Z., Sipma, H.: Linear ranking with reachability. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 491–504. Springer, Heidelberg (2005)
3. Bradley, A., Manna, Z., Sipma, H.: Termination of polynomial programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 113–129. Springer, Heidelberg (2005)
4. Chawdhary, A., Cook, B., Gulwani, S., Sagiv, M., Yang, H.: Ranking abstractions. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 148–162. Springer, Heidelberg (2008)
5. Colón, M., Sipma, H.: Synthesis of linear ranking functions. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 67–81. Springer, Heidelberg (2001)

6. Colón, M., Sipma, H.: Practical methods for proving program termination. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 442–454. Springer, Heidelberg (2002)
7. Conchon, S., Filliâtre, J.-C., Signoles, J.: Designing a generic graph library using ML functors. In: TFP 2007, pp. 124–140 (2008)
8. Contejean, E., Marché, C., Tomás, A.P., Urbain, X.: Mechanically proving termination using polynomial interpretations. JAR 34, 325–363 (2005)
9. Cook, B., Podelski, A., Rybalchenko, A.: Abstraction refinement for termination. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 87–101. Springer, Heidelberg (2005)
10. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI 2006, pp. 415–426 (2006)
11. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond safety. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 415–418. Springer, Heidelberg (2006)
12. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL 1978, pp. 84–96 (1978)
13. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
14. Falke, S., Kapur, D.: Dependency pairs for rewriting with built-in numbers and semantic data structures. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 94–109. Springer, Heidelberg (2008)
15. Falke, S., Kapur, D.: A term rewriting approach to the automated termination analysis of imperative programs. Technical Report TR-CS-2009-02, Department of Computer Science, University of New Mexico (2009), `http://www.cs.unm.edu/research/tech-reports/`
16. Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: SAT solving for termination analysis with polynomial interpretations. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 340–354. Springer, Heidelberg (2007)
17. Fuhs, C., Giesl, J., Plücker, M., Schneider-Kamp, P., Falke, S.: Proving termination of integer term rewriting. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 32–47. Springer, Heidelberg (2009)
18. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic termination proofs in the dependency pair framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006)
19. Giesl, J., Thiemann, R., Schneider-Kamp, P.: The dependency pair framework: Combining techniques for automated termination proofs. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 301–331. Springer, Heidelberg (2005)
20. Giesl, J., Thiemann, R., Swiderski, S., Schneider-Kamp, P.: Proving termination by bounded increase. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 443–459. Springer, Heidelberg (2007)
21. Hirokawa, N., Middeldorp, A.: Tyrolean Termination Tool: Techniques and features. IC 205, 474–511 (2007)
22. Hong, H., Jakuš, D.: Testing positiveness of polynomials. JAR 21, 23–38 (1998)
23. Korp, M., Sternagel, C., Zankl, H., Middeldorp, A.: Tyrolean Termination Tool 2. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 295–304. Springer, Heidelberg (2009)

24. Lankford, D.: On proving term rewriting systems are Noetherian. Memo MTP-3, Mathematics Department, Louisiana Tech. University, Ruston (1979)
25. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part I. CACM 3(4), 184–195 (1960)
26. Miné, A.: Weakly Relational Numerical Abstract Domains. PhD thesis, École Polytechnique, Palaiseau, France (2004)
27. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
28. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS 2004, pp. 32–41 (2004)
29. Tiwari, A.: Termination of linear programs. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 70–82. Springer, Heidelberg (2004)
30. Zantema, H.: Termination. In: TeReSe (ed.) Term Rewriting Systems, ch. 6. Cambridge University Press, Cambridge (2003)

Solving Non-linear Polynomial Arithmetic via SAT Modulo Linear Arithmetic*

Cristina Borralleras[1], Salvador Lucas[2], Rafael Navarro-Marset[2], Enric Rodríguez-Carbonell[3], and Albert Rubio[3]

[1] Universidad de Vic, Spain
[2] Universitat Politècnica de València, Spain
[3] Universitat Politècnica de Catalunya, Barcelona, Spain

Abstract. Polynomial constraint-solving plays a prominent role in several areas of engineering and software verification. In particular, polynomial constraint solving has a long and successful history in the development of tools for proving termination of programs. Well-known and very efficient techniques, like SAT algorithms and tools, have been recently proposed and used for implementing polynomial constraint solving algorithms through appropriate encodings. However, powerful techniques like the ones provided by the SMT (SAT modulo theories) approach for *linear arithmetic constraints* (over the rationals) are underexplored to date. In this paper we show that the use of these techniques for developing polynomial constraint solvers outperforms the best existing solvers and provides a new and powerful approach for implementing better and more general solvers for termination provers.

Keywords: Constraint solving, polynomial constraints, SAT modulo theories, termination, program analysis.

1 Introduction

Polynomial (non-linear) constraints are present in many application domains, like program analysis or the analysis of hybrid systems. In particular, polynomial constraint solving has a long and successful history in the development of tools for proving termination of programs and especially in the proof of termination of symbolic programs as well as rewrite systems (see e.g. [5,12,15,18,21]). In this setting, recent works have shown that solving polynomial constraints *over the reals* can improve the power of termination provers [11,19].

In this paper, we are interested in developing non-linear arithmetic constraint solvers that can find solutions over finite subsets of the integers or the reals (although in fact, we restrict ourselves to rational numbers). Even though there

* This work has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN, under grants TIN 2007-68093-C02-01 and TIN 2007-68093-C02-02. Rafael Navarro-Marset was partially supported by the Spanish MEC/MICINN under FPU grant AP2006-026.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 294–305, 2009.

might be other applications, our decisions are guided by the use of our solvers inside state-of-the-art automatic termination provers (e.g., AProVE [13], C*i*ME [4], MU-TERM [17], TTT [16], ...). For this particular application, our solvers must be always sound, i.e. we cannot provide a wrong solution. Completeness, when the solution domain is finite, is also desirable since it shows the strength of the solver. The solvers we present in this paper are all sound and complete for any given finite solution domain.

There have been some recent works providing new implementations of solvers for polynomial constraints that outperform previous existing solvers [5]. These new solvers are implemented by translating the problem into SAT [10,11] or into CSP [19]. Following the success of the translation into SAT, it is reasonable to consider whether there is a better target language than propositional logic to keep as much as possible the arithmetic structure of the source language[1]. In this line we propose a simple method for solving non-linear polynomial constraints over the integers and the reals by considering finite subdomains of integer and real numbers and translating the constraints into *SAT modulo linear (integer or real) arithmetic* [7], i.e. satisfiability of quantifier-free boolean combinations of linear equalities, inequalities and disequalities. An interesting feature of this approach (in contrast to the SAT-based treatment of [10,11]) is that we can handle domains with negative values for free. The use of negative values can be useful in termination proofs as shown in e.g. [9,15,16,18].

The resulting problem can be solved by taking off-the-shelf any of the state-of-the-art solvers for *SAT modulo Theories (SMT)* that handles linear (real or integer) arithmetic (e.g. Barcelogic [3], Yices [6] or Z3 [20]). The efficiency of these tools is one of the keys for the success of our approach. The resulting solvers we obtain are faster and more flexible in handling different solution domains than all their predecessors. To show their performance we have compared our solvers with the ones used inside AProVE, based on SAT translation, and the ones used in MU-TERM, based on CSP. This comparison is really fair since for both AProVE and MU-TERM we have been provided with versions of the systems that can use different polynomial constraint solvers as a black box, giving us the opportunity to check the real performance of our solvers compared to the ones in use.

Linearization has also been considered for (non-linear) pseudo-boolean constraints in the literature[2]. However, this is a simpler case of linearization as it coincides with polynomial constraints over the domain $\{0, 1\}$, where products of variables are always in $\{0, 1\}$ as well.

Although our aim is to provide solvers for termination tools, our constraint solvers have shown to be very effective for relatively small solution domains, performing better, for some domains, than specialized solvers like HySAT [8] based on *interval analysis* using an SMT approach. Therefore, our results may have in

[1] An obvious possibility is the use of the *first-order theory of real closed fields* which was proved decidable by Tarski. In practice, though, this is unfeasible due to the complexity of the related algorithms (see [2] for a recent account).

[2] See `http://www.cril.univ-artois.fr/PB07/coding.html`, the webpage of the Pseudo-Boolean Evaluation 2007.

the future a wider application, as this kind of constraints is also considered for instance in the analysis of hybrid systems [14].

Another interesting feature is that our approach can handle general boolean formulas over non-linear inequalities, which can be very useful for improving the performance of termination tools. For instance, one can delegate to the SAT engine the decisions on the different options that produce a correct termination proof, which is known to be one of the strong points of SAT solvers.

Altogether, we believe that our solvers can have both an immediate and a future impact on termination tools.

The paper is structured as follows. Our method is described in Section 2 and Section 3 is devoted to implementation features and experiments. Some conclusions and future work are given in Section 4.

2 Translation from Non-linear to Linear Constraints

As said, the constraints we have to solve are quantifier-free propositional formulas over non-linear arithmetic. Hence, we move from SAT modulo non-linear arithmetic, for which there are no suitable solvers, to SAT modulo linear arithmetic, for which fast solvers exist. As usual, in the following, by *non-linear arithmetic* we mean polynomial arithmetic not restricted to the linear case.

Definition 1. *A* non-linear monomial *is an expression* $v_1^{p_1} \dots v_m^{p_m}$ *where* $m > 0$ *and* $p_i > 0$ *for all* $i \in \{1 \dots m\}$ *and* $v_i \neq v_j$ *for all* $i, j \in \{1 \dots m\}$, $i \neq j$.

A non-linear arithmetic formula *is a propositional formula, where atoms are of the form*

$$\Sigma_{1 \leq i \leq n}\ c_i \cdot M_i \bowtie k$$

where $\bowtie \in \{=, \geq, >, \leq, <\}$, k *is an integer or a rational number, every* M_i *is a non-linear monomial and every* c_i *is an integer or a rational number.*

In the following, we assume that we have a fresh set of variables $\mathcal{X}_\mathcal{M}$ containing a variable x_M for every monomial M of the form $v_1^{p_1} \cdot \ldots \cdot v_m^{p_m}$ that can be built out of the variables $v_1 \dots v_m$. By $C[c \cdot M]$ we denote that $c \cdot M$ occurs in the constraint C and, by $C[c \cdot x]$ we ambiguously denote that the monomial M has been replaced by x.

Now, we describe two transformations from non-linear to linear constraints. We consider two kinds of domains: integer intervals and finite sets of rational numbers.

2.1 Integer Intervals

First, we consider solution domains consisting of integer intervals, i.e. integers in the domain $\{\mathcal{B}_1, \dots, \mathcal{B}_2\}$ for some lower bound $\mathcal{B}_1$ and upper bound $\mathcal{B}_2$. One particular case that we will often use is when $\mathcal{B}_1$ is 0.

Example 1. Consider the atom:

$$2a^3b - 5cd^2e \geq 0$$

with $\mathcal{B}_1 = 0$ and $\mathcal{B}_2 = 2$. Then, the translation is done by adding variables x_{a^3b}, y_{d^2e} and x_{cd^2e}, which represent $a^3 \cdot b$, $d^2 \cdot e$ and $c \cdot y_{d^2e}$ respectively.

Linearizing, we obtain the following equisatisfiable constraint:

$$
\begin{array}{lll}
2x_{a^3b} - 5x_{cd^2e} \geq 0 & & \\
a = 0 \rightarrow x_{a^3b} = 0 & c = 0 \rightarrow x_{cd^2e} = 0 & d = 0 \rightarrow y_{d^2e} = 0 \\
a = 1 \rightarrow x_{a^3b} = b & c = 1 \rightarrow x_{cd^2e} = y_{d^2e} & d = 1 \rightarrow y_{d^2e} = e \\
a = 2 \rightarrow x_{a^3b} = 8b & c = 2 \rightarrow x_{cd^2e} = 2y_{d^2e} & d = 2 \rightarrow y_{d^2e} = 4e \\
0 \leq a \leq 2 & 0 \leq c \leq 2 \quad 0 \leq d \leq 2 \quad 0 \leq b \leq 2 & 0 \leq e \leq 2
\end{array}
$$

In the following definition, the Abstraction rule linearizes the initial constraint, while the following three rules linearize the equalities introduced by the abstraction. Linearization rules 1 and 2 remove a non-linear equality by adding new linear formulas but without introducing new intermediate variables. In these two rules, making a case analysis on one initial variable is enough to linearize. Finally in rule Linearization 3 one variable of the monomial of some non-linear equality is removed by adding a case analysis and a new equality with a smaller monomial is obtained.

Definition 2. *Let C be a constraint. The transformation rules are the following:*

Abstraction:

$C[c \cdot M] \Longrightarrow C[c \cdot x_M] \wedge x_M = M$ *if M is not linear and c is a constant*

Linearization 1:

$C \wedge x = v_i^{p_i} \Longrightarrow C \wedge \bigwedge_{\alpha=\mathcal{B}_1}^{\mathcal{B}_2}(v_i = \alpha \rightarrow x = \alpha^{p_i}) \wedge \mathcal{B}_1 \leq v_i \leq \mathcal{B}_2$ *if $p_i > 1$*

Linearization 2:

$C \wedge x = v_i^{p_i} \cdot v_j \Longrightarrow C \wedge \bigwedge_{\alpha=\mathcal{B}_1}^{\mathcal{B}_2}(v_i = \alpha \rightarrow x = \alpha^{p_i} \cdot v_j) \wedge \mathcal{B}_1 \leq v_i \leq \mathcal{B}_2 \ \wedge \ \mathcal{B}_1 \leq v_j \leq \mathcal{B}_2$

Linearization 3:

$C \wedge x = v_i^{p_i} \cdot M \Longrightarrow C \wedge \bigwedge_{\alpha=\mathcal{B}_1}^{\mathcal{B}_2}(v_i = \alpha \rightarrow x = \alpha^{p_i} \cdot x_M)$ *if M is not linear* $\wedge \mathcal{B}_1 \leq v_i \leq \mathcal{B}_2 \ \wedge \ x_M = M$

Correctness and complexity. Since the rules above are terminating, we will obtain a normal form after a finite number of steps. By a simple analysis of the rules, we have that a constraint in normal form is linear; moreover, since the left and the right hand sides of every rule are equisatisfiable, any normal form D of an initial constraint C with respect to these rules is a linear constraint and C and D are equisatisfiable. So our transformation provides a sound and complete method for deciding non-linear constraints over integer intervals.

Regarding the size of the resulting formula, let N be $\mathcal{B}_2 - \mathcal{B}_1 + 1$, i.e. the cardinality of the domain, and C be the problem to be linearized. Then the size of the linearized formula is in $O(N \cdot \text{size}(C))$, since, in the worst case, we add N clauses for every variable in every monomial of C.

2.2 Rationals as Integers

In this case, we consider the set of rational numbers $\{\frac{n}{D} \mid \mathcal{B}_1 \leq n \leq \mathcal{B}_2\}$ which are obtained by fixing the denominator D and bounding the numerator. For instance taking $D = 4$ and the numerator in $\{0 \dots 16\}$ we consider all rational numbers in the set $\{\frac{0}{4}, \frac{1}{4}, \dots, \frac{15}{4}, \frac{16}{4}\}$. We denote this domain by $\mathcal{B}_1..\mathcal{B}_2/D$.

Then, we simply replace any variable x by $\frac{x'}{D}$ for some fresh variable x' and eliminate denominators from the resulting constraint by multiplying as many times as needed by D. As a result we obtain a constraint over the integers to be solved in the integer interval domain of the numerator. It is straightforward to show completeness of the transformation.

This translation turns out to be reasonably effective. The performance is similar to the integer interval case, but depending on the size of D it works worse due to the fact that the involved integer numbers become much larger.

We have also considered more general domains like the rational numbers expressed by $k + \frac{n}{D}$ with $0 \leq n < D$ for a bounded k and a fixed D that can also be transformed into constraints over bounded integers. Our experiments revealed a bad trade-off between the gain in the expressiveness of the domain and the performance of the SMT solver on the resulting constraints.

Similarly, we have studied domains of rational values of the form $\frac{n}{d}$ with a bounded numerator n and a bounded denominator d. In this case, in order to solve the problem over the integers, every variable a is replaced by $\frac{n_a}{d_a}$ where n_a and d_a are fresh integer variables. Due to the increase of the complexity of the monomials after the elimination of denominators, there is an explosion in the number of intermediate variables needed to linearize the constraint, which finally cause a very poor performance of the linear arithmetic solver. This kind of domains was also considered, with the same conclusion, in [11].

2.3 Finite Rational Domains

Now, we consider that the solution domain is a finite subset Q of the rational numbers. The only difference with respect to the approach in Section 2.1 is that the domain of the variables is described by a disjunction of equality literals.

Definition 3. *Let C be a constraint. The transformation rules are the following:*

Abstraction:

$C[c \cdot M] \Longrightarrow C[c \cdot x_M] \wedge x_M = M$ *if M is not linear and c is a constant*

Linearization 1:

$C \wedge x = v_i^{p_i} \Longrightarrow C \wedge \bigwedge_{\alpha \in Q}(v_i = \alpha \rightarrow x = \alpha^{p_i}) \wedge \bigvee_{\alpha \in Q} v_i = \alpha$ *if $p_i > 1$*

Linearization 2:

$C \wedge x = v_i^{p_i} \cdot v_j \Longrightarrow C \wedge \bigwedge_{\alpha \in Q}(v_i = \alpha \rightarrow x = \alpha^{p_i} \cdot v_j) \wedge \bigvee_{\alpha \in Q} v_i = \alpha \ \wedge \bigvee_{\alpha \in Q} v_j = \alpha$

Linearization 3:

$C \wedge x = x_i^{p_i} \cdot M \Longrightarrow C \wedge \bigwedge_{\alpha \in Q}(v_i = \alpha \rightarrow x = \alpha^{p_i} \cdot x_M) \wedge \bigvee_{\alpha \in Q} v_i = \alpha \ \wedge \ x_M = M$ *if M is not linear*

Example 2. Consider the atom: $3abc - 4cd + 2a \geq 0$
with $Q = \{0, \frac{1}{2}, 1, 2\}$ we have the following equisatisfiable linear constraint:

$$3x_{abc} - 4x_{cd} + 2a \geq 0$$

$a = 0 \rightarrow x_{abc} = 0$	$b = 0 \rightarrow y_{bc} = 0$	$c = 0 \rightarrow x_{cd} = 0$
$a = \frac{1}{2} \rightarrow 2x_{abc} = y_{bc}$	$b = \frac{1}{2} \rightarrow 2y_{bc} = c$	$c = \frac{1}{2} \rightarrow 2x_{cd} = d$
$a = 1 \rightarrow x_{abc} = y_{bc}$	$b = 1 \rightarrow y_{bc} = c$	$c = 1 \rightarrow x_{cd} = d$
$a = 2 \rightarrow x_{abc} = 2y_{bc}$	$b = 2 \rightarrow y_{bc} = 2c$	$c = 2 \rightarrow x_{cd} = 2d$

$$(a = 0 \vee a = \tfrac{1}{2} \vee a = 1 \vee a = 2) \quad (b = 0 \vee b = \tfrac{1}{2} \vee b = 1 \vee b = 2)$$
$$(c = 0 \vee c = \tfrac{1}{2} \vee c = 1 \vee c = 2) \quad (d = 0 \vee d = \tfrac{1}{2} \vee d = 1 \vee d = 2)$$

3 Implementation Features and Experiments

In this section, we present some implementation decisions we have taken when implementing the general transformation rules given in previous sections that are relevant for the performance of the SMT solver on the final formula.

3.1 Choosing Variables for Simplification

In every step of our transformation, some variable corresponding to a non-linear expression, e.g. x_{ab^2c}, is treated by choosing one of the original variables in its expression, in this case a, b or c, and writing the value of the variable depending on the different values of the original variable and the appropriate intermediate variable.

Both decisions, namely which non-linear expression we handle first and which original variable we take, have an important impact on the final solution. The number of intermediate variables and thus the number of clauses in the final formula are highly dependent on these decisions. Not surprisingly, in general, the performance is improved when the number of intermediate variables is reduced. A similar notion of intermediate variables (only representing products of two variables), called product and square variables, and heuristics for choosing them are also considered in [5].

Let us now formalize the problem of finding a *minimal* (wrt. cardinality) set of intermediate variables for linearizing the initial constraint.

In this section, a non-linear monomial $v_1^{k_1} \dots v_p^{k_p}$, where all v_i are assumed to be different, is represented by a set of pairs $M = \{(v_1, k_1) \dots (v_p, k_p)\}$.

Now, we can define a *closed set of non-linear monomials* C for a given initial set C_0 of non-linear monomials, as a set fulfilling $C_0 \subseteq C$ and for every $M_i \in C$ we have that either

- $M_i = \{(v_1, k_1)\}$, or
- $M_i = \{(v_1, k_1), (v_2, k_2)\}$ with $k_1 = 1$ or $k_2 = 1$, or
- there exists $M_j \in C$ such that either:
  - there is $(v, k) \in M_i$ and $(v, k') \in M_j$ with $k > k'$ such that $M_i \setminus \{(v, k)\} = M_j \setminus \{(v, k')\}$, or
  - there is $(v, k) \in M_i$ such that $M_i \setminus \{(v, k)\} = M_j$.

Lemma 1. *Let C_0 be the set of monomials of a non-linear formula F. If C is a closed set of C_0 then we can linearize F using the set of variables $\{x_M \mid M \in C\}$.*

Example 3. Let C_0 be the set of non-linear monomials $\{\ ab^2c^3d\ ,\ a^2b^2\ ,\ bde\ \}$. A closed set of non-linear monomials required to linearize C_0 could be $C = \{\ ab^2c^3d,\ a^2b^2,\ bde,\ ab^2d,\ ab^2,\ de\ \}$.

As said, we are interested in finding a minimal closed set C for C_0. The decision version of this problem can be shown to be NP-complete. Thus finding a minimal set of monomials can be, in general, too expensive as a subproblem of our transformation algorithm. For this reason, we have implemented a greedy algorithm that provides an approximation to this minimal solution.

Our experiments have shown that applying a reduction of intermediate variables produces a linear constraint that is, in general, easier to be checked by the SMT solver. However, this impact is more important when considering integer interval domains than when considering domains with rationals which are expressed by a set of particular elements. In any case, further analysis on the many different ways to implement efficient algorithms approximating the minimal solution is still necessary.

3.2 Bounds for Intermediate Variables

As it usually happens in SAT translations, adding redundancy may help the solver. In our case, we have observed that, in general, to improve the performance of the solver it is convenient to add upper and lower bound constraints to all intermediate variables. For instance, if we have a solution domain $\{0 \dots \mathcal{B}\}$ and a monomial abc then for the variable x_{abc} the constraint $0 \leq x_{abc} \leq \mathcal{B}^3$ is added.

Our experiments on integer intervals of the form $\{0 \dots \mathcal{B}\}$ have shown that adding bounds for the intermediate variables has, in general, a positive impact. This is not that clear when expressing domains as disjunctions of values.

3.3 Choosing Domains

In our particular application to termination of rewriting, it turned out that having small domains suffices in general. For instance, when dealing with domains of non-negative integer coefficients, only one more example can be proved by considering an upper bound 7 instead of 4, and no more examples are proved for the given time limit even considering bound 15 or 31. On the other hand, increasing the bound increases the number of timeouts, losing some positive answers as well. However in all cases, our experiments showed that our solver can be used, with a reasonable performance, even with not so small bounds.

In the case of using rational solutions, again small domains are better, but in this case making a right choice is a bit trickier, since there are many possibilities. As a starting point, we have considered domains that can be handled by at least one of the original solvers included in the tools we are using for our experiments. Again, in all considered domains, our solver showed a very good performance, behaving better in time and number of solved examples.

For rationals we have considered several domains, but the best results were obtained with the domain called $Q4$, which includes $\{0, 1, 2, 4, \frac{1}{2}, \frac{1}{4}\}$, and its extension with the value 8, called $Q4 + 8$. Another interesting domain that is treated with rationals as integers is the already mentioned $0..16/4$. The domains $Q4$ and $Q4 + 8$ cannot be handled by the version of the AProVE we have, while the domain $0..16/4$ and $Q4 + 8$ cannot be handled by the MU-TERM solver. We report our experimental results on these domains in Sections 3.5 and 3.6.

3.4 Comparison with Existing Solvers

We provide a comparison of our solver with the solvers used inside AProVE and MU-TERM. In order to do that, we have used a parameterized version of both that can use different solvers provided by the user. In this way, we have been able to test the performance and the impact of using our solvers instead of the solvers that both systems are currently using.

As SMT solver for linear arithmetic, we use Yices 1.0.16, since it has shown the best performance on the formulas we are producing. We have also tried Barcelogic and Z3.

For our experiments we have considered the benchmarks included in the Termination Problems Data Base (TPDB; `www.lri.fr/~marche/tpdb/`), version 5.0, in the category of term rewriting systems (TRS). For the experiments using AProVE, we have removed all examples that use special kinds of rewriting (basically, rewriting modulo an equational theory and conditional, relative and context sensitive rewriting) since they cannot be handled by the simplified version of AProVE we are using. We have performed experiments on a 2GHz 2GB Intel Core Duo with a time limit of 60 seconds. Detailed information about the experiments and our solver can be found in `www.lsi.upc.edu/~albert/nonlinear.html`.

The tables we have included split, for every experiment, the results (in number of problems and total running time in seconds) depending on whether the answer is YES (then we have a termination proof), MAYBE (we cannot prove termination) or KILLED (we have exceeded the time limit).

3.5 Experiments Using MU-TERM

We have used MU-TERM to generate the polynomial constraints which are sent to the parameterized solver. The symbolic constraints for each termination problem are generated according to the Dependency Pairs technique [1] and using polynomial interpretations over the integers and the rationals.

For integers we have tried the domain $N4$ which includes $\{0, 1, 2, 4\}$ using our SMT-based solver (with disjunction of values) and the CSP-based solver of MU-TERM. To show that even when enlarging the domain, using integer intervals instead of disjunction of values, our solver is faster, we have also considered the integer interval domain $B4 = N4 \cup \{3\}$, for which the same number of examples are solved in less time. For rationals, we have considered the domain $Q4$.

As can be seen in Figure 1, the results are far better using our solver than using the CSP solver. In fact, with the domain $Q4$, there are 142 new problems proved terminating and about 700 less problems killed.

	CSP N4		SMT N4		SMT B4		CSP Q4		SMT Q4	
	Total	Time	Total	Time	Total	Time	Total	Time	Total	Time
YES	691	765	783	749	783	700	717	1118	873	1142
MAYBE	736	3892	1131	2483	1129	2691	437	2268	1000	3618
KILLED	559		72		74		832		113	

Fig. 1. Experimental results with MU-TERM

3.6 Experiments Using AProVE

We have been provided with a *simplified* version of the AProVE tool, which is parameterized by the solver. The system is also based on the dependency pair method and generates polynomial constraints over the integers and the rational numbers. As before, the constraints are sent to the solver which in turn returns a solution if one is found.

	BOUND 1						BOUND 2					
	SAT		HySAT		SMT		SAT		HySAT		SMT	
	Total	Time	Total	Time	Total	Time	Total	Time	Total	Time	Total	Time
YES	649	864	649	783	649	790	701	1360	697	1039	701	1191
MAYBE	1093	2289	1091	2259	1093	2233	1039	2449	1033	2818	1033	2225
KILLED	13		15		13		15		25		21	

	BOUND 4						BOUND 7					
	SAT		HySAT		SMT		SAT		HySAT		SMT	
	Total	Time	Total	Time	Total	Time	Total	Time	Total	Time	Total	Time
YES	705	1596	705	1446	710	1222	705	1762	702	1246	708	1357
MAYBE	1014	3551	967	4061	1019	2481	989	4224	871	4482	1008	2862
KILLED	36		83		26		61		182		39	

Fig. 2. Experiments with integers in AProVE

In order to check not only the performance of our solver compared to the AProVE original one, but also with some other existing solver for non-linear arithmetic, we have included, in the case of integer domains, a comparison with HySAT, a solver for non-linear arithmetic based on interval analysis. Although HySAT can give, in general, wrong solutions (as said in its own documentation), it has never happened to us when considering integers. Moreover, in the case of integers, the comparison is completely fair since we send to HySAT exactly the same constraint provided by AProVE adding only the declaration of the variables with the integer interval under consideration (for instance, `int [0,4] x`). We have included here the results using HySAT since all three solvers can handle the same domains.

The results in Figure 2 show that, in general, HySAT is a little faster when the answer is YES, but has a lot more KILLED problems. On the other hand, our solver is in general faster than the SAT-based AProVE solver and always faster in the overall runtime (without counting timeouts), starting very similar

and increasing as the domain grows. This improvement is more significant if we take into account that there is an important part of the process that is common (namely the generation of constraints) independently of the solver. Moreover, the number of KILLED problems by our solver is only bigger than or equal to that of the SAT-based one for the smallest two domains but is always the lowest from that point on and the difference grows as the domain is enlarged.

0..16/4	SAT		HySAT		SMT	
	Total	Time	Total	Time	Total	Time
YES	797	3091	774	2088	824	2051
MAYBE	666	6605	508	3078	826	4380
KILLED	292		473		105	

SMT	Q4		Q4+8	
	Total	Time	Total	Time
YES	833	1755	834	1867
MAYBE	876	2861	870	2910
KILLED	46		51	

Fig. 3. Experiments with rationals in AProVE

Regarding the use of rationals, we can only compare the performance on domains of rationals as integers with fixed denominator and a bounded numerator, since this is the only case the original solver of our version of AProVE can handle (more general domains are available in the full version of AProVE, but they turned out to have a poorer performance). In particular, we have considered the domain 0..16/4. Additionally, we have included the results of AProVE when using our solver on the domain $Q4$ and $Q4+8$ since these are the domains which overall produce the best results in number of YES.

As for integer intervals, Figure 3 shows that our solver has a better performance, when comparable, than the other two solvers when using rationals. Moreover, we obtain the best overall results when using our solver with rational domains that are not handled by any other solver.

4 Conclusions

We have proposed a simple method for solving non-linear polynomial constraints over finite domains of the integer and the rational numbers, which is based on translating the constraints into *SAT modulo linear (real or integer) arithmetic*. Our method can handle general boolean formulas and domains with negative values for free, making them available for future improvements in termination tools. By means of several experiments, our solvers are shown to be faster and more flexible in handling different solution domains than all their predecessors. Altogether, we believe that these results can have both an immediate and a future impact on termination tools.

As future work, we want to analyze the usefulness of some features that the SMT solvers usually have, like being *incremental* or *backtrackable*, to avoid repeated work when proving termination of a TRS.

Acknowledgments. We would like to thank Jürgen Giesl, Carsten Fuhs and Karsten Behrmann for their very quick reaction in providing us with a version of AProVE to be used in our experiments, and especially to Carsten for some interesting and helpful discussion on possible useful extensions of the solver.

References

1. Arts, T., Giesl, J.: Termination of Term Rewriting Using Dependency Pairs. Theoretical Computer Science 236, 133–178 (2000)
2. Basu, S., Pollack, R., Roy, M.-F.: Algorithms in Real Algebraic Geometry. Springer, Heidelberg (2003)
3. Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonella, E., Rubio, A.: The Barcelogic SMT Solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 294–298. Springer, Heidelberg (2008)
4. Contejean, E., Marché, C., Monate, B., Urbain, X.: Proving termination of rewriting with CiME. In: Proc. of 6th Int. Workshop on Termination (2003)
5. Contejean, E., Marché, C., Tomás, A.-P., Urbain, X.: Mechanically proving termination using polynomial interpretations. Journal of Automated Reasoning 34(4), 325–363 (2006)
6. Dutertre, B., de Moura, L.: The Yices SMT solver. System description report, `http://yices.csl.sri.com/`
7. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
8. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure. Journal on Satisfiability, Boolean Modeling and Computation 1, 209–236 (2007)
9. Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: Maximal termination. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 110–125. Springer, Heidelberg (2008)
10. Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: SAT Solving for Termination Analysis with Polynomial Interpretations. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 340–354. Springer, Heidelberg (2007)
11. Fuhs, C., Navarro-Marset, R., Otto, C., Giesl, J., Lucas, S., Schneider-Kamp, P.: Search Techniques for Rational Polynomial Orders. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) AISC 2008, Calculemus 2008, and MKM 2008. LNCS (LNAI), vol. 5144, pp. 109–124. Springer, Heidelberg (2008)
12. Giesl, J.: Generating Polynomial Orderings for Termination Proofs. In: Hsiang, J. (ed.) RTA 1995. LNCS, vol. 914. Springer, Heidelberg (1995)
13. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS, vol. 4130, pp. 281–286. Springer, Heidelberg (2006)
14. Gulwani, S., Tiwari, A.: Constraint-Based Approach for Analysis of Hybrid Systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 190–203. Springer, Heidelberg (2008)
15. Hirokawa, N., Middeldorp, A.: Polynomial Interpretations with Negative Coefficients. In: Buchberger, B., Campbell, J. (eds.) AISC 2004. LNCS (LNAI), vol. 3249, pp. 185–198. Springer, Heidelberg (2004)
16. Hirokawa, N., Middeldorp, A.: Tyrolean termination tool: Techniques and features. Information and Computation 205, 474–511 (2007)
17. Lucas, S.: MU-TERM: A Tool for Proving Termination of Context-Sensitive Rewriting. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 200–209. Springer, Heidelberg (2004), `http://zenon.dsic.upv.es/muterm`

18. Lucas, S.: Polynomials over the reals in proofs of termination: from theory to practice. RAIRO Theoretical Informatics and Applications 39(3), 547–586 (2005)
19. Lucas, S.: Practical use of polynomials over the reals in proofs of termination. In: Proc. of 9th PPDP. ACM Press, New York (2007)
20. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
21. Nguyen, M.T., De Schreye, D.: Polynomial Interpretations as a Basis for Termination Analysis of Logic Programs. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 311–325. Springer, Heidelberg (2005)

Building Theorem Provers

Mark E. Stickel

Artificial Intelligence Center, SRI International, Menlo Park, California
stickel@ai.sri.com

Abstract. This talk discusses some of the challenges of building a usable theorem prover. These include the chasm between theory and code, conflicting requirements, feature interaction, and competitive performance. The talk draws on the speaker's experiences with devising extensions of resolution and building theorem provers that have been used as embedded reasoners in various systems.

1 The British Museum Algorithm Reconsidered

One of the first automated theorem provers, Newell, Shaw, and Simon's Logic Theory Machine (LT) [22], proved theorems of propositional calculus in *Principia Mathematica*. Its heuristic approach was contrasted with exhaustive search referred to as "The British Museum Algorithm" in a suggestion of how *not* to prove theorems:

> The algorithm constructs all possible proofs in a systematic manner, checking each time (1) to eliminate duplicates, and (2) to see if the final theorem in the proof coincides with the expression to be proved. With this algorithm the set of one-step proofs is identical with the set of axioms (i.e., each axiom is a one-step proof of itself). The set of n step proofs is obtained from the set of $(n-1)$-step proofs by making all permissable substitutions and replacements in the expressions of the $(n-1)$-step proofs, and by making all the permissable detachments of pairs of expressions as permitted by the recursive definitions of proof.

A disadvantage of such exhaustive search is the large search space, but it can be more useful in practice than generally recognized, as I will show.

A similar contemporary research enterprise is the use of automated theorem provers to find original or better proofs in various propositional calculi [21, 45]. Many such proofs have been found using the *condensed detachment* rule of inference:

$$\frac{(\alpha \rightarrow \beta) \quad \gamma}{\beta\sigma}$$

where σ is the most general unifier of α and γ. This improves on the style of proof in *Principia Mathematica* reproduced by LT. The separate operations of substitution and detachment are combined into condensed detachment that uses unification to create the most general result of detachment of substitution instances of its premises (eliminating a need in LT for heuristics to create instances).

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 306–321, 2009.

A typical problem is to prove that the formula **BCI**-Candidate 42: $p \to ((q \to r) \to (((s \to s) \to (t \to (p \to q))) \to (t \to r)))$ is a single axiom of **BCI** logic, which is defined by the axioms

B: $(p \to q) \to ((r \to p) \to (r \to q))$
C: $(p \to (q \to r)) \to (q \to (p \to r))$
I: $p \to p$.

Dolph Ulrich identified the formula as a candidate for being a shortest single axiom of **BCI** logic after eliminating the possibility of shorter single axioms and proving that it is a theorem of **BCI** logic. Whether **B**, **C**, and **I** (or a known single axiom) are provable from **BCI**-Candidate 42 by condensed detachment was an open question.

This type of problem has often been solved with Otter [20], but this time experiments by Wos led him to suggest it is not a single axiom. He writes in [43]

> It is, in fact a theorem of **BCI**, but so far I have been unable to show that it is strong enough to be a single axiom. I doubt it is, my suspicion being based on the fact that, as it appears, all deducible theorems from the formula contain an alphabetic variant of $p \to p$.

In 2003, I wrote a function `coder` (condensed detacher) in Snark [35] to exhaustively generate condensed detachment derivations in order of increasing length, like the denigrated British Museum Algorithm. Two problems with breadth-first search are the time needed to generate results and the memory needed to store them. Depth-first iterative-deepening search [34, 12] eliminates the latter problem by storing only one derivation at a time with alternatives being tried by backtracking. It simulates breadth-first search by doing a sequence of bounded depth-first searches: first it exhaustively generates all derivations with one condensed detachment step, then all derivations with two condensed detachment steps, etc. The first proof found will be a shortest one, computer memory needs are limited by the size of a single derivation, and recomputation of earlier levels of search is a fraction of the total search cost (since search space size grows exponentially by level).

There are up to $n!^2$ condensed detachment derivations of length n from a single axiom. This can be reduced substantially by rejecting derivations in which

- the latest formula is the same as or an instance of an earlier formula in the derivation (forward subsumption).
- the latest formula is a generalization of an earlier formula in the derivation, unless the earlier formula is used to derive the latest (backward subsumption).
- not all formulas are used in the final derivation. Backtrack if the number of so far unused formulas is more than can be used as premises in the remaining steps of the length-bounded derivation.
- steps appear in other than a single standard order. Use a total term ordering to compare justifications of latest and next to latest steps.

The search space remains huge, but the worst-case number of 9-step derivations, when every condensed detachment succeeds and subsumption never does, is reduced from $n!^2 = 131,681,894,400$ to $97,608,831$.[1]

Returning to the question of whether **BCI**-Candidate 42 is a single axiom of **BCI** logic, `coder` discovered that every formula derivable in 8 or fewer condensed detachment steps properly contains an alphabetic variant of $p \to p$, but there is a 9-step proof of $(((p \to q) \to q) \to r) \to (p \to r)$ (whose largest formula contains 135 symbols and 34 distinct variables) and a 9-step proof of **I** (whose largest formula contains 167 symbols and 42 variables).

Proofs with such large formulas are often difficult for OTTER to find. Its default search strategy uses formula size to order inferences. This is complete in principle but incomplete in practice because large formulas are discarded to conserve memory. Ordering by size is a technique so pervasive and successful that one is liable to forget that it is a heuristic that may lead away from a short proof, as it did here. Still, it is slightly mysterious why it works as well as it does. It is fortunate that formulas that are too large to store are also rarely necessary to find a proof. Why are large formulas so relatively useless?

To complete the proof that **BCI**-Candidate 42 is a single axiom, **B** and **C** are each proved from it and $(((p \to q) \to q) \to r) \to (p \to r)$ in 7 steps. The following open questions by Dolph Ulrich of whether formulas are single axioms of **BCI** logic or **BCK** logic, which is defined by **B**, **C**, and **K**: $p \to (q \to p)$, have all been answered affirmatively by `coder`:

BCI-Cand. 19: $(p \to q) \to (((((r \to r) \to (s \to p)) \to q) \to t) \to (s \to t))$
BCI-Cand. 42: $p \to ((q \to r) \to (((s \to s) \to (t \to (p \to q))) \to (t \to r)))$
BCI-Cand. 48: $((((p \to p) \to (q \to r)) \to s) \to t) \to ((r \to s) \to (q \to t))$
BCK-Cand. 3: $(p \to ((q \to r) \to s)) \to (r \to ((t \to p) \to (t \to s)))$
BCK-Cand. 7: $p \to ((q \to ((r \to p) \to s)) \to ((t \to q) \to (t \to s)))$
BCK-Cand. 8: $p \to ((q \to (p \to r)) \to (((s \to t) \to q) \to (t \to r)))$
BCK-Cand. 12: $(p \to (q \to r)) \to (q \to (((s \to t) \to p) \to (t \to r)))$

Although I expect the British Museum Algorithm to often fail to prove theorems in a reasonable amount of time, these successes illustrate its value in the "Arsenal of Weapons" [42] for theorem proving.

Theorem provers are usually incomplete in practice. Proofs, if they exist, may not be found for various reasons such as time limits, memory limits, or incomplete combinations of restrictions and strategies. `coder`'s completeness is usually limited only by time. A theorem prover with more options and heuristics may succeed more often, but when it fails, more causes of incompleteness can leave the user in a quandary about what to change to improve chances for success. With `coder`, only more—maybe impossibly more—patience is required.

[1] Newell, Shaw, and Simon's best guess of how many proofs would have to be generated to include proofs of all theorems of Chapter 2 of *Principia Mathematica* might be one hundred million. This is a much less formidable number now after fifty years of progress in computer technology.

2 PTTP, Another Simple Complete Theorem Prover

The Prolog Technology Theorem Prover (PTTP) offers the same promise of completeness as SNARK's `coder`, but is not specialized to condensed detachment. PTTP is goal-oriented, and it is capable of a very high inference rate. `coder` enumerates forward-reasoning derivations in order of length by depth-first iterative-deepening search. PTTP does much the same except the derivations are incomplete backward-reasoning ones.

Prolog is a logic programming language that has the standard theorem-proving functions of resolution and unification, in restricted form, as its basic operations. The high inference rate of Prolog systems makes it worthwhile to examine how Prolog implementation techniques can be used to improve the performance of automated theorem provers.

Prolog has three huge deficiencies as a theorem prover:

- its unification algorithm is unsound and possibly nonterminating.
- its search strategy is incomplete.
- its inference system is incomplete for non-Horn clauses.

PTTP [32] represents a particular approach to preserving as much as possible the character and performance of an implementation of a Prolog system while overcoming these deficiencies.

A simple and effective means of doing this is to transform clauses of theorem-proving problems to executable Prolog clauses [33]. Three transformations are used:

- A transformation for sound unification renames variables in clause heads, linearizing them, so that the "occur check" problem cannot occur while unifying a clause head and a goal. Remaining unification steps are placed in the body of the clause where they are performed by a built-in predicate that does sound unification with the occur check.
- A transformation for complete depth-bounded search adds depth-bound arguments to each predicate and depth-bound test and decrement operations to the bodies of nonunit clauses. A driver predicate can then be used to control depth-first iterative-deepening search.
- A transformation for complete model elimination (ME) inference adds an argument for the list of ancestor goals to each predicate and adds ancestor-list update operations to the bodies of nonunit clauses; clauses are added to perform ME reduction and pruning operations. Prolog inference plus the ME reduction operation, which succeeds if a goal is unifiable with the complement of an ancestor goal, is complete for non-Horn clauses.

PTTP is basically Loveland's model elimination (ME) theorem-proving procedure [15, 16] implemented in the same manner as Prolog.

PTTP is one of the fastest theorem provers in existence when evaluated by its inference rate and performance on easy problems. At SRI, it was used to solve reasoning problems in robot planning and natural-language-understanding

systems. Amir and Maynard-Zhang used PTTP as the embedded reasoner for a logic-based subsumption architecture [1].

Wos et al. [44] argue persuasively that subsumption to control redundancy in the search space is indispensable when solving hard problems, thus making PTTP generally unsuitable, because Prolog and PTTP lack subsumption and their implementations largely preclude it. Subsumption requires storage of derived clauses and comparison of old and new derived clauses to see if one subsumes another. Derived clauses are not stored in Prolog and PTTP. Instead, they are implicitly represented, one at a time on the stack, and are destroyed on backtracking. The absence of subsumption is the result of a deliberate trade-off between high speed inference and architectural simplicity versus greater control of redundancy and more complex implementation but it makes it difficult for PTTP to solve hard problems.[2]

An advantage of PTTP is that the ME procedure is in effect a highly restricted form of resolution that is compatible with the set of support strategy:[3] it is goal-oriented.

As with Prolog, PTTP's goal-oriented backward chaining from the theorem to be proved is important in applications in which there are many irrelevant axioms, such as planning, natural-language understanding, and deductive databases. The advantage of being goal-oriented can be overstated though. While pure forward reasoning may generate many irrelevant but true conclusions, pure backward reasoning may generate many unprovable but relevant subgoals. Backward reasoning is as blind to the premises as forward reasoning is to the goal.

Models could conceivably be used to eliminate unprovable goals. Not naively, however, because proving C by PTTP from $\neg A \vee C$, $\neg B \vee C$, $A \vee B$, $\neg C \vee A$, $\neg C \vee B$, which has model $\{A, B, C\}$, requires expansion of subgoal $\neg A$ or $\neg B$, which are false in the model.

The theorem proving methods of `coder` and PTTP are particularly amenable to being implemented in a functional programming language for parallel execution. This is increasingly important as recent microprocessor development has focused more on increasing the number of cores than on speed.

3 Building Applications with Theorem Provers

Much of my work in building theorem provers has involved their use in applications for robot planning, natural-language understanding, question answering, etc. For this, the single inference method approach of a PTTP is not always

[2] It can be difficult to anticipate what will be hard for a particular theorem prover. One of Wos el al.'s challenges for theorem provers without subsumption, Theorem 4, *is* provable by PTTP. The OTTER proof took 29,486 seconds on a Sun 3. André Marien was first to use PTTP to prove it on a Sun 3/50 in 14,351 seconds in 1989. PTTP now takes 193 seconds on a 2.8 GHz Mac Pro.

[3] The model elimination procedure (or linear resolution) arguably provides an answer to Basic Research Problem #1, extending the set of support strategy, in [41] but is unsatisfyingly incompatible with the usual subsumption.

enough; it lacks equality reasoning, for example. Resolution and paramodulation calculi provide more options to enable solution of difficult problems. My and my collaborators' (notably Richard Waldinger) attitude is that difficult problems will often not be immediately solvable with acceptable performance, but will instead require customizing the axioms and options to the problem and perhaps even extending the theorem prover. It is hoped that this customization and extension generalizes well to other problems in the same domain so that the axioms, options, and theorem prover combine into an effective domain-specific reasoner.

SNARK is a theorem prover meant to support this sort of activity. Taken as a whole, the applications can be regarded as hybrid reasoning systems because SNARK often uses procedural attachment and other features instead of axioms to rewrite terms or resolve away literals. For example, a proof may include calls on a geographic database or a biological computation tool. SNARK is written in Lisp, which is not only a superb language for writing symbolic applications, but is well suited as an extension language for procedural attachment code and as a scripting language for controlling SNARK and allows for easy embedding in (Lisp-based) applications. SNARK uses standard Lisp representations for constant (symbol, number, and string) and list terms, which eliminates the need for translation to and from a nonstandard internal representation across the procedural attachment boundary.

SNARK has been applied most often to deductive question answering, which is described in the next section adapted from [39]. The following sections mostly relate to extensions or their challenges and complications.

4 Deductive Question Answering

Deductive question answering, the extraction of answers to questions from machine-discovered proofs, is the poor cousin of program synthesis. It involves much of the same technology—theorem proving and answer extraction—but the bar is lower. Instead of constructing a general program to meet a given specification for any input—the program synthesis problem—we need only construct answers for specific inputs; question answering is a special case of program synthesis. Since the input is known, there is less emphasis on case analysis (to construct conditional programs) and mathematical induction (to construct looping constructs), those bugbears of theorem proving that are central to general program synthesis.

Slagle's DEDUCOM [28] obtained answers from resolution proofs; knowledge was encoded in a knowledge base of logical axioms (the subject domain theory), the question was treated as a conjecture, a theorem prover attempted to prove that the conjecture followed from the axioms of the theory, and an answer to the question was extracted from the proof. The answer-extraction method was based on keeping track of how existentially quantied variables in the conjecture were instantiated in the course of the proof.

The QA3 program [11] of Green, Yates, and Raphael integrated answer extraction with theorem proving via the answer literal. Chang and Lee [5] and Manna

and Waldinger [18] introduced improved methods for conditional and looping answer construction. Answer extraction became a standard feature of automated resolution theorem provers, such as OTTER and SNARK, and was also the basis for logic programming systems, in which a special-purpose theorem prover served as an interpreter for programs encoded as axioms.

The AMPHION system [17, 36] for answering questions posed by NASA planetary astronomers computed an answer by extracting from a SNARK proof a straight-line program composed of procedures from a subroutine library; because the program contained no conditionals and no loops, it was possible for AMPHION to construct programs that were dozens of instructions long, completely automatically. Software composed by AMPHION has been used for the planning of photography in the Cassini mission to Saturn.

While traditional question-answering systems stored all their knowledge as axioms in a formal language, this proves impractical when answers depend on large, constantly changing external data sources; a procedural-attachment mechanism allows external data and software sources to be consulted by a theorem prover while the proof is underway. As a consequence, relatively little information needs to be encoded in the subject domain theory; it can be acquired if and when needed. While external sources may not adhere to any standard representational conventions, procedural attachment allows the theorem prover to invoke software sources that translate data in the form produced by one source into that required by another. Procedural attachment is particularly applicable to Semantic Web applications, in which some of the external sources are Web sites, whose capabilities can be advertised by axioms in the subject domain theory.

SRI employed a natural-language front end and SNARK equipped with procedural attachment to answer questions posed by an intelligence analyst (QUARK) or an Earth systems scientist (GEOLOGICA) [38].

Recent efforts include BIODEDUCTA for deductive question answering in molecular biology [27]. Questions expressed in logical form are treated as conjectures and proved by SNARK from a biological subject domain theory; access to multiple biological data and software resources is provided by procedural attachment.

SNARK produces a detailed refutation proof as well as the answer. Proofs can be a source for precise explanations and justifications for the answers that could be invaluable for assuring their correctness and documenting how they were obtained.

5 Goal-Orientedness, an Elusive Goal

A critical issue for these application is that the reasoning must be substantially goal-oriented to be feasible. This continues to be the largest single unsolved problem in using SNARK in these applications. The difficulty arises from incompatibilities between goal-orientedness and preferred reasoning methods.

Equality reasoning using term orderings is too effective to ignore, but it is incompatible with the set of support restriction. Relaxed or lazy paramodulation [29] theoretically provides a solution to this problem and for tableau provers

like PTTP as well [23]. How effective these methods are in practice has yet to be determined.

Constraint propagation algorithms, used in SNARK for temporal and spatial reasoning, also lack goal-orientedness.

6 Indexing, a Necessary Evil

Indexing [26, 10] is expensive in time and space but is indispensable for high performance resolution theorem provers.[4] Indexing is also costly because performance may depend on using more than one type of indexing. For example, SNARK uses both discrimination tree indexing, which is fast for retrieving generalizations of terms during forward subsumption and rewriting, and path indexing, which is fast for retrieving instances of terms during backward subsumption and rewriting. This increases memory use, index insertion and deletion time, and implementation effort.

But there is a conflict between performance and extensibility. Indexing is a huge impediment to building theories into theorem provers. For example, adding associative-commutative (AC) unification requires extension of indexing. A quick and dirty solution, used in SNARK, is to abbreviate terms headed by an AC function f by just f for indexing. However, this does not work for building in equations with different head symbols.

Building in theories that interpret relation symbols as in theory resolution and chain resolution [31, 7, 37] requires similar accommodation by indexing: pairs of literals with different relation symbols and with the same or different polarity may resolve or subsume.

7 Procedural Attachment, Get Someone Else to Do It

Procedural attachment is an essential ingredient of deductive question answering applications of SNARK. Lisp code can be attached to function or relation symbols. During rewriting and resolution operations, this code can be used to compute values and substitutions in place of equality rewrites and complementary literals in other clauses.

Procedural attachment is often used to rewrite terms or atoms: code for the `$$sum` function will rewrite `($$sum 1 1)` to `2` and code for the equality relation will rewrite `(= a a)` to `true` and `(= 1 2)` to `false`.[5]

This mechanism is easy to implement and use, and often works as well as hoped for, but it is a form of deliberately incomplete reasoning. Ordinary term

[4] Tableau provers like PTTP may be able to perform well without much indexing. The number of formulas available for inference is limited by a hopefully small number of input formulas and shallow branches in the tableau.

[5] I use SNARK's native Lisp S-expression syntax here. SNARK treats numbers and strings as *constructors* or *unique names* that are unequal by definition to other constructors.

rewriting is incomplete; it must be backed up by paramodulation for complete reasoning but this is absent in the case of procedural attachments. Although `(= ($$sum 1 1) 2)` can be rewritten to `true`, `(= ($$sum x 1) 2)` cannot be solved.

The intent of procedural attachment for functions like `$$sum` is to simulate the use of an infinite set of equalities `(= ($$sum 0 0) 0)`, `(= ($$sum 0 1) 1)`, etc. but these equalities are to be used only for rewriting and not for paramodulation because `($$sum x 1)` would have an infinite number of paramodulants.

It is up to the user's care or luck that terms with procedural attachments are sufficiently instantiated to be rewritten when they have to be. A speculative solution is to *purify* expressions with procedurally attached functions so that unification can succeed in the absence of rewriting that will only happen later. For example, `(= ($$sum x 1) 2)` would be replaced `(= y 2)` with the added constraint `(= y ($$sum x 1))` that must be satisfied eventually. This raises a further question of whether the theorem prover should be doing arithmetic or algebra. If the former, no proof of `(= ($$sum x 1) 2)` will be found unless `x` is instantiated to a number. Adding algebraic constraints would increase the theorem prover's capabilities, but would be an unbounded additional implementation requirement unless there is an effective way to use an existing computer algebra system.

Another cautionary note is that procedural attachment may introduce symbols that were not present in the original problem. These must be accommodated by term ordering and indexing and maybe even the search strategy. The axioms

```
(p 0)
(forall (x) (implies (p x) (p (s x))))
```

can produce an infinite sequence of formulas of the form `(p (s ... (s 0) ...)))`, and ordering formulas for inference by size is a fair search strategy but

```
(p 0)
(forall (x) (implies (p x) (p ($$sum x 1))))
```

can produce an infinite sequence of formulas `(p 1)`, `(p 2)`, etc. that may all be preferred for their size to other formulas resulting in an unfair (incomplete) search strategy.

Besides code for procedurally attached rewriting, SNARK provides for procedural attachments to relation symbols to simulate resolution with all the tuples in the relation.

A major reason to use procedural attachment is to help SNARK avoid doing types of reasoning that resolution theorem provers are poor at but that are easy for database systems, for example. Problems like "Alice's children are Bill, Bob, and Bruce. How many children does Alice have?" are difficult and similar in character to subproblems in applications that use SNARK. To be able to give a definite answer to this problem, one has to make the closed world assumption (there are no other children of Alice) and the unique names assumption (Bill, Bob, and Bruce or all different). The formulation of the problem includes

```
(forall (x) (iff (childof Alice x)
                 (or (= x Bill) (x Bob) (x Bruce))))
(not (= Bill Bob))
(not (= Bill Bruce))
(not (= Bob Bruce))
```

The inequalities can be omitted in SNARK if strings `"Bill"`, `"Bob"`, and `"Bruce"` are used. These are assumed to be distinct so that equations of them will be rewritten to `false`. The representation is more concise and more easily reasoned with, but some incompleteness has been introduced: `(exists (x) (not (= x "Bill")))` cannot be answered because the ability to enumerate terms unequal to `"Bill"` has been lost.

If the question were "Does Alice have (exactly) 3 children?", this could be augmented by formulas, resembling in character pigeonhole principle problems, that test whether there are 3 but not 4 distinct children of Alice. But the "How many" question suggests a need to collect the children in a set and compute the size of the set, which is easy for a database system but hard for a theorem prover. Counting is even more challenging for a theorem prover when we are counting x that satisfy some condition $P(x)$ where P is a complex formula, with connectives, quantifiers, etc.

The same procedurally attached literals may occur in many clauses generated during a search for a proof. Unless the procedural attachment is very cheap to compute, its results should be cached for later use so that the search will not be impeded and the information provider will not be burdened by duplicate requests.

8 Sorts, Hybrid Reasoning Made Easy

Applications often require reasoning about taxonomies, which can be done very effectively with specialized description logic reasoners [4, 2].

Taxonomic reasoning can combined with resolution theorem proving by including taxonomic literals in clauses and permitting pairs of literals to be resolved if they are inconsistent (or conditionally inconsistent) according to the taxonomic reasoner. For example, the clauses

```
(man Socrates)
(forall (x) (implies (human x) (mortal x)))
(not (mortal Socrates))
```

can be refuted because the positive occurrence of `man` and negative occurrence of `human` are inconsistent (after unifying x and Socrates) if the fact that all men are human is part of a built-in taxonomic theory.

Theory resolution [31] is a general method for this idea of extending procedural attachment to sets of literals. Chain resolution [37] is a recent proposal for building in theories of binary clauses (an early example is Dixon's Z-resolution [7]). Tautology checking, factoring, and subsumption as well as resolution must all be

extended. A serious complication is that extending resolution and subsumption requires modification of indexing so that clauses with taxonomically relevant but not syntactically matching literals will be retrieved for inference.

An extremely useful subset of the capability of incorporating taxonomic reasoning into a theorem prover can be obtained more easily by using sorted logic instead of standard unsorted logic [40, 9]. With sorted logics, theories can be expressed more concisely, reasoning can be more efficient, answers can be more general, and requiring axioms to be well-sorted reduces errors.[6]

For example, the clauses

```
(forall ((x :sort human)) (mortal x))
(not (mortal Socrates))
```

can be refuted in a sort theory that includes the declarations that Socrates is a man and all men are human.

I call this reasoning *with* rather than reasoning *about* taxonomic information. Socrates can be declared to be a man and the consequences of that can be derived, but that Socrates is a man cannot be derived (by satisfying the defining conditions for being a man).

SNARK uses sorted reasoning with several simplifying assumptions that allow for easy implementation and a simple interface to the sort reasoner:

- Sorts are nonempty.
- The sort of a term is determined by its head symbol.
- If s_1 and s_2 are two different sorts, then either
  1. s_1 and s_2 have no common elements,
  2. s_1 is a subsort of s_2,
  3. s_2 is a subsort of s_1, or
  4. there is a sort s_3 that exactly contains their intersection.

Unification is restricted so that a variable can only be bound to a term of the same sort or a subsort. When unifying variables x_1 and x_2 of sorts s_1 and s_2, in Case 4, x_1 and x_2 are both bound to a new variable x_3 of sort s_3. Paramodulation and rewriting are restricted to replace terms only by terms of the same sort or a subsort.

Argument sorts can be declared for functions and relations so that only well-sorted formulas can be input. The restrictions on unification, paramodulation, and rewriting ensure that derived clauses are well-sorted.

SNARK has a simple but useful and fast sort reasoner that allows declaration of subsorts, sort intersections, and emptiness of intersections. But the simple interface makes it rather easy to substitute a more powerful description logic reasoner for it.

[6] We have found this to be a major benefit when developing theories for deductive question answering applications.

9 AC Unification, the Long Road to Full Use

Associative-commutative (AC) functions occur frequently in theories and are poorly handled by theorem provers without special treatment. But even the conceptually simple idea of substituting AC unification for ordinary unification has many ramifications for theorem provers that require a startling amount of theoretical and implementation effort.

An AC unification algorithm was first developed in 1975 [30], but not proved complete until 1984 [8]. It has proven to be quite useful, notably contributing to McCune's proof that all Robbins algebras are Boolean [19].

But developing and implementing a special unification algorithm is not enough for its full use. The architecture of the theorem prover must support a unification algorithm produce more than one unifier. It is important to be able to index AC terms efficiently. There is a complex and never implemented AC-discrimination tree method [26]. For SNARK at least it would be helpful to have an AC compatible extension of path indexing as well.

Rewriting and paramodulation with AC functions require *extended rewrites* [24]. A rewrite $f(s_1, \ldots, s_n) \rightarrow t$ with AC function f must usually be accompanied by an extended rewrite $f(s_1, \ldots, s_n, x) \rightarrow f(t, x)$. For example, the rewrite $f(g(x), x) \rightarrow e$ is not applicable to $f(g(a), a, b, c)$, but its extended rewrite is.

Term ordering is a contemporary necessity for effective equality reasoning, but the usual term orderings are incompatible with AC functions. AC compatible term orderings have now been developed, but they are much more work to implement than the orderings on which they are based. SNARK has an implementation of the AC compatible extension of recursive path ordering [25] but its exponential behavior is sometimes a serious problem in practice. An AC compatible extension of Knuth-Bendix ordering is claimed to overcome this deficiency [13]. Because recursive path ordering and Knuth-Bendix ordering can each orient some systems of equations that the other cannot, it is important that users be given the choice, but this increases the cost of special treatment of AC functions.

Supporting bag (unordered list) or set datatypes may be more important for applications than AC per se. There is a temptation to use AC unification for bags and AC1 unification for sets. But $bag(x, y, z)$, the flattened form of $bag(x, bag(y, z))$ with AC function bag, is not a suitable representation for the 3-element bag with elements x, y, and z because it does not allow proper nesting of bags. If $x = bag(u, v)$ then $bag(x, y, z) = bag(u, v, y, z)$ instead of a bag whose elements are $bag(u, v)$, y, and z. Basing the theory of bag or sets on union operations is more reasonable. The equations

$$bagunion(x, bagunion(y, z)) = bagunion(bagunion(x, y), z)$$
$$bagunion(x, y) = bagunion(y, x)$$

are valid but the arguments of the AC *bagunion* function are expected to be bags, not elements. This problem can be overcome by wrapping elements in singleton bag constructors so that $bagunion(x, y)$ means the union of bags x and y and

$bagunion(bag(x), bag(y))$ means the bag whose elements are x and y. However, this is cumbersome if one is mainly interested in bags and their elements, not their unions.

Just as lists are conventionally represented in theorem provers by means *cons* and *nil*, an attractive alternative approach [6] is to represent bags by means of *bagcons* and *emptybag* so that $bagcons(x, bagcons(y, bagcons(z, emptybag)))$ represents the 3-element bag with elements x, y, and z. Instead of being AC, the function *bagcons* has the property

$$bagcons(x, bagcons(y, z)) = bagcons(y, bagcons(x, z)).$$

And like *cons*, *bagcons* can have a variable as its second argument to represent bags with unspecified additional elements. This representation provides for useful operations on bags without the cost (or benefit) of complete bag reasoning with unions and should be easier to accommodate in indexing and ordering than AC functions.

Sets are often even more useful than bags and can be treated in a similar fashion [6]. However, sets pose additional problems unless inequality of elements can be assumed or determined. For example, $setcons(x, setcons(y, emptyset)) = setcons(y, emptyset)$ if $x = y$. Only bag unification has been implemented in SNARK.

Incorporation of specialized reasoning for other equational theories requires similar attention to make them fully effective.

10 Conclusion

Some theorem provers like SNARK's `coder` for condensed detachment problems and PTTP's implementation of the model elimination procedure are complete, fast, easy to implement, sometimes surprisingly successful, but also quite limited.

Resolution and paramodulation offer a large range of options and opportunities for extension to support the reasoning required by applications such as deductive question answering. Fundamental difficulties like the incompatibility of paramodulation and set of support and the need to adapt indexing to changes in how terms and formulas match make maintaining completeness and performance very challenging. Simple additions like AC unification can require large changes throughout the theorem prover to indexing, rewriting, and term ordering.

On a more positive note, occasionally a powerful reasoning technique can be added easily. This is the case for sorted resolution and paramodulation. I believe this may be the quickest and best way to add taxonomic reasoning.

Theorem prover implementation is harder than it needs to be. Algorithms in the literature are described concisely and mathematically to facilitate proofs of correctness rather than provide guides to efficient implementation. A good example is that the standard presention of lexical recursive path orderings encourages an inefficient implementation with exponential behavior [14]. To devise an efficient one is left as an exercise in [3]. Another example of valuing pedagogical

simplicity over implementation reality is the presentation of ordered resolution and paramodulation as if equality is the only relation.

I believe that executable code in high-level languages like Lisp, ML, or Python can communicate algorithms as well as pseudocode etc., be less subject to ambiguity or misinterpretation, and provide a reference against which to test new implementations for correctness and performance.

Although theorem prover code is available, there is sadly little reuse of components. Theorem provers have many components that require substantial theoretical and implementation effort. Many of these are so interdependent and tightly coupled that they must be developed in concert. But when tasks are more easily separated, such as CNF conversion, relevancy testing, extracting sort theories, and deciding what inference rules to use, it is undesirable for them to be implemented independently, often suboptimally, in many theorem provers.

Acknowledgments

I am very grateful for Richard Waldinger's collaboration and support over many years. His domain theory development and collaboration with users are the key to SNARK's usefulness for deductive question answering. Larry Wos, John Halleck, and Dolph Ulrich provided vital information and encouragement for the work on single axioms in **BCI** and **BCK** logics. This paper was improved by comments from Renate Schmidt, Richard Waldinger, and Larry Wos.

References

[1] Amir, E., Maynard-Zhang, P.: Logic-based subsumption architecture. Artificial Intelligence 153(1-2), 167–237 (2004)
[2] Baader, F., Calvanese, D., McGuiness, D., Nardi, D., Patel-Schneider, P. (eds.): The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press, Cambridge (2003)
[3] Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
[4] Brachman, R.J., Levesque, H.J.: Knowledge Representation and Reasoning. Morgan Kaufmann, San Francisco (2004)
[5] Chang, C.-L., Lee, R.C.-T.: Symbolic Logic and Mechanical Theorem Proving. Academic Press, London (1973)
[6] Dantsin, E., Voronkov, A.: A nondeterministic polynomial-time unification algorithm for bags, sets and trees. In: Thomas, W. (ed.) FOSSACS 1999. LNCS, vol. 1578, pp. 180–196. Springer, Heidelberg (1999)
[7] Dixon, J.K.: Z-resolution: Theorem-proving with compiled axioms. J. ACM 20(1), 127–147 (1973)
[8] Fages, F.: Associative-commutative unification. J. Symbolic Computation 3(3), 257–275 (1987)
[9] Frisch, A.M.: The substitutional framework for sorted deduction: Fundamental results on hybrid reasoning. Artificial Intelligence 49(1-3), 161–198 (1991)
[10] Graf, P.: Term Indexing. LNCS (LNAI), vol. 1053. Springer, Heidelberg (1996)

[11] Green, C.: Theorem-proving by resolution as a basis for question-answering systems. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence 4, pp. 183–205. Edinburgh University Press (1969)
[12] Korf, R.E.: Iterative-deepening-A*: An optimal admissible tree search. In: IJCAI, pp. 1034–1036 (1985)
[13] Korovin, K., Voronkov, A.: An AC-compatible Knuth-Bendix order. In: Baader, F. (ed.) CADE 2003. LNCS, vol. 2741, pp. 47–59. Springer, Heidelberg (2003)
[14] Löchner, B.: Things to know when implementing LPO. International Journal on Artificial Intelligence Tools 15(1), 53–80 (2006)
[15] Loveland, D.W.: A simplified format for the model elimination theorem-proving procedure. J. ACM 16(3), 349–363 (1969)
[16] Loveland, D.W.: Automated Theorem Proving: A Logical Basis. North-Holland, Amsterdam (1978)
[17] Lowry, M., Philpot, A., Pressburger, T., Underwood, I., Waldinger, R., Stickel, M.: Amphion: Automatic programming for the NAIF toolkit. NASA Science Information Systems Newsletter 31, 22–25 (1994)
[18] Manna, Z., Waldinger, R.J.: A deductive approach to program synthesis. ACM Trans. Programming Languages and Systems 2(1), 90–121 (1980)
[19] McCune, W.: Solution of the robbins problem. J. Automated Reasoning 19(3), 263–276 (1997)
[20] McCune, W.: Otter 3.3 reference manual. Technical Memorandum 263, Mathematics and Computer Science Divison, Argonne National Laboratory (August 2003)
[21] McCune, W., Wos, L.: Experiments in automated deduction with condensed detachment. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 209–223. Springer, Heidelberg (1992)
[22] Newell, A., Shaw, J.C., Simon, H.A.: Empirical explorations with the logic theory machine: a case study in heuristics. In: Feigenbaum, E.A., Feldman, J. (eds.) Computers and Thought, pp. 109–133. McGraw-Hill, New York (1963)
[23] Paskevich, A.: Connection tableaux with lazy paramodulation. J. Automated Reasoning 40(2-3), 179–194 (2008)
[24] Peterson, G.E., Stickel, M.E.: Complete sets of reductions for some equational theories. J. ACM 28(2), 233–264 (1981)
[25] Rubio, A.: A fully syntactic AC-RPO. Inf. Comput. 178(2), 515–533 (2002)
[26] Sekar, R., Ramakrishnan, I.V., Voronkov, A.: Term indexing. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 1853–1964. MIT Press, Cambridge (2001)
[27] Shrager, J., Waldinger, R., Stickel, M., Massar, J.P.: Deductive biocomputing. PLoS ONE 2(4), e339 (2007)
[28] Slagle, J.R.: Experiments with a deductive question-answering program. Commun. ACM 8(12), 792–798 (1965)
[29] Snyder, W., Lynch, C.: Goal directed strategies for paramodulation. In: Book, R.V. (ed.) RTA 1991. LNCS, vol. 488, pp. 150–161. Springer, Heidelberg (1991)
[30] Stickel, M.E.: A unification algorithm for associative-commutative functions. J. ACM 28(3), 423–434 (1981)
[31] Stickel, M.E.: Automated deduction by theory resolution. J. Automated Reasoning 1(4), 333–355 (1985)
[32] Stickel, M.E.: A Prolog technology theorem prover: Implementation by an extended Prolog compiler. J. Automated Reasoning 4(4), 353–380 (1988)
[33] Stickel, M.E.: A Prolog technology theorem prover: A new exposition and implementation in Prolog. Theoretical Computer Science 104(1), 109–128 (1992)

[34] Stickel, M.E., Tyson, M.: An analysis of consecutively bounded depth-first search with applications in automated deduction. In: IJCAI, pp. 1073–1075 (1985)
[35] Stickel, M.E., Waldinger, R.J., Chaudhri, V.K.: A guide to Snark. Technical report, Artificial Intelligence Center, SRI International (May 2000), http://www.ai.sri.com/snark/tutorial/tutorial.html
[36] Stickel, M.E., Waldinger, R.J., Lowry, M.R., Pressburger, T., Underwood, I.: Deductive composition of astronomical software from subroutine libraries. In: Bundy, A. (ed.) CADE 1994. LNCS, vol. 814, pp. 341–355. Springer, Heidelberg (1994)
[37] Tammet, T.: Chain resolution for the semantic web. In: Basin, D.A., Rusinowitch, M. (eds.) IJCAR 2004. LNCS, vol. 3097, pp. 307–320. Springer, Heidelberg (2004)
[38] Waldinger, R., Appelt, D.E., Dungan, J.L., Fry, J., Hobbs, J., Israel, D.J., Jarvis, P., Martin, D., Riehemann, S., Stickel, M.E., Tyson, M.: Deductive question answering from multiple resources. In: Maybury, M.T. (ed.) New Directions in Question Answering, pp. 253–262. AAAI Press, Menlo Park (2004)
[39] Waldinger, R.J.: Whatever happened to deductive question answering? In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS, vol. 4790, pp. 15–16. Springer, Heidelberg (2007)
[40] Walther, C.: A Many-Sorted Calculus Based on Resolution and Paramodulation. Pitman, London (1987)
[41] Wos, L.: Automated Reasoning: 33 Basic Research Problems. Prentice-Hall, Englewood Cliffs (1988)
[42] Wos, L.: A milestone reached and a secret revealed. J. Automated Reasoning 27(2), 89–95 (2001)
[43] Wos, L.: From the AAR president. Association for Automated Reasoning Newsletter 83 (April 2009)
[44] Wos, L., Overbeek, R.A., Lusk, E.L.: Subsumption, a sometimes undervalued procedure. In: Lassez, J.-L., Plotkin, G. (eds.) Computational Logic - Essays in Honor of Alan Robinson, pp. 3–40 (1991)
[45] Wos, L., Pieper, G.W.: Automated Reasoning and the Discovery of Missing and Elegant Proofs. Rinton Press, Paramus (2003)

Termination Analysis by Dependency Pairs and Inductive Theorem Proving*

Stephan Swiderski[1], Michael Parting[1], Jürgen Giesl[1], Carsten Fuhs[1], and Peter Schneider-Kamp[2]

[1] LuFG Informatik 2, RWTH Aachen University, Germany
[2] Dept. of Mathematics & CS, University of Southern Denmark, Odense, Denmark

Abstract. Current techniques and tools for automated termination analysis of term rewrite systems (TRSs) are already very powerful. However, they fail for algorithms whose termination is essentially due to an *inductive* argument. Therefore, we show how to couple the *dependency pair* method for TRS termination with inductive theorem proving. As confirmed by the implementation of our new approach in the tool AProVE, now TRS termination techniques are also successful on this important class of algorithms.

1 Introduction

There are many powerful techniques and tools to prove termination of TRSs automatically. Moreover, TRS tools are also very successful in termination analysis of real programming languages like, e.g., Haskell and Prolog [12,31]. To measure their performance, there is an annual *International Competition of Termination Provers*,[1] where the tools compete on a large data base of TRSs. Nevertheless, there exist natural algorithms like the following one where all these tools fail.

Example 1. Consider the following TRS $\mathcal{R}_{sort}$.

$$\begin{aligned}
\mathsf{ge}(x,\mathsf{0}) &\to \mathsf{true} & \mathsf{eq}(\mathsf{0},\mathsf{0}) &\to \mathsf{true}\\
\mathsf{ge}(\mathsf{0},\mathsf{s}(y)) &\to \mathsf{false} & \mathsf{eq}(\mathsf{s}(x),\mathsf{0}) &\to \mathsf{false}\\
\mathsf{ge}(\mathsf{s}(x),\mathsf{s}(y)) &\to \mathsf{ge}(x,y) & \mathsf{eq}(\mathsf{0},\mathsf{s}(y)) &\to \mathsf{false}\\
& & \mathsf{eq}(\mathsf{s}(x),\mathsf{s}(y)) &\to \mathsf{eq}(x,y)\\
\mathsf{max}(\mathsf{nil}) &\to \mathsf{0} & & \\
\mathsf{max}(\mathsf{co}(x,\mathsf{nil})) &\to x & \mathsf{if}_1(\mathsf{true},x,y,xs) &\to \mathsf{max}(\mathsf{co}(x,xs))\\
\mathsf{max}(\mathsf{co}(x,\mathsf{co}(y,xs))) &\to \mathsf{if}_1(\mathsf{ge}(x,y),x,y,xs) & \mathsf{if}_1(\mathsf{false},x,y,xs) &\to \mathsf{max}(\mathsf{co}(y,xs))\\
\mathsf{del}(x,\mathsf{nil}) &\to \mathsf{nil} & \mathsf{if}_2(\mathsf{true},x,y,xs) &\to xs\\
\mathsf{del}(x,\mathsf{co}(y,xs)) &\to \mathsf{if}_2(\mathsf{eq}(x,y),x,y,xs) & \mathsf{if}_2(\mathsf{false},x,y,xs) &\to \mathsf{co}(y,\mathsf{del}(x,xs))\\
\mathsf{sort}(\mathsf{nil}) &\to \mathsf{nil} & & \\
\mathsf{sort}(\mathsf{co}(x,xs)) &\to \mathsf{co}(\,\mathsf{max}(\mathsf{co}(x,xs)),\ \mathsf{sort}(\mathsf{del}(\,\mathsf{max}(\mathsf{co}(x,xs)),\ \mathsf{co}(x,xs)\,))\,) & &
\end{aligned}$$

* Supported by the DFG Research Training Group 1298 (*AlgoSyn*), the DFG grant GI 274/5-2, and the G.I.F. grant 966-116.6.

[1] `http://termination-portal.org/wiki/Termination_Competition`

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 322–338, 2009.

Here, numbers are represented with 0 *and* s *(for the successor function) and lists are represented with* nil *(for the empty list) and* co *(for list insertion). For any list* xs, $\mathsf{max}(xs)$ *computes its maximum (where* $\mathsf{max}(\mathsf{nil})$ *is* $\mathsf{0}$*), and* $\mathsf{del}(n, xs)$ *deletes the first occurrence of* n *from the list* xs*. If* n *does not occur in* xs*, then* $\mathsf{del}(n, xs)$ *returns* xs*. Algorithms like* max *and* del *are often expressed with conditions. Such conditional rules can be automatically transformed into unconditional ones (cf. e.g. [27]) and we already did this transformation in our example. To sort a non-empty list* ys *(i.e., a list of the form "*$\mathsf{co}(x, xs)$*"),* $\mathsf{sort}(ys)$ *reduces to "*$\mathsf{co}(\mathsf{max}(ys), \mathsf{sort}(\mathsf{del}(\mathsf{max}(ys), ys))$*". So* $\mathsf{sort}(ys)$ *starts with the maximum of* ys *and then* sort *is called recursively on the list that results from* ys *by deleting the first occurrence of its maximum. Note that*

$$\textit{every non-empty list contains its maximum.} \tag{1}$$

Hence, the list $\mathsf{del}(\mathsf{max}(ys), ys)$ *is shorter than* ys *and thus,* $\mathcal{R}_{sort}$ *is terminating.*

So (1) is the main argument needed for termination of $\mathcal{R}_{sort}$. Thus, when trying to prove termination of TRSs like $\mathcal{R}_{sort}$ automatically, one faces 2 problems:

(a) One has to detect the main argument needed for termination and one has to find out that the TRS is terminating provided that this argument is valid.
(b) One has to prove that the argument detected in (a) is valid.

In our example, (1) requires a non-trivial induction proof that relies on the max- and del-rules. Such proofs cannot be done by TRS termination techniques, but they could be performed by state-of-the-art inductive theorem provers [4,5,7,8,20,21,33,34,36]. So to solve Problem (b), we would like to couple termination techniques for TRSs (like the *dependency pair* (DP) method which is implemented in virtually every current TRS termination tool) with an inductive theorem prover. Ideally, this prover should perform the validity proof in (b) fully automatically, but of course it is also possible to have user interaction here. However, it still remains to solve Problem (a). Thus, one has to extend the TRS termination techniques such that they can automatically synthesize an argument like (1) and find out that this argument is sufficient in order to complete the termination proof. This is the subject of the current paper.

There is already work on applying inductive reasoning in termination proofs. Some approaches like [6,15,16,28] integrate special forms of inductive reasoning into the termination method itself. These approaches are successful on certain forms of algorithms, but they cannot handle examples like Ex. 1 where one needs more general forms of inductive reasoning. Therefore, in this paper our goal is to couple the termination method with an arbitrary (black-box) inductive theorem prover which may use any kind of proof techniques.

There exist also approaches like [5,10,22,25,32] where a full inductive theorem prover is used to perform the whole termination proof of a functional program. Such approaches could potentially handle algorithms like Ex. 1 and indeed, Ex. 1 is similar to an algorithm from [10,32]. In general, to prove termination one has to solve two tasks: (i) one has to synthesize suitable well-founded orders and

(ii) one has to prove that recursive calls decrease w.r.t. these orders. If there is just an inductive theorem prover available for the termination proof, then for Task (i) one can only use a fixed small set of orders or otherwise, ask the user to provide suitable well-founded orders manually. Moreover, then Task (ii) has to be tackled by the full theorem prover which may often pose problems for automation. In contrast, there are many TRS techniques and tools available that are extremely powerful for Task (i) and that offer several specialized methods to perform Task (ii) fully automatically in a very efficient way. So in most cases, no inductive theorem prover is needed for Task (ii). Nevertheless, there exist important algorithms (like $\mathcal{R}_{sort}$) where Task (ii) indeed requires inductive theorem proving. Thus, we propose to use the "best of both worlds", i.e., to apply TRS techniques whenever possible, but to use an inductive theorem prover for those parts where it is needed.

After recapitulating the DP method in Sect. 2, in Sect. 3 we present the main idea for our improvement. To make this improvement powerful in practice, we need the new result that innermost termination of many-sorted term rewriting and of unsorted term rewriting is equivalent. We expect that this observation will be useful also for other applications in term rewriting, since TRSs are usually considered to be unsorted. We use this result in Sect. 4 where we show how the DP method can be coupled with inductive theorem proving in order to prove termination of TRSs like $\mathcal{R}_{sort}$ automatically.

We implemented our new technique in the termination prover AProVE [13]. Here, we used a small inductive theorem prover inspired by [5,7,21,33,34,36] which had already been implemented in AProVE before. Although this inductive theorem prover is less powerful than the more elaborated full theorem provers in the literature, it suffices for many of those inductive arguments that typically arise in termination proofs. This is confirmed by the experimental evaluation of our contributions in Sect. 5. Note that the results of this paper allow to couple *any* termination prover implementing DPs with *any* inductive theorem prover. Thus, by using a more powerful inductive theorem prover than the one integrated in AProVE, the power of the resulting tool could even be increased further.

2 Dependency Pairs

We assume familiarity with term rewriting [3] and briefly recapitulate the DP method. See e.g. [2,11,14,18,19] for further motivations and extensions.

Definition 2 (Dependency Pairs). *For a TRS $\mathcal{R}$, the* defined *symbols $\mathcal{D}_\mathcal{R}$ are the root symbols of left-hand sides of rules. All other function symbols are called* constructors. *For every defined symbol $f \in \mathcal{D}_\mathcal{R}$, we introduce a fresh* tuple symbol $f^\sharp$ *with the same arity. To ease readability, we often write F instead of $f^\sharp$, etc. If $t = f(t_1, \ldots, t_n)$ with $f \in \mathcal{D}_\mathcal{R}$, we write $t^\sharp$ for $f^\sharp(t_1, \ldots, t_n)$. If $\ell \to r \in \mathcal{R}$ and t is a subterm of r with defined root symbol, then the rule $\ell^\sharp \to t^\sharp$ is a* dependency pair *of $\mathcal{R}$. The set of all dependency pairs of $\mathcal{R}$ is denoted $DP(\mathcal{R})$.*

We get the following set $DP(\mathcal{R}_{sort})$, where GE is ge's tuple symbol, etc.

$$\mathsf{GE}(\mathsf{s}(x),\mathsf{s}(y)) \to \mathsf{GE}(x,y) \quad (2)$$
$$\mathsf{EQ}(\mathsf{s}(x),\mathsf{s}(y)) \to \mathsf{EQ}(x,y) \quad (3)$$
$$\mathsf{MAX}(\mathsf{co}(x,\mathsf{co}(y,xs))) \to \mathsf{IF}_1(\mathsf{ge}(x,y),x,y,xs) \quad (4)$$
$$\mathsf{MAX}(\mathsf{co}(x,\mathsf{co}(y,xs))) \to \mathsf{GE}(x,y) \quad (5)$$
$$\mathsf{IF}_1(\mathsf{true},x,y,xs) \to \mathsf{MAX}(\mathsf{co}(x,xs)) \quad (6)$$
$$\mathsf{IF}_1(\mathsf{false},x,y,xs) \to \mathsf{MAX}(\mathsf{co}(y,xs)) \quad (7)$$
$$\mathsf{DEL}(x,\mathsf{co}(y,xs)) \to \mathsf{IF}_2(\mathsf{eq}(x,y),x,y,xs) \quad (8)$$
$$\mathsf{DEL}(x,\mathsf{co}(y,xs)) \to \mathsf{EQ}(x,y) \quad (9)$$
$$\mathsf{IF}_2(\mathsf{false},x,y,xs) \to \mathsf{DEL}(x,xs) \quad (10)$$
$$\mathsf{SORT}(\mathsf{co}(x,xs)) \to \mathsf{SORT}(\mathsf{del}(\mathsf{max}(\mathsf{co}(x,xs)),\mathsf{co}(x,xs))) \quad (11)$$
$$\mathsf{SORT}(\mathsf{co}(x,xs)) \to \mathsf{DEL}(\mathsf{max}(\mathsf{co}(x,xs)),\mathsf{co}(x,xs)) \quad (12)$$
$$\mathsf{SORT}(\mathsf{co}(x,xs)) \to \mathsf{MAX}(\mathsf{co}(x,xs)) \quad (13)$$

In this paper, we only regard the *innermost* rewrite relation $\xrightarrow{\mathsf{i}}$ and prove innermost termination, since techniques for innermost termination are considerably more powerful than those for full termination. For large classes of TRSs (e.g., TRSs resulting from programming languages [12,31] or non-overlapping TRSs like Ex. 1), innermost termination is sufficient for termination.

For 2 TRSs $\mathcal{P}$ and $\mathcal{R}$ (where $\mathcal{P}$ usually consists of DPs), an *innermost* $(\mathcal{P},\mathcal{R})$-*chain* is a sequence of (variable-renamed) pairs $s_1 \to t_1, s_2 \to t_2, \ldots$ from $\mathcal{P}$ such that there is a substitution σ (with possibly infinite domain) where $t_i\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s_{i+1}\sigma$ and $s_i\sigma$ is in normal form w.r.t. $\mathcal{R}$, for all i.[2] The main result on DPs states that $\mathcal{R}$ is innermost terminating iff there is no infinite innermost $(DP(\mathcal{R}),\mathcal{R})$-chain.

As an example for a chain, consider "(11), (11)", i.e.,

$$\mathsf{SORT}(\mathsf{co}(x\ ,xs\ )) \to \mathsf{SORT}(\mathsf{del}(\mathsf{max}(\mathsf{co}(x\ ,xs\ )),\mathsf{co}(x\ ,xs\ ))),$$
$$\mathsf{SORT}(\mathsf{co}(x',xs')) \to \mathsf{SORT}(\mathsf{del}(\mathsf{max}(\mathsf{co}(x',xs')),\mathsf{co}(x',xs'))).$$

Indeed, if $\sigma(x) = \sigma(x') = \mathsf{0}$, $\sigma(xs) = \mathsf{co}(\mathsf{s}(\mathsf{0}),\mathsf{nil})$, and $\sigma(xs') = \mathsf{nil}$, then

$$\mathsf{SORT}(\mathsf{del}(\mathsf{max}(\mathsf{co}(x,xs)),\mathsf{co}(x,xs)))\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}_{sort}} \mathsf{SORT}(\mathsf{co}(x',xs'))\sigma.$$

Termination techniques are now called *DP processors* and they operate on pairs of TRSs $(\mathcal{P},\mathcal{R})$ (which are called *DP problems*).[3] Formally, a DP processor *Proc* takes a DP problem as input and returns a set of new DP problems which then have to be solved instead. A processor *Proc* is *sound* if for all DP problems $(\mathcal{P},\mathcal{R})$ with an infinite innermost $(\mathcal{P},\mathcal{R})$-chain there is also a $(\mathcal{P}',\mathcal{R}') \in \mathit{Proc}(\,(\mathcal{P},\mathcal{R})\,)$ with an infinite innermost $(\mathcal{P}',\mathcal{R}')$-chain. Soundness of a DP processor is required to prove innermost termination and in particular, to conclude that there is no infinite innermost $(\mathcal{P},\mathcal{R})$-chain if $\mathit{Proc}(\,(\mathcal{P},\mathcal{R})\,) = \varnothing$.

So innermost termination proofs in the DP framework start with the initial problem $(DP(\mathcal{R}),\mathcal{R})$. Then the problem is simplified repeatedly by sound DP

[2] All results of the present paper also hold if one regards *minimal* instead of ordinary innermost chains, i.e., chains where all $t_i\sigma$ are innermost terminating.

[3] To ease readability we use a simpler definition of *DP problems* than [11], since this simple definition suffices for the presentation of the new results of this paper.

processors. If all DP problems have been simplified to $\varnothing$, then innermost termination is proved. Thm. 3-5 recapitulate three of the most important processors.

Thm. 3 allows us to replace the TRS $\mathcal{R}$ in a DP problem $(\mathcal{P}, \mathcal{R})$ by the *usable rules*. These include all rules that can be used to reduce the terms in right-hand sides of $\mathcal{P}$ when their variables are instantiated with normal forms.

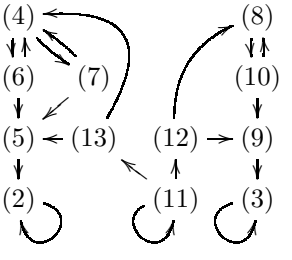
Theorem 3 (Usable Rule Processor [2,11]). *Let $\mathcal{R}$ be a TRS. For any function symbol f, let $Rls(f) = \{\ell \to r \in \mathcal{R} \mid \mathrm{root}(\ell) = f\}$. For any term t, the* usable rules $\mathcal{U}(t)$ *are the smallest set such that*

- $\mathcal{U}(x) = \varnothing$ *for every variable x and*
- $\mathcal{U}(f(t_1, \ldots, t_n)) = Rls(f) \cup \bigcup_{\ell \to r \in Rls(f)} \mathcal{U}(r) \cup \bigcup_{i=1}^{n} \mathcal{U}(t_i)$

For a TRS $\mathcal{P}$, its usable rules *are $\mathcal{U}(\mathcal{P}) = \bigcup_{s \to t \in \mathcal{P}} \mathcal{U}(t)$. Then the following DP processor Proc is sound: $Proc(\,(\mathcal{P}, \mathcal{R})\,) = \{\,(\mathcal{P}, \mathcal{U}(\mathcal{P}))\,\}$.*

In Ex. 1, this processor transforms the initial DP problem $(DP(\mathcal{R}_{sort}), \mathcal{R}_{sort})$ into $(DP(\mathcal{R}_{sort}), \mathcal{R}'_{sort})$. $\mathcal{R}'_{sort}$ is $\mathcal{R}_{sort}$ without the two sort-rules, since sort does not occur in the right-hand side of any DP and thus, its rules are not usable.

The next processor decomposes a DP problem into several sub-problems. To this end, one determines which pairs can follow each other in innermost chains by constructing an *innermost dependency graph*. For a DP problem $(\mathcal{P}, \mathcal{R})$, the nodes of the innermost dependency graph are the pairs of $\mathcal{P}$, and there is an arc from $s \to t$ to $v \to w$ iff $s \to t, v \to w$ is an innermost $(\mathcal{P}, \mathcal{R})$-chain. The graph obtained in our example is depicted on the side.



In general, the innermost dependency graph is not computable, but there exist many techniques to overapproximate this graph automatically, cf. e.g. [2,18]. In our example, these estimations would even yield the exact innermost dependency graph.

A set $\mathcal{P}' \neq \varnothing$ of DPs is a *cycle* if for every $s \to t$, $v \to w \in \mathcal{P}'$, there is a non-empty path from $s \to t$ to $v \to w$ traversing only pairs of $\mathcal{P}'$. A cycle $\mathcal{P}'$ is a (non-trivial) *strongly connected component (SCC)* if $\mathcal{P}'$ is not a proper subset of another cycle. The next processor allows us to prove termination separately for each SCC.

Theorem 4 (Dependency Graph Processor [2,11]). *The following DP processor Proc is sound: $Proc(\,(\mathcal{P}, \mathcal{R})\,) = \{(\mathcal{P}_1, \mathcal{R}), \ldots, (\mathcal{P}_n, \mathcal{R})\}$, where $\mathcal{P}_1, \ldots, \mathcal{P}_n$ are the SCCs of the innermost dependency graph.*

Our graph has the SCCs $\mathcal{P}_1 = \{(2)\}$, $\mathcal{P}_2 = \{(3)\}$, $\mathcal{P}_3 = \{(4), (6), (7)\}$, $\mathcal{P}_4 = \{(8), (10)\}$, $\mathcal{P}_5 = \{(11)\}$. Thus, $(DP(\mathcal{R}_{sort}), \mathcal{R}'_{sort})$ is transformed into the 5 new DP problems $(\mathcal{P}_i, \mathcal{R}'_{sort})$ for $1 \leq i \leq 5$ that have to be solved instead. For all problems except $(\{(11)\}, \mathcal{R}'_{sort})$ this is easily possible by the DP processors of this section (and this can also be done automatically by current termination tools). Therefore, we now concentrate on the remaining DP problem $(\{(11)\}, \mathcal{R}'_{sort})$.

A *reduction pair* $(\succsim, \succ)$ consists of a stable monotonic quasi-order $\succsim$ and a stable well-founded order $\succ$, where $\succsim$ and $\succ$ are compatible (i.e., $\succsim \circ \succ \circ \succsim \subseteq \succ$). For a DP problem $(\mathcal{P}, \mathcal{R})$, the following processor requires that all DPs in $\mathcal{P}$ are strictly or weakly decreasing and all rules $\mathcal{R}$ are weakly decreasing. Then one can delete all strictly decreasing DPs. Note that both TRSs and relations can be seen as sets of pairs of terms. Thus, $\mathcal{P} \setminus \succ$ denotes $\{s \to t \in \mathcal{P} \mid s \not\succ t\}$.

Theorem 5 (Reduction Pair Processor [2,11,18]). *Let $(\succsim, \succ)$ be a reduction pair. Then the following DP processor Proc is sound.*

$$Proc(\,(\mathcal{P}, \mathcal{R})\,) = \begin{cases} \{\,(\mathcal{P} \setminus \succ, \mathcal{R})\,\}, & \text{if } \mathcal{P} \subseteq \succsim \cup \succ \text{ and } \mathcal{R} \subseteq \succsim \\ \{\,(\mathcal{P}, \mathcal{R})\,\}, & \text{otherwise} \end{cases}$$

For the problem $(\{(11)\}, \mathcal{R}'_{sort})$, we search for a reduction pair where (11) is strictly decreasing (w.r.t. $\succ$) and the rules in $\mathcal{R}'_{sort}$ are weakly decreasing (w.r.t. $\succsim$). However, this is not satisfied by the orders available in current termination tools. That is not surprising, because termination of this DP problem essentially relies on the argument (1) that every non-empty list contains its maximum.

3 Many-Sorted Rewriting

Recall that our goal is to prove the absence of infinite innermost $(\mathcal{P}, \mathcal{R})$-chains. Each such chain would correspond to a reduction of the following form

$$s_1\sigma \;\to_{\mathcal{P}}\; t_1\sigma \;\xrightarrow{i}{}^{!}_{\mathcal{R}}\; s_2\sigma \;\to_{\mathcal{P}}\; t_2\sigma \;\xrightarrow{i}{}^{!}_{\mathcal{R}}\; s_3\sigma \;\to_{\mathcal{P}}\; t_3\sigma \;\xrightarrow{i}{}^{!}_{\mathcal{R}}\; \dots$$

where $s_i \to t_i$ are variable-renamed DPs from $\mathcal{P}$ and "$\xrightarrow{i}{}^{!}_{\mathcal{R}}$" denotes zero or more reduction steps to a normal form. The reduction pair processor ensures

$$s_1\sigma \;{}_{(}\!\succsim_{)}\; t_1\sigma \;\succsim\; s_2\sigma \;{}_{(}\!\succsim_{)}\; t_2\sigma \;\succsim\; s_3\sigma \;{}_{(}\!\succsim_{)}\; t_3\sigma \;\succsim\; \dots$$

Hence, strictly decreasing DPs (i.e., where $s_i\sigma \succ t_i\sigma$) cannot occur infinitely often in innermost chains and thus, they can be removed from the DP problem.

However, instead of requiring a strict decrease when going from the left-hand side $s_i\sigma$ of a DP to the right-hand side $t_i\sigma$, it would also be sufficient to require a strict decrease when going from the right-hand side $t_i\sigma$ to the *next* left-hand side $s_{i+1}\sigma$. In other words, if every reduction of $t_i\sigma$ to normal form makes the term strictly smaller w.r.t. $\succ$, then we would have $t_i\sigma \succ s_{i+1}\sigma$. Hence, then the DP $s_i \to t_i$ cannot occur infinitely often and could be removed from the DP problem. Our goal is to formulate a new processor based on this idea.

So essentially, we can remove a DP $s \to t$ from the DP problem, if

$$\text{for every normal substitution } \sigma,\ t\sigma \xrightarrow{i}{}^{!}_{\mathcal{R}} q \text{ implies } t\sigma \succ q. \tag{14}$$

In addition, all DPs and rules still have to be weakly decreasing. A substitution σ is called *normal* iff $\sigma(x)$ is in normal form w.r.t. $\mathcal{R}$ for all variables x.

So to remove (11) from the remaining DP problem $(\{(11)\}, \mathcal{R}'_{sort})$ of Ex. 1 with the criterion above, we have to use a reduction pair satisfying (14). Here, t is the right-hand side of (11), i.e., $t = \mathsf{SORT}(\mathsf{del}(\mathsf{max}(\mathsf{co}(x, xs)), \mathsf{co}(x, xs)))$.

Now we will weaken the requirement (14) step by step to obtain a condition amenable to automation. The current requirement (14) is still unnecessarily hard. For instance, in our example we also have to regard substitutions like $\sigma(x) = \sigma(xs) = \mathsf{true}$ and require that $t\sigma \succ q$ holds, although intuitively, here x stands for a natural number and xs stands for a list (and not a Boolean value). We will show that one does not have to require (14) for *all* normal substitutions, but only for "well-typed" ones. The reason is that if there is an infinite innermost reduction, then there is also an infinite innermost reduction of "well-typed" terms.

First, we make precise what we mean by "well-typed". Recall that up to now we regarded ordinary TRSs over untyped signatures $\mathcal{F}$. The following definition shows how to extend such signatures by (monomorphic) types, cf. e.g. [35].

Definition 6 (Typing). *Let $\mathcal{F}$ be an (untyped) signature. A many-sorted signature $\mathcal{F}'$ is a* typed variant *of $\mathcal{F}$ if it contains the same function symbols as $\mathcal{F}$, with the same arities. So f is a symbol of $\mathcal{F}$ with arity n iff f is a symbol of $\mathcal{F}'$ with a type of the form $\tau_1 \times \ldots \times \tau_n \to \tau$. Similarly, a typed variant $\mathcal{V}'$ of the set of variables $\mathcal{V}$ contains the same variables as $\mathcal{V}$, but now every variable has a sort τ. We always assume that for every sort τ, $\mathcal{V}'$ contains infinitely many variables of sort τ. A term over $\mathcal{F}$ and $\mathcal{V}$ is* well typed *w.r.t. $\mathcal{F}'$ and $\mathcal{V}'$ iff*

- *t is a variable (of some type τ in $\mathcal{V}'$) or*
- *$t = f(t_1, \ldots, t_n)$ with $n \geq 0$, where all t_i are well typed and have some type τ_i, and where f has type $\tau_1 \times \ldots \times \tau_n \to \tau$ in $\mathcal{F}'$. Then t has type τ.*

We only permit typed variants $\mathcal{F}'$ where there exist well-typed ground terms of types $\tau_1, \ldots, \tau_n$ over $\mathcal{F}'$, whenever some $f \in \mathcal{F}'$ has type $\tau_1 \times \ldots \times \tau_n \to \tau$.[4]

A TRS $\mathcal{R}$ over[5] *$\mathcal{F}$ and $\mathcal{V}$ is* well typed *w.r.t. $\mathcal{F}'$ and $\mathcal{V}'$ if for all $\ell \to r \in \mathcal{R}$, we have that ℓ and r are well typed and that they have the same type.*[6]

For any TRS $\mathcal{R}$ over a signature $\mathcal{F}$, one can use a standard type inference algorithm to compute a typed variant $\mathcal{F}'$ of $\mathcal{F}$ automatically such that $\mathcal{R}$ is well typed. Of course, a trivial solution is to use a many-sorted signature with just one sort (then every term and every TRS are trivially well typed). But to make our approach more powerful, it is advantageous to use the most general typed variant where $\mathcal{R}$ is well typed instead. Here, the set of terms is decomposed into as many sorts as possible. Then fewer terms are considered to be "well typed" and hence, the condition (14) has to be required for fewer substitutions σ.

For example, let $\mathcal{F} = \{\mathsf{0}, \mathsf{s}, \mathsf{true}, \mathsf{false}, \mathsf{nil}, \mathsf{co}, \mathsf{ge}, \mathsf{eq}, \mathsf{max}, \mathsf{if}_1, \mathsf{del}, \mathsf{if}_2, \mathsf{SORT}\}$. To make $\{(11)\} \cup \mathcal{R}'_{sort}$ well typed, we obtain the typed variant $\mathcal{F}'$ of $\mathcal{F}$ with the sorts nat, bool, list, and tuple. Here the function symbols have the following types.

[4] This is not a restriction, as one can simply add new constants to $\mathcal{F}$ and $\mathcal{F}'$.

[5] Note that $\mathcal{F}$ may well contain function symbols that do not occur in $\mathcal{R}$.

[6] W.l.o.g., here one may rename the variables in every rule. Then it is not a problem if the variable x is used with type τ_1 in one rule and with type τ_2 in another rule.

$$
\begin{array}{rcl@{\qquad}rcl}
0 & : & \mathsf{nat} & \mathsf{ge}, \mathsf{eq} & : & \mathsf{nat} \times \mathsf{nat} \to \mathsf{bool} \\
\mathsf{s} & : & \mathsf{nat} \to \mathsf{nat} & \mathsf{max} & : & \mathsf{list} \to \mathsf{nat} \\
\mathsf{true}, \mathsf{false} & : & \mathsf{bool} & \mathsf{if}_1, \mathsf{if}_2 & : & \mathsf{bool} \times \mathsf{nat} \times \mathsf{nat} \times \mathsf{list} \to \mathsf{list} \\
\mathsf{nil} & : & \mathsf{list} & \mathsf{SORT} & : & \mathsf{list} \to \mathsf{tuple} \\
\mathsf{co}, \mathsf{del} & : & \mathsf{nat} \times \mathsf{list} \to \mathsf{list} & & &
\end{array}
$$

Now we show that innermost termination is a *persistent* property, i.e., a TRS is innermost terminating iff it is innermost terminating on well-typed terms. Here, one can use any typed variant where the TRS is well typed. As noted by [26], persistence of innermost termination follows from results of [30], but to our knowledge, it has never been explicitly stated or applied in the literature before. Note that in contrast to innermost termination, full termination is only persistent for very restricted classes of TRSs, cf. [35].

Theorem 7 (Persistence). *Let $\mathcal{R}$ be a TRS over $\mathcal{F}$ and $\mathcal{V}$ and let $\mathcal{R}$ be well typed w.r.t. the typed variants $\mathcal{F}'$ and $\mathcal{V}'$. $\mathcal{R}$ is innermost terminating for all well-typed terms w.r.t. $\mathcal{F}'$ and $\mathcal{V}'$ iff $\mathcal{R}$ is innermost terminating (for* all *terms).*

Proof. For persistence, it suffices to show *component closedness* and *sorted modularity* [30]. A property is *component closed* if (a) $\Leftrightarrow$ (b) holds for all TRSs $\mathcal{R}$.

(a) $\to_{\mathcal{R}}$ has the property for all terms
(b) for every equivalence class Cl w.r.t. $\leftrightarrow^*_{\mathcal{R}}$, the restriction of $\to_{\mathcal{R}}$ to Cl has the property

Innermost termination is clearly component closed, since all terms occurring in an innermost reduction are from the same equivalence class.

A property is *sorted modular* if (c) and (d) are equivalent for all TRSs $\mathcal{R}_1$ and $\mathcal{R}_2$ forming a disjoint combination. So each $\mathcal{R}_i$ is a TRS over $\mathcal{F}_i$ and $\mathcal{V}$, $\mathcal{F}_i'$ and $\mathcal{V}'$ are typed variants of $\mathcal{F}_i$ and $\mathcal{V}$ where $\mathcal{R}_i$ is well typed, and $\mathcal{F}_1 \cap \mathcal{F}_2 = \varnothing$.

(c) for both i, $\mathcal{R}_i$ has the property for all well-typed terms w.r.t. $\mathcal{F}_i'$ and $\mathcal{V}'$
(d) $\mathcal{R}_1 \cup \mathcal{R}_2$ has the property for all well-typed terms w.r.t. $\mathcal{F}_1' \cup \mathcal{F}_2'$ and $\mathcal{V}'$

For innermost termination, (d) $\Rightarrow$ (c) is trivial. To show (c) $\Rightarrow$ (d), we adapt the proof for (unsorted) modularity of innermost termination in [17]. Assume there is a well-typed term t over $\mathcal{F}_1' \cup \mathcal{F}_2'$ and $\mathcal{V}'$ with infinite innermost $\mathcal{R}_1 \cup \mathcal{R}_2$-reduction. Then there is also a minimal such term (its proper subterms are innermost terminating w.r.t. $\mathcal{R}_1 \cup \mathcal{R}_2$). The reduction has the form $t \xrightarrow{\mathrm{i}}{}^*_{\mathcal{R}_1 \cup \mathcal{R}_2} t_1 \xrightarrow{\mathrm{i}}_{\mathcal{R}_1 \cup \mathcal{R}_2} t_2 \xrightarrow{\mathrm{i}}_{\mathcal{R}_1 \cup \mathcal{R}_2} \ldots$ where the step from t_1 to t_2 is the first root step. Such a root step must exist since t is minimal. Due to the innermost strategy, all proper subterms of t_1 are in $\mathcal{R}_1 \cup \mathcal{R}_2$-normal form. W.l.o.g., let $\mathrm{root}(t_1) \in \mathcal{F}_1$. Then $t_1 = C[s_1, \ldots, s_m]$ with $m \geq 0$, where C is a context without symbols from $\mathcal{F}_2$ and the roots of $s_1, \ldots, s_m$ are from $\mathcal{F}_2$. Since $s_1, \ldots, s_m$ are irreducible, the reduction from t_1 onwards is an $\mathcal{R}_1$-reduction, i.e., $t_1 \xrightarrow{\mathrm{i}}_{\mathcal{R}_1} t_2 \xrightarrow{\mathrm{i}}_{\mathcal{R}_1} \ldots$ Let $\overline{t_j}$ result from t_j by replacing $s_1, \ldots, s_m$ by fresh variables[7] $x_1, \ldots, x_m$. Thus, the $\overline{t_j}$ are well-typed terms over $\mathcal{F}_1'$ and $\mathcal{V}'$ with $\overline{t_1} \xrightarrow{\mathrm{i}}_{\mathcal{R}_1} \overline{t_2} \xrightarrow{\mathrm{i}}_{\mathcal{R}_1} \ldots$ which shows that $\overline{t_1}$ starts an infinite innermost $\mathcal{R}_1$-reduction. □

[7] Recall that $\mathcal{V}'$ has infinitely many variables for every sort.

We expect that there exist several points where Thm. 7 could simplify innermost termination proofs.[8] In this paper, we use Thm. 7 to weaken the condition (14) required to remove a DP from a DP problem $(\mathcal{P}, \mathcal{R})$. Now one can use any typed variant where $\mathcal{P} \cup \mathcal{R}$ is well typed. To remove $s \to t$ from $\mathcal{P}$, it suffices if

$$\text{for every normal } \sigma \textit{ where } t\sigma \textit{ is well typed, } t\sigma \xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}} q \text{ implies } t\sigma \succ q. \tag{15}$$

4 Coupling DPs and Inductive Theorem Proving

Condition (15) is still too hard, because up to now, $t\sigma$ does not have to be ground. We show (in Thm. 12) that for DP problems $(\mathcal{P}, \mathcal{R})$ satisfying suitable non-overlappingness requirements and where $\mathcal{R}$ is already innermost terminating, (15) can be relaxed to ground substitutions σ. Then $s \to t$ can be removed from $\mathcal{P}$ if

$$\begin{array}{l}\text{for every normal substitution } \sigma \text{ where } t\sigma \text{ is a well-typed } \textit{ground} \text{ term,}\\ t\sigma \xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}} q \text{ implies } t\sigma \succ q.\end{array} \tag{16}$$

Example 8. Innermost termination of $\mathcal{R}$ is really needed to replace (15) by (16). To see this, consider the DP problem $(\mathcal{P}, \mathcal{R})$ with $\mathcal{P} = \{\mathsf{F}(x) \to \mathsf{F}(x)\}$ and the non-innermost terminating TRS $\mathcal{R} = \{\mathsf{a} \to \mathsf{a}\}$.[9] *Let $\mathcal{F} = \{\mathsf{F}, \mathsf{a}\}$. We use a typed variant $\mathcal{F}'$ where $\mathsf{F} : \tau_1 \to \tau_2$ and $\mathsf{a} : \tau_1$. For the right-hand side $t = \mathsf{F}(x)$ of the DP, the only well-typed ground instantiation is $\mathsf{F}(\mathsf{a})$. Since this term has no normal form q, the condition (16) holds. Nevertheless, it is not sound to remove the only DP from $\mathcal{P}$, since $\mathsf{F}(x_1) \to \mathsf{F}(x_1), \mathsf{F}(x_2) \to \mathsf{F}(x_2), \ldots$ is an infinite innermost $(\mathcal{P}, \mathcal{R})$-chain (but there is no infinite innermost* ground *chain).*

To see the reason for the non-overlappingness requirement, consider $(\mathcal{P}, \mathcal{R})$ with $\mathcal{P} = \{\mathsf{F}(\mathsf{f}(x)) \to \mathsf{F}(\mathsf{f}(x))\}$ and $\mathcal{R} = \{\mathsf{f}(\mathsf{a}) \to \mathsf{a}\}$. Now $\mathcal{F} = \{\mathsf{F}, \mathsf{f}, \mathsf{a}\}$ and in the typed variant we have $\mathsf{F} : \tau_1 \to \tau_2$, $\mathsf{f} : \tau_1 \to \tau_1$, and $\mathsf{a} : \tau_1$. For the right-hand side $t = \mathsf{F}(\mathsf{f}(x))$ of the DP, the only well-typed ground instantiations are $\mathsf{F}(\mathsf{f}^n(\mathsf{a}))$ with $n \geq 1$. If we take the embedding order $\succ_{emb}$, then all well-typed ground instantiations of t are $\succ_{emb}$-greater than their normal form $\mathsf{F}(\mathsf{a})$. So Condition (16) would allow us to remove the only DP from $\mathcal{P}$. But again, this is unsound, since there is an infinite innermost $(\mathcal{P}, \mathcal{R})$-chain (but no such ground *chain).*

To prove a condition like (16), we replace (16) by the following condition (17), which is easier to check. Here, we require that for all instantiations $t\sigma$ as above, *every* reduction of $t\sigma$ to its normal form uses a strictly decreasing rule $\ell \to r$ (i.e., a rule with $\ell \succ r$) on a strongly monotonic position π. A position π in a term u is *strongly monotonic* w.r.t. $\succ$ iff $t_1 \succ t_2$ implies $u[t_1]_\pi \succ u[t_2]_\pi$ for all terms t_1 and t_2. So to remove $s \to t$ from $\mathcal{P}$, now it suffices if

[8] E.g., by Thm. 7 one could switch to termination methods like [24] exploiting sorts.

[9] One cannot assume that DP problems $(\mathcal{P}, \mathcal{R})$ always have a specific form, e.g., that $\mathcal{P}$ includes $\mathsf{A} \to \mathsf{A}$ whenever $\mathcal{R}$ includes $\mathsf{a} \to \mathsf{a}$. The reason is that a DP problem $(\mathcal{P}, \mathcal{R})$ can result from arbitrary DP processors that were applied before. Hence, one really has to make sure that processors are sound for *arbitrary* DP problems $(\mathcal{P}, \mathcal{R})$.

for every normal substitution σ where $t\sigma$ is a well-typed ground term, every reduction "$t\sigma \xrightarrow{\mathrm{i}}{}^{!}_{\mathcal{R}} q$" has the form

$$t\sigma \xrightarrow{\mathrm{i}}{}^{*}_{\mathcal{R}} s[\ell\delta]_\pi \xrightarrow{\mathrm{i}}_{\mathcal{R}} s[r\delta]_\pi \xrightarrow{\mathrm{i}}{}^{!}_{\mathcal{R}} q \tag{17}$$

for a rule $\ell \to r \in \mathcal{R}$ where $\ell \succ r$
and where the position π in s is strongly monotonic w.r.t. $\succ$.[10]

For example, for $\mathcal{R}_{sort}$'s termination proof one may use a reduction pair $(\succsim, \succ)$ based on a *polynomial interpretation* [9,23]. A polynomial interpretation $\mathcal{P}ol$ maps every n-ary function symbol f to a polynomial $f_{\mathcal{P}ol}$ over n variables $x_1, \ldots, x_n$ with coefficients from $\mathbb{N}$. This mapping is extended to terms by $[x]_{\mathcal{P}ol} = x$ for all variables x and $[f(t_1, \ldots, t_n)]_{\mathcal{P}ol} = f_{\mathcal{P}ol}([t_1]_{\mathcal{P}ol}, \ldots, [t_n]_{\mathcal{P}ol})$. Now $s \succ_{\mathcal{P}ol} t$ (resp. $s \succsim_{\mathcal{P}ol} t$) iff $[s]_{\mathcal{P}ol} > [t]_{\mathcal{P}ol}$ (resp. $[s]_{\mathcal{P}ol} \geq [t]_{\mathcal{P}ol}$) holds for all instantiations of the variables with natural numbers. For instance, consider the interpretation $\mathcal{P}ol_1$ with

$\mathsf{0}_{\mathcal{P}ol_1} = \mathsf{nil}_{\mathcal{P}ol_1} = \mathsf{true}_{\mathcal{P}ol_1} = \mathsf{false}_{\mathcal{P}ol_1} = \mathsf{ge}_{\mathcal{P}ol_1} = \mathsf{eq}_{\mathcal{P}ol_1} = 0$ $\quad \mathsf{s}_{\mathcal{P}ol_1} = 1 + x_1$
$\mathsf{co}_{\mathcal{P}ol_1} = 1 + x_1 + x_2$ $\quad \mathsf{max}_{\mathcal{P}ol_1} = x_1$
$\mathsf{if}_{1\mathcal{P}ol_1} = 1 + x_2 + x_3 + x_4$ $\quad \mathsf{del}_{\mathcal{P}ol_1} = x_2$
$\mathsf{if}_{2\mathcal{P}ol_1} = 1 + x_3 + x_4$ $\quad \mathsf{SORT}_{\mathcal{P}ol_1} = x_1$

When using the reduction pair $(\succsim_{\mathcal{P}ol_1}, \succ_{\mathcal{P}ol_1})$, the DP (11) and all rules of $\mathcal{R}'_{sort}$ are weakly decreasing. Moreover, then Condition (17) is indeed satisfied for the right-hand side t of (11). To see this, note that in *every* reduction $t\sigma \xrightarrow{\mathrm{i}}{}^{!}_{\mathcal{R}} q$ where $t\sigma$ is a well-typed ground term, eventually one has to apply the rule "$\mathsf{if}_2(\mathsf{true}, x, y, xs) \to xs$" which is strictly decreasing w.r.t. $\succ_{\mathcal{P}ol_1}$. This rule is used by the del-algorithm to delete an element, i.e., to reduce the length of the list. Moreover, the rule is used within a context of the form $\mathsf{SORT}(\mathsf{co}(..., \mathsf{co}(..., \quad \ldots \quad \mathsf{co}(..., \Box))))$. Note that the polynomial $\mathsf{SORT}_{\mathcal{P}ol_1}$ resp. $\mathsf{co}_{\mathcal{P}ol_1}$ is strongly monotonic in its first resp. second argument. Thus, the strictly decreasing rule is indeed used on a strongly monotonic position.

To check automatically whether every reduction of $t\sigma$ to normal form uses a strictly decreasing rule on a strongly monotonic position, we add new rules and function symbols to the TRS $\mathcal{R}$ which results in an extended TRS $\mathcal{R}^{\succ}$. Moreover, for every term u we define a corresponding term $u^{\succ}$. For non-overlapping TRSs $\mathcal{R}$, we have the following property, cf. Lemma 10: if $u^{\succ} \xrightarrow{\mathrm{i}}{}^{*}_{\mathcal{R}^{\succ}} \mathsf{tt}$, then for every reduction $u \xrightarrow{\mathrm{i}}{}^{!}_{\mathcal{R}} q$, we have $u \succ q$. We now explain how to construct $\mathcal{R}^{\succ}$.

[10] In special cases, condition (17) can be automated by k-times *narrowing* the DP $s \to t$ [14]. However, this only works if for any substitution σ, the reduction $t\sigma \xrightarrow{\mathrm{i}}{}^{*}_{\mathcal{R}} s[\ell\delta]_\pi$ is shorter than a fixed number k. So it fails for TRSs like $\mathcal{R}_{sort}$ where termination relies on an inductive property. Here, the reduction

$$\mathsf{SORT}(\mathsf{del}(\mathsf{max}(\mathsf{co}(x, xs)), \mathsf{co}(x, xs)))\sigma \xrightarrow{\mathrm{i}}{}^{*}_{\mathcal{R}_{sort}} \mathsf{SORT}(\mathsf{if}_2(\mathsf{true}, \ldots, \ldots))$$

can be arbitrarily long, depending on σ. Therefore, narrowing the DP (11) a fixed number of times does not help.

For every $f \in \mathcal{D}_\mathcal{R}$, we introduce a new symbol $f^\succ$. Now $f^\succ(u_1, ..., u_n)$ should reduce to tt in the new TRS $\mathcal{R}^\succ$ whenever the reduction of $f(u_1, ..., u_n)$ in the original TRS $\mathcal{R}$ uses a strictly decreasing rule on a strongly monotonic position. Thus, if a rule $f(\ell_1, ..., \ell_n) \to r$ of $\mathcal{R}$ was strictly decreasing (i.e., $f(\ell_1, ..., \ell_n) \succ r$), then we add the rule $f^\succ(\ell_1, ..., \ell_n) \to \mathsf{tt}$ in $\mathcal{R}^\succ$. Otherwise, a strictly decreasing rule will be used on a strongly monotonic position to reduce an instance of $f(\ell_1, \ldots, \ell_n)$ if this holds for the corresponding instance of the right-hand side r. Hence, then we add the rule $f^\succ(\ell_1, ..., \ell_n) \to r^\succ$ in $\mathcal{R}^\succ$ instead. It remains to define $u^\succ$ for any term u over the signature of $\mathcal{R}$. If $u = f(u_1, ..., u_n)$, then we regard the subterms on the strongly monotonic positions of u and check whether their reduction uses a strictly decreasing rule. For any n-ary symbol f, let $mon_\succ(f)$ contain those positions from $\{1, \ldots, n\}$ where the term $f(x_1, ..., x_n)$ is strongly monotonic. If $mon_\succ(f) = \{i_1, \ldots, i_m\}$, then for $u = f(u_1, ..., u_n)$ we obtain $u^\succ = u_{i_1}^\succ \vee \ldots \vee u_{i_m}^\succ$, if f is a constructor. If f is defined, then a strictly decreasing rule could also be applied on the root position of u. Hence, then we have $u^\succ = u_{i_1}^\succ \vee \ldots \vee u_{i_m}^\succ \vee f^\succ(u_1, ..., u_n)$. Of course, $\mathcal{R}^\succ$ also contains appropriate rules for the disjunction "$\vee$".[11] The empty disjunction is represented by ff.

Definition 9 ($\mathcal{R}^\succ$). *Let $\succ$ be an order on terms and let $\mathcal{R}$ be a TRS over $\mathcal{F}$ and $\mathcal{V}$. We extend $\mathcal{F}$ to a new signature $\mathcal{F}^\succ = \mathcal{F} \uplus \{f^\succ \mid f \in \mathcal{D}_\mathcal{R}\} \uplus \{\mathsf{tt}, \mathsf{ff}, \vee\}$. For any term u over $\mathcal{F}$ and $\mathcal{V}$, we define the term $u^\succ$ over $\mathcal{F}^\succ$ and $\mathcal{V}$:*

$$u^\succ = \begin{cases} \bigvee_{i \in mon_\succ(f)} u_i^\succ, & \text{if } u = f(u_1, \ldots, u_n) \text{ and } f \notin \mathcal{D}_\mathcal{R} \\ \bigvee_{i \in mon_\succ(f)} u_i^\succ \quad \vee \quad f^\succ(u_1, \ldots, u_n), & \text{if } u = f(u_1, \ldots, u_n) \text{ and } f \in \mathcal{D}_\mathcal{R} \\ \mathsf{ff}, & \text{if } u \in \mathcal{V} \end{cases}$$

$$\begin{aligned} \text{Moreover, we define } \mathcal{R}^\succ = \quad & \{f^\succ(\ell_1, ..., \ell_n) \to \mathsf{tt} \mid f(\ell_1, ..., \ell_n) \to r \in \mathcal{R} \cap \succ\} \\ & \cup \{f^\succ(\ell_1, ..., \ell_n) \to r^\succ \mid f(\ell_1, ..., \ell_n) \to r \in \mathcal{R} \setminus \succ\} \\ & \cup \mathcal{R} \cup \{\mathsf{tt} \vee b \to \mathsf{tt},\ \mathsf{ff} \vee b \to b\}. \end{aligned}$$

In our example, the only rules of $\mathcal{R}'_{sort}$ which are strictly decreasing w.r.t. $\succ_{\mathcal{P}ol_1}$ are the last two max-rules and the rule "$\mathsf{if}_2(\mathsf{true}, x, y, xs) \to xs$". So according to Def. 9, the TRS $\mathcal{R}'^{\succ_{\mathcal{P}ol_1}}_{sort}$ contains $\mathcal{R}'_{sort} \cup \{\mathsf{tt} \vee b \to \mathsf{tt},\ \mathsf{ff} \vee b \to b\}$ and the following rules. Here, we already simplified disjunctions of the form "$\mathsf{ff} \vee t$" or "$t \vee \mathsf{ff}$" to t. To ease readability, we wrote "$\mathsf{ge}^\succ$" instead of "$\mathsf{ge}^{\succ_{\mathcal{P}ol_1}}$", etc.

$$\begin{aligned} \mathsf{ge}^\succ(x, 0) &\to \mathsf{ff} \\ \mathsf{ge}^\succ(0, \mathsf{s}(y)) &\to \mathsf{ff} \\ \mathsf{ge}^\succ(\mathsf{s}(x), \mathsf{s}(y)) &\to \mathsf{ge}^\succ(x, y) \\ \mathsf{max}^\succ(\mathsf{nil}) &\to \mathsf{ff} \\ \mathsf{max}^\succ(\mathsf{co}(x, \mathsf{nil})) &\to \mathsf{tt} \\ \mathsf{max}^\succ(\mathsf{co}(x, \mathsf{co}(y, xs))) &\to \mathsf{tt} \\ \mathsf{del}^\succ(x, \mathsf{nil}) &\to \mathsf{ff} \\ \mathsf{del}^\succ(x, \mathsf{co}(y, xs)) &\to \mathsf{if}_2^\succ(\mathsf{eq}(x, y), x, y, xs) \end{aligned}$$

$$\begin{aligned} \mathsf{eq}^\succ(0, 0) &\to \mathsf{ff} \\ \mathsf{eq}^\succ(\mathsf{s}(x), 0) &\to \mathsf{ff} \\ \mathsf{eq}^\succ(0, \mathsf{s}(y)) &\to \mathsf{ff} \\ \mathsf{eq}^\succ(\mathsf{s}(x), \mathsf{s}(y)) &\to \mathsf{eq}^\succ(x, y) \\ \mathsf{if}_1^\succ(\mathsf{true}, x, y, xs) &\to \mathsf{max}^\succ(\mathsf{co}(x, xs)) \\ \mathsf{if}_1^\succ(\mathsf{false}, x, y, xs) &\to \mathsf{max}^\succ(\mathsf{co}(y, xs)) \\ \mathsf{if}_2^\succ(\mathsf{true}, x, y, xs) &\to \mathsf{tt} \\ \mathsf{if}_2^\succ(\mathsf{false}, x, y, xs) &\to \mathsf{del}^\succ(x, xs) \end{aligned}$$

[11] It suffices to include just the rules "$\mathsf{tt} \vee b \to \mathsf{tt}$" and "$\mathsf{ff} \vee b \to b$", since $\mathcal{R}^\succ$ is only used for inductive proofs and "$b \vee \mathsf{tt} = \mathsf{tt}$" and "$b \vee \mathsf{ff} = b$" are inductive consequences.

Lemma 10 (Soundness of $\mathcal{R}^\succ$). *Let $(\succsim, \succ)$ be a reduction pair and let $\mathcal{R}$ be a non-overlapping TRS over $\mathcal{F}$ and $\mathcal{V}$ with $\mathcal{R} \subseteq \succsim$. For any terms u and q over $\mathcal{F}$ and $\mathcal{V}$ with $u^\succ \xrightarrow{\mathrm{i}}{}^*_{\mathcal{R}^\succ} \mathsf{tt}$ and $u \xrightarrow{\mathrm{i}}{}^!_{\mathcal{R}} q$, we have $u \succ q$.*

Proof. We use induction on the lexicographic combination of the length of the reduction $u \xrightarrow{\mathrm{i}}{}^!_{\mathcal{R}} q$ and of the structure of u.

First let u be a variable. Here, $u^\succ = \mathsf{ff}$ and thus, $u^\succ \xrightarrow{\mathrm{i}}{}^*_{\mathcal{R}^\succ} \mathsf{tt}$ is impossible.

Now let $u = f(u_1, ..., u_n)$. The reduction $u \xrightarrow{\mathrm{i}}{}^!_{\mathcal{R}} q$ starts with $u = f(u_1, ..., u_n) \xrightarrow{\mathrm{i}}{}^*_{\mathcal{R}^\succ} f(q_1, ..., q_n)$ where the reductions $u_i \xrightarrow{\mathrm{i}}{}^!_{\mathcal{R}} q_i$ are at most as long as $u \xrightarrow{\mathrm{i}}{}^!_{\mathcal{R}} q$. If there is a $j \in mon_\succ(f)$ with $u_j^\succ \xrightarrow{\mathrm{i}}{}^*_{\mathcal{R}^\succ} \mathsf{tt}$, then $u_j \succ q_j$ by induction hypothesis. So $u = f(u_1, ..., u_j, ..., u_n) \succ f(u_1, ..., q_j, ..., u_n) \succsim f(q_1, ..., q_j, ..., q_n) \succsim q$, as $\mathcal{R} \subseteq \succsim$.

Otherwise, $u^\succ \xrightarrow{\mathrm{i}}{}^*_{\mathcal{R}^\succ} \mathsf{tt}$ means that $f^\succ(u_1, \ldots, u_n) \xrightarrow{\mathrm{i}}{}^*_{\mathcal{R}^\succ} \mathsf{tt}$. As $\mathcal{R} \subseteq \mathcal{R}^\succ$, we have $f^\succ(u_1, \ldots, u_n) \xrightarrow{\mathrm{i}}{}^*_{\mathcal{R}^\succ} f^\succ(q_1, \ldots, q_n)$. Since $\mathcal{R}$ is non-overlapping, $\mathcal{R}^\succ$ is non-overlapping as well. This implies confluence of $\xrightarrow{\mathrm{i}}{}_{\mathcal{R}^\succ}$, cf. [29]. Hence, we also get $f^\succ(q_1, \ldots, q_n) \xrightarrow{\mathrm{i}}{}^*_{\mathcal{R}^\succ} \mathsf{tt}$. There is a rule $f(\ell_1, \ldots, \ell_n) \to r \in \mathcal{R}$ and a normal substitution δ with $f^\succ(q_1, \ldots, q_n) = f(\ell_1, \ldots, \ell_n)\delta \xrightarrow{\mathrm{i}}{}_{\mathcal{R}} r\delta \xrightarrow{\mathrm{i}}{}^!_{\mathcal{R}} q$. Note that the q_i only contain symbols of $\mathcal{F}$. Thus, as the q_i are normal forms w.r.t. $\mathcal{R}$, they are also normal forms w.r.t. $\mathcal{R}^\succ$. Therefore, as $\mathcal{R}^\succ$ is non-overlapping, the only rule of $\mathcal{R}^\succ$ applicable to $f^\succ(q_1, \ldots, q_n)$ is the one resulting from $f(\ell_1, \ldots, \ell_n) \to r \in \mathcal{R}$. If $f(\ell_1, \ldots, \ell_n) \succ r$, then that rule would be "$f^\succ(\ell_1, \ldots, \ell_n) \to \mathsf{tt}$" and

$$u = f(u_1, \ldots, u_n) \succsim f(q_1, \ldots, q_n) = f(\ell_1, \ldots, \ell_n)\delta \succ r\delta \succsim q.$$

Otherwise, the rule is "$f^\succ(\ell_1, ..., \ell_n) \to r^\succ$", i.e., $f^\succ(q_1, ..., q_n) = f^\succ(\ell_1, ..., \ell_n)\delta \xrightarrow{\mathrm{i}}{}_{\mathcal{R}^\succ} r^\succ\delta \xrightarrow{\mathrm{i}}{}^*_{\mathcal{R}^\succ} \mathsf{tt}$. Since the reduction $r\delta \xrightarrow{\mathrm{i}}{}^!_{\mathcal{R}} q$ is shorter than the original reduction $u \xrightarrow{\mathrm{i}}{}^!_{\mathcal{R}} q$, the induction hypothesis implies $r\delta \succ q$. Thus,

$$u = f(u_1, \ldots, u_n) \succsim f(q_1, \ldots, q_n) = f(\ell_1, \ldots, \ell_n)\delta \succsim r\delta \succ q. \qquad \square$$

With Lemma 10, the condition (17) needed to remove a DP from a DP problem can again be reformulated. To remove $s \to t$ from $\mathcal{P}$, now it suffices if

$$\text{for every normal substitution } \sigma \text{ where } t\sigma \text{ is a well-typed ground term, we have } t^\succ\sigma \xrightarrow{\mathrm{i}}{}^*_{\mathcal{R}^\succ} \mathsf{tt}. \tag{18}$$

So in our example, to remove the DP (11) using the reduction pair $(\succsim_{\mathcal{P}ol_1}, \succ_{\mathcal{P}ol_1})$, we require "$t^{\succ_{\mathcal{P}ol_1}}\sigma \xrightarrow{\mathrm{i}}{}^*_{\mathcal{R}'^{\succ_{\mathcal{P}ol_1}}_{sort}} \mathsf{tt}$", where t is the right-hand side of (11), i.e., $t = \mathsf{SORT}(\mathsf{del}(\mathsf{max}(\mathsf{co}(x, xs)), \mathsf{co}(x, xs)))$. Since $mon_{\succ_{\mathcal{P}ol_1}}(\mathsf{SORT}) = \{1\}$, $mon_{\succ_{\mathcal{P}ol_1}}(\mathsf{del}) = \{2\}$, $mon_{\succ_{\mathcal{P}ol_1}}(\mathsf{co}) = \{1, 2\}$, and $x^{\succ_{\mathcal{P}ol_1}} = xs^{\succ_{\mathcal{P}ol_1}} = \mathsf{ff}$, $t^{\succ_{\mathcal{P}ol_1}}$ is $\mathsf{del}^{\succ_{\mathcal{P}ol_1}}(\mathsf{max}(\mathsf{co}(x, xs)), \mathsf{co}(x, xs))$ when simplifying disjunctions with ff. So to remove (11), we require the following for all normal substitutions σ where $t\sigma$ is well typed and ground.[12]

$$\mathsf{del}^{\succ_{\mathcal{P}ol_1}}(\mathsf{max}(\mathsf{co}(x, xs)), \mathsf{co}(x, xs))\sigma \xrightarrow{\mathrm{i}}{}^*_{\mathcal{R}'^{\succ_{\mathcal{P}ol_1}}_{sort}} \mathsf{tt} \tag{19}$$

[12] Note that the restriction to *well-typed ground terms* is crucial. Indeed, (19) does not hold for non-ground or non-well-typed substitutions like $\sigma(x) = \sigma(xs) = \mathsf{true}$.

Note that the rules for $\mathsf{del}^{\succ_{Pol_1}}$ (given before Lemma 10) compute the *member*-function. In other words, $\mathsf{del}^{\succ_{Pol_1}}(x, xs)$ holds iff x occurs in the list xs. Thus, (19) is equivalent to the main termination argument (1) of Ex. 1, i.e., to the observation that every non-empty list contains its maximum. Thus, now we can detect and express termination arguments like (1) within the DP framework.

Our goal is to use inductive theorem provers to verify arguments like (1) or, equivalently, to verify conditions like (18). Indeed, (18) corresponds to the question whether a suitable conjecture is *inductively valid* [4,5,7,8,20,21,33,34,36].

Definition 11 (Inductive Validity). *Let $\mathcal{R}$ be a TRS and let s, t be terms over $\mathcal{F}$ and $\mathcal{V}$. We say that $t = s$ is* inductively valid *in $\mathcal{R}$ (denoted $\mathcal{R} \models_{ind} t = s$) iff there exist typed variants $\mathcal{F}'$ and $\mathcal{V}'$ such that $\mathcal{R}$, t, s are well typed where t and s have the same type, and such that $t\sigma \overset{\mathsf{i}}{\leftrightarrow}^*_{\mathcal{R}} s\sigma$ holds for all substitutions σ over $\mathcal{F}'$ where $t\sigma$ and $s\sigma$ are well-typed ground terms. To make the specific typed variants explicit, we also write "$\mathcal{R} \models_{ind}^{\mathcal{F}',\mathcal{V}'} t = s$".*

Of course, in general $\mathcal{R} \models_{ind} t = s$ is undecidable, but it can often be proved automatically by *inductive theorem provers*. By reformulating Condition (18), we now obtain that in a DP problem $(\mathcal{P}, \mathcal{R})$, $s \to t$ can be removed from $\mathcal{P}$ if

$$\mathcal{R}^{\succ} \models_{ind} t^{\succ} = \mathsf{tt}. \tag{20}$$

Of course, in addition all DPs $\mathcal{P}$ and all rules $\mathcal{R}$ have to be weakly decreasing.

Now we formulate a new DP processor based on Condition (20). Recall that to derive (20) we required a non-overlappingness condition and innermost termination of $\mathcal{R}$. (These requirements ensure that it suffices to regard only *ground* instantiations when proving that reductions of $t\sigma$ to normal form are strictly decreasing, cf. Ex. 8. Moreover, non-overlappingness is needed for Lemma 10 to make sure that $t^{\succ}\sigma \overset{\mathsf{i}}{\to}^*_{\mathcal{R}^{\succ}} \mathsf{tt}$ really guarantees that *all* reductions of $t\sigma$ to normal form are strictly decreasing. Non-overlappingness also ensures that $t^{\succ}\sigma \overset{\mathsf{i}}{\to}^*_{\mathcal{R}^{\succ}} \mathsf{tt}$ in Condition (18) is equivalent to $t^{\succ}\sigma \overset{\mathsf{i}}{\leftrightarrow}^*_{\mathcal{R}^{\succ}} \mathsf{tt}$ in Condition (20).)

To ensure innermost termination of $\mathcal{R}$, the following processor transforms $(\mathcal{P}, \mathcal{R})$ not only into the new DP problem $(\mathcal{P} \setminus \{s \to t\}, \mathcal{R})$, but it also generates the problem $(DP(\mathcal{R}), \mathcal{R})$. Absence of infinite innermost $(DP(\mathcal{R}), \mathcal{R})$-chains is equivalent to innermost termination of $\mathcal{R}$. Note that in practice $\mathcal{R}$ only contains the usable rules of $\mathcal{P}$ (since one should have applied the usable rule processor of Thm. 3 before). Then the DP problem $(DP(\mathcal{R}), \mathcal{R})$ means that the TRS consisting just of the usable rules must be innermost terminating. An application of the dependency graph processor of Thm. 4 will therefore transform $(DP(\mathcal{R}), \mathcal{R})$ into DP problems that have already been generated *before*. So (except for algorithms with nested or mutual recursion), the DP problem $(DP(\mathcal{R}), \mathcal{R})$ obtained by the following processor does not lead to new proof obligations.

In Thm. 12, we restrict ourselves to DP problems $(\mathcal{P}, \mathcal{R})$ with the *tuple property*. This means that for all $s \to t \in \mathcal{P}$, root(s) and root(t) are tuple symbols and tuple symbols neither occur anywhere else in s or t nor in $\mathcal{R}$. This is always satisfied for the initial DP problem and it is maintained by almost all DP processors in the literature (including all processors of this paper).

Theorem 12 (Induction Processor). *Let* $(\succsim,\succ)$ *be a reduction pair, let* $(\mathcal{P},\mathcal{R})$ *have the tuple property, let* $\mathcal{R}$ *be non-overlapping and let there be no critical pairs between* $\mathcal{R}$ *and* $\mathcal{P}$.[13] *Let* $\mathcal{F}'$, $\mathcal{V}'$ *be typed variants of* $\mathcal{P} \cup \mathcal{R}$*'s signature such that* $\mathcal{P} \cup \mathcal{R}$ *is well typed. Then the following DP processor Proc is sound.*

$$Proc(\,(\mathcal{P},\mathcal{R})\,) = \begin{cases} \{\ (\mathcal{P} \setminus \{s \to t\},\, \mathcal{R}),\ \ (DP(\mathcal{R}),\, \mathcal{R})\ \} & \text{if } \mathcal{R}^{\succ} \models_{ind}^{\mathcal{F}',\mathcal{V}'} t^{\succ} = \mathtt{tt} \\ & \text{and } \mathcal{P} \cup \mathcal{R} \subseteq \succsim \\ \{\,(\mathcal{P},\mathcal{R})\,\}, & \text{otherwise} \end{cases}$$

Proof. Suppose there is an infinite innermost $(\mathcal{P},\mathcal{R})$-chain, i.e., $\mathcal{P} \cup \mathcal{R}$ is not innermost terminating. By persistence of innermost termination (Thm. 7), there is a well-typed term that is not innermost terminating w.r.t. $\mathcal{P} \cup \mathcal{R}$. Let q be a minimal such term (i.e., q's proper subterms are innermost terminating). Due to the tuple property, w.l.o.g. q either contains no tuple symbol or q contains a tuple symbol only at the root. In the first case, only $\mathcal{R}$-rules can reduce q. Thus, $\mathcal{R}$ is not innermost terminating and there is an infinite innermost $(DP(\mathcal{R}),\, \mathcal{R})$-chain.

Now let $\mathcal{R}$ be innermost terminating. So root(q) is a tuple symbol and q contains no further tuple symbol. Hence, in q's infinite innermost $\mathcal{P} \cup \mathcal{R}$-reduction, $\mathcal{R}$-rules are only applied below the root and $\mathcal{P}$-rules are only applied on the root position. Moreover, there are infinitely many $\mathcal{P}$-steps. Hence, this infinite reduction corresponds to an infinite innermost $(\mathcal{P},\mathcal{R})$-chain $s_1 \to t_1, s_2 \to t_2, \ldots$ where $t_i\sigma \xrightarrow{\mathrm{i}}_{\mathcal{R}}^{!} s_{i+1}\sigma$ for all i and all occurring terms are well typed.

Next we show that due to innermost termination of $\mathcal{R}$, there is even an infinite innermost $(\mathcal{P},\mathcal{R})$-chain on well-typed *ground* terms. Let δ instantiate all variables in $s_1\sigma$ by ground terms of the corresponding sort. (Recall that in any typed variant there are such ground terms.) We define the normal substitution σ' such that $\sigma'(x)$ is the $\mathcal{R}$-normal form of $x\sigma\delta$ for all variables x. This normal form must exist since $\mathcal{R}$ is innermost terminating and it is unique since $\mathcal{R}$ is non-overlapping. Clearly, $t_i\sigma \xrightarrow{\mathrm{i}}_{\mathcal{R}}^{*} s_{i+1}\sigma$ implies $t_i\sigma\delta \to_{\mathcal{R}}^{*} s_{i+1}\sigma\delta$, i.e., $t_i\sigma' \to_{\mathcal{R}}^{*} s_{i+1}\sigma'$. As left-hand sides s_i of DPs do not overlap with rules of $\mathcal{R}$, all $s_i\sigma'$ are in normal form. Due to non-overlappingness of $\mathcal{R}$, $s_{i+1}\sigma'$ is the *only* normal form of $t_i\sigma'$ and thus, it can also be reached by innermost steps, i.e., $t_i\sigma' \xrightarrow{\mathrm{i}}_{\mathcal{R}}^{!} s_{i+1}\sigma'$. Hence, there is an infinite innermost $(\mathcal{P},\mathcal{R})$-chain on well-typed *ground* terms.

If this chain does not contain infinitely many variable-renamed copies of the DP $s \to t$, then its tail is an infinite innermost $(\mathcal{P}\setminus\{s \to t\},\, \mathcal{R})$-chain. Otherwise, $s_{i_1} \to t_{i_1}, s_{i_2} \to t_{i_2}, \ldots$ are variable-renamed copies of $s \to t$ and thus, $t_{i_1}\sigma'$, $t_{i_2}\sigma'$, $\ldots$ are well-typed ground instantiations of t. As $\mathcal{R}^{\succ} \models_{ind} t^{\succ} = \mathtt{tt}$, we have $(t_{i_j}\sigma')^{\succ} = t_{i_j}^{\succ}\sigma' \xrightarrow{\mathrm{i}}_{\mathcal{R}^{\succ}}^{*} \mathtt{tt}$ for all j. Since $t_{i_j}\sigma' \xrightarrow{\mathrm{i}}_{\mathcal{R}}^{!} s_{i_j+1}\sigma'$, Lemma 10 implies $t_{i_j}\sigma' \succ s_{i_j+1}\sigma'$ for all (infinitely many) j. Moreover, $s_i\sigma' \succsim t_i\sigma'$ and $t_i\sigma' \succsim s_{i+1}\sigma'$ for all i, since $\mathcal{P} \cup \mathcal{R} \subseteq \succsim$. This contradicts the well-foundedness of $\succ$. □

[13] More precisely, for all $v \to w$ in $\mathcal{P}$, non-variable subterms of v may not unify with left-hand sides of rules from $\mathcal{R}$ (after variable renaming).

In our example, we ended up with the DP problem $(\{(11)\}, \mathcal{R}'_{sort})$. To remove the DP (11) from the DP problem, we use an inductive theorem prover to prove

$$\mathcal{R}'^{\succ_{\mathcal{P}ol_1}}_{sort} \models_{ind} \mathsf{del}^{\succ_{\mathcal{P}ol_1}}(\mathsf{max}(\mathsf{co}(x, xs)), \mathsf{co}(x, xs)) = \mathsf{tt}, \tag{21}$$

i.e., that every non-empty list contains its maximum. The tuple property and the non-overlappingness requirements in Thm. 12 are clearly fulfilled. Moreover, all rules decrease w.r.t. $\succsim_{\mathcal{P}ol_1}$. Hence, the induction processor results in the trivial problem $(\varnothing, \mathcal{R}'_{sort})$ and the problem $(DP(\mathcal{R}'_{sort}), \mathcal{R}'_{sort}) = (\{(2), ..., (10)\}, \mathcal{R}'_{sort})$. The dependency graph processor transforms the latter problem into the problems $(\mathcal{P}_i, \mathcal{R}'_{sort})$ with $1 \leq i \leq 4$ that had already been solved before, cf. Sect. 2. For example, the induction prover in AProVE proves (21) automatically and thus, it can easily perform the above proof and verify termination of the TRS $\mathcal{R}_{sort}$.

5 Experiments and Conclusion

We introduced a new processor in the DP framework which can handle TRSs that terminate because of inductive properties of their algorithms. This processor automatically tries to extract these properties and transforms them into conjectures which are passed to an inductive theorem prover for verification. To obtain a powerful method, we showed that it suffices to prove these conjectures only for well-typed terms, even though the original TRSs under examination are untyped.

We implemented the new processor of Thm. 12 in our termination tool AProVE [13] and coupled it with the small inductive theorem prover that was already available in AProVE. To automate Thm. 12, AProVE selects a DP $s \to t$ and searches for a reduction pair $(\succsim, \succ)$ which orients at least one rule of $\mathcal{U}(t)$ strictly (on a strongly monotonic position). Then AProVE tests if $t^{\succ} = \mathsf{tt}$ is inductively valid. So in contrast to previous approaches that use inductive theorem provers for termination analysis (cf. Sect. 1), our automation can search for arbitrary reduction pairs instead of being restricted to a fixed small set of orders. The search for the reduction pair is guided by the fact that there has to be a strictly decreasing usable rule on a strongly monotonic position.

To demonstrate the power of our method, [1] features a collection of 19 typical TRSs where an inductive argument is needed for the termination proof. This collection contains several TRSs computing classical arithmetical algorithms as well as many TRSs with standard algorithms for list manipulation like sorting, reversing, etc. The previous version of AProVE was the most powerful tool for termination of term rewriting at the *International Competition of Termination Provers*. Nevertheless, this previous AProVE version as well as all other tools in the competition failed on all of these examples. In contrast, with a time limit of 60 seconds per example, our new version of AProVE automatically proves termination of 16 of them. At the same time, the new version of AProVE is as successful as the previous one on the remaining examples of the *Termination Problem Data Base*, which is the collection of examples used in the termination competition. Thus, the present paper is a substantial advance in automated

termination proving, since it allows the first combination of powerful TRS termination tools with inductive theorem provers. For details on our experiments and to access our implementation via a web-interface, we refer to [1].

Acknowledgements. We are very grateful to Aart Middeldorp and Hans Zantema for suggesting the proof idea of Thm. 7 and for pointing us to [30].

References

1. AProVE website, `http://aprove.informatik.rwth-aachen.de/eval/Induction/`
2. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theoretical Computer Science 236, 133–178 (2000)
3. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge (1998)
4. Bouhoula, A., Rusinowitch, M.: SPIKE: A system for automatic inductive proofs. In: Alagar, V.S., Nivat, M. (eds.) AMAST 1995. LNCS, vol. 936, pp. 576–577. Springer, Heidelberg (1995)
5. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic Press, London (1979)
6. Brauburger, J., Giesl, J.: Termination analysis by inductive evaluation. In: Kirchner, C., Kirchner, H. (eds.) CADE 1998. LNCS (LNAI), vol. 1421, pp. 254–269. Springer, Heidelberg (1998)
7. Bundy, A.: The automation of proof by mathematical induction. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 1, pp. 845–911. Elsevier & MIT (2001)
8. Comon, H.: Inductionless induction. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 1, pp. 913–962. Elsevier & MIT (2001)
9. Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: SAT solving for termination analysis with polynomial interpretations. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 340–354. Springer, Heidelberg (2007)
10. Giesl, J.: Termination analysis for functional programs using term orderings. In: Mycroft, A. (ed.) SAS 1995. LNCS, vol. 983, pp. 154–171. Springer, Heidelberg (1995)
11. Giesl, J., Thiemann, R., Schneider-Kamp, P.: The DP framework: Combining techniques for automated termination proofs. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 301–331. Springer, Heidelberg (2005)
12. Giesl, J., Swiderski, S., Schneider-Kamp, P., Thiemann, R.: Automated termination analysis for Haskell: From term rewriting to programming languages. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 297–312. Springer, Heidelberg (2006)
13. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic termination proofs in the DP framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006)
14. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. Journal of Automated Reasoning 37(3), 155–203 (2006)
15. Giesl, J., Thiemann, R., Swiderski, S., Schneider-Kamp, P.: Proving termination by bounded increase. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 443–459. Springer, Heidelberg (2007)
16. Gnaedig, I., Kirchner, H.: Termination of rewriting under strategies. ACM Transactions on Computational Logic 10(3) (2008)
17. Gramlich, B.: Abstract relations between restricted termination and confluence properties of rewrite systems. Fundamenta Informaticae 24(1-2), 2–23 (1995)

18. Hirokawa, N., Middeldorp, A.: Automating the dependency pair method. Information and Computation 199(1-2), 172–199 (2005)
19. Hirokawa, N., Middeldorp, A.: Tyrolean Termination Tool: Techniques and features. Information and Computation 205(4), 474–511 (2007)
20. Kapur, D., Musser, D.R.: Proof by consistency. Artif. Int. 31(2), 125–157 (1987)
21. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer, Dordrecht (2000)
22. Krauss, A.: Certified size-change termination. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 460–475. Springer, Heidelberg (2007)
23. Lankford, D.: On proving term rewriting systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA (1979)
24. Lucas, S., Meseguer, J.: Order-sorted dependency pairs. In: Proc. PPDP 2008, pp. 108–119. ACM Press, New York (2008)
25. Manolios, P., Vroon, D.: Termination analysis with calling context graphs. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 401–414. Springer, Heidelberg (2006)
26. Middeldorp, A., Zantema, H.: Personal communication (2008)
27. Ohlebusch, E.: Termination of logic programs: Transformational approaches revisited. Appl. Algebra in Engineering, Comm. and Computing 12, 73–116 (2001)
28. Panitz, S.E., Schmidt-Schauß, M.: TEA: Automatically proving termination of programs in a non-strict higher-order functional language. In: Van Hentenryck, P. (ed.) SAS 1997. LNCS, vol. 1302, pp. 345–360. Springer, Heidelberg (1997)
29. Plaisted, D.A.: Equational reasoning and term rewriting systems. In: Gabbay, D.M., Hogger, C.J., Robinson, J.A. (eds.) Handbook of Logic in Artificial Intelligence and Logic Programming, Oxford, vol. 1, pp. 273–364 (1993)
30. van de Pol, J.: Modularity in many-sorted term rewriting. Master's Thesis, Utrecht University (1992), http://homepages.cwi.nl/~vdpol/papers/
31. Schneider-Kamp, P., Giesl, J., Serebrenik, A., Thiemann, R.: Automated termination proofs for logic programs by term rewriting. ACM Transactions on Computational Logic (to appear, 2009)
32. Walther, C.: On proving the termination of algorithms by machine. Artificial Intelligence 71(1), 101–157 (1994)
33. Walther, C.: Mathematical induction. In: Gabbay, D., Hogger, C.J., Robinson, J.A. (eds.) Handbook of Logic in AI and Logic Programming, Oxford, vol. 2, pp. 127–228 (1994)
34. Walther, C., Schweitzer, S.: About VeriFun. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 322–327. Springer, Heidelberg (2003)
35. Zantema, H.: Termination of term rewriting: Interpretation and type elimination. Journal of Symbolic Computation 17(1), 23–50 (1994)
36. Zhang, H., Kapur, D., Krishnamoorthy, M.S.: A mechanizable induction principle for equational specifications. In: Lusk, E., Overbeek, R. (eds.) CADE 1988. LNCS (LNAI), vol. 310, pp. 162–181. Springer, Heidelberg (1988)

Beyond Dependency Graphs

Martin Korp and Aart Middeldorp

Institute of Computer Science
University of Innsbruck
Austria
{martin.korp,aart.middeldorp}@uibk.ac.at

Abstract. The dependency pair framework is a powerful technique for proving termination of rewrite systems. One of the most frequently used methods within this framework is the dependency graph processor. In this paper we improve this processor by incorporating right-hand sides of forward closures. In combination with tree automata completion we obtain an efficient processor which can be used instead of the dependency graph approximations that are in common use in termination provers.

1 Introduction

Proving termination of term rewrite systems is a very active research area. Several tools exist that perform this task automatically. The most powerful ones are based on the dependency pair framework. This framework combines a great variety of termination techniques in a modular way by means of dependency pair processors. In this paper we are concerned with the dependency graph processor. It is one of the most important processors as it enables the decomposition of termination problems into smaller subproblems. The processor requires the computation of an over-approximation of the dependency graph. In the literature several such approximations are proposed. Arts and Giesl [1] gave an effective algorithm based on abstraction and unification. Kusakari and Toyama [20,21] employed Huet and Lévy's notion of ω-reduction to approximate dependency graphs for AC-termination. Middeldorp [22] advocated the use of tree automata techniques and in [23] improved the approximation of [1] by taking symmetry into account. Giesl, Thiemann, and Schneider-Kamp [12] tightly coupled abstraction and unification, resulting in an improvement of [1] which is especially suited for applicative systems.

In this paper we return to tree automata techniques. We show that tree automata *completion* is much more effective for approximating dependency graphs than the method based on approximating the underlying rewrite system to ensure regularity preservation proposed in [22]. The dependency graph determines whether dependency pairs can follow each other. It does not determine whether dependency pairs follow each other *infinitely often*. We further show that by incorporating *right-hand sides of forward closures* [4,9], a technique that recently became popular in connection with the match-bound technique [8,18], we can

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 339–354, 2009.

eliminate arcs from the (real) dependency graph. Experimental results confirm the competitiveness of the resulting improved dependency graph processor.

The remainder of the paper is organized as follows. In Section 2 we recall some basic facts about dependency graphs and processors. In Section 3 we employ tree automata completion to approximate dependency graphs. Incorporating right-hand sides of forward closures is the topic of Section 4. In Section 5 we compare our approach with existing approximations of dependency graphs. Dependency graphs for innermost termination are briefly addressed in Section 6. In Section 7 we report on the extensive experiments that we conducted.

2 Preliminaries

Familiarity with term rewriting [2] and tree automata [3] is assumed. Knowledge of the dependency pair framework [11,25] and the match-bound technique [8,18] will be helpful. Below we recall important definitions concerning the former needed in the remainder of the paper. Throughout this paper we assume that term rewrite systems are finite.

Let $\mathcal{R}$ be a term rewrite system (TRS for short) over a signature $\mathcal{F}$. If $l \to r \in \mathcal{R}$ and t is a subterm of r with a defined root symbol that is not a proper subterm of l then $l^\sharp \to t^\sharp$ is a *dependency pair* of $\mathcal{R}$. Here $l^\sharp$ and $t^\sharp$ denote the terms that are obtained by marking the root symbols of l and t. In examples we use capital letters to represent marked function symbols. The set of dependency pairs of $\mathcal{R}$ is denoted by $\mathsf{DP}(\mathcal{R})$. A *DP problem* is a triple $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ where $\mathcal{P}$ and $\mathcal{R}$ are two TRSs and $\mathcal{G} \subseteq \mathcal{P} \times \mathcal{P}$ is a directed graph. A DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is called *finite* if there are no infinite rewrite sequences of the form $s_1 \xrightarrow{\epsilon}_{\alpha_1} t_1 \to^*_{\mathcal{R}} s_2 \xrightarrow{\epsilon}_{\alpha_2} t_2 \to^*_{\mathcal{R}} \cdots$ such that all terms $t_1, t_2, \ldots$ are terminating with respect to $\mathcal{R}$ and $(\alpha_i, \alpha_{i+1}) \in \mathcal{G}$ for all $i \geqslant 1$. Such an infinite sequence is said to be *minimal*. The main result underlying the dependency pair approach states that a TRS $\mathcal{R}$ is terminating if and only if the DP problem $(\mathsf{DP}(\mathcal{R}), \mathcal{R}, \mathsf{DP}(\mathcal{R}) \times \mathsf{DP}(\mathcal{R}))$ is finite. The latter is shown by applying functions that take a DP problem as input and return a set of DP problems as output, the so-called *DP processors*. These processors must have the property that a DP problem is finite whenever all DP problems returned by the processor are finite, which is known as *soundness*. To use DP processors for establishing non-termination, they must additionally be *complete* which means that if one of the DP problems returned by the processor is not finite then the original DP problem is not finite.

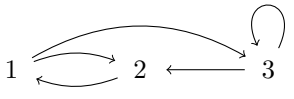
Numerous DP processors have been developed. In this paper we are concerned with the dependency graph processor. This is one of the most important processors as it enables to decompose a DP problem into smaller subproblems. The dependency graph processor determines which dependency pairs can follow each other in infinite rewrite sequences. Following [25], we find it convenient to split the processor into one which computes the graph and one which computes the strongly connected components (SCCs). This separates the part that needs to be approximated from the computable part and is important to properly describe the experiments in Section 7 where we combine several graph approximations before computing SCCs.

Definition 1. *The* dependency graph processor *maps a DP problem* $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ *to* $\{(\mathcal{P}, \mathcal{R}, \mathcal{G} \cap \mathsf{DG}(\mathcal{P}, \mathcal{R}))\}$. *Here* $\mathsf{DG}(\mathcal{P}, \mathcal{R})$ *is the* dependency graph *of* $\mathcal{P}$ *and* $\mathcal{R}$*, which has the rules in* $\mathcal{P}$ *as nodes and there is an arc from* $s \to t$ *to* $u \to v$ *if and only if there exist substitutions* σ *and* τ *such that* $t\sigma \to_{\mathcal{R}}^* u\tau$.

Example 2. Consider the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ with $\mathcal{R}$ consisting of the rewrite rules $\mathsf{f}(\mathsf{g}(x), y) \to \mathsf{g}(\mathsf{h}(x, y))$ and $\mathsf{h}(\mathsf{g}(x), y) \to \mathsf{f}(\mathsf{g}(\mathsf{a}), \mathsf{h}(x, y))$, $\mathcal{P} = \mathsf{DP}(\mathcal{R})$ consisting of

$$1\colon \mathsf{F}(\mathsf{g}(x), y) \to \mathsf{H}(x, y) \qquad 2\colon \mathsf{H}(\mathsf{g}(x), y) \to \mathsf{F}(\mathsf{g}(\mathsf{a}), \mathsf{h}(x, y))$$
$$3\colon \mathsf{H}(\mathsf{g}(x), y) \to \mathsf{H}(x, y)$$

and $\mathcal{G} = \mathcal{P} \times \mathcal{P}$. Because $\mathsf{H}(\mathsf{g}(x), y)$ is an instance of $\mathsf{H}(x, y)$ and $\mathsf{F}(\mathsf{g}(\mathsf{a}), \mathsf{h}(x, y))$ is an instance of $\mathsf{F}(\mathsf{g}(x), y)$, $\mathsf{DG}(\mathcal{P}, \mathcal{R})$ has five arcs:



The dependency graph processor returns the new DP problem $(\mathcal{P}, \mathcal{R}, \mathsf{DG}(\mathcal{P}, \mathcal{R}))$.

Definition 3. *The* SCC processor *transforms a DP problem* $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ *into* $\{(\mathcal{P}_1, \mathcal{R}, \mathcal{G}_1), \ldots, (\mathcal{P}_n, \mathcal{R}, \mathcal{G}_n)\}$. *Here* $\mathcal{P}_1, \ldots, \mathcal{P}_n$ *are the strongly connected components of* $\mathcal{G}$ *and* $\mathcal{G}_i = \mathcal{G} \cap (\mathcal{P}_i \times \mathcal{P}_i)$ *for every* $1 \leqslant i \leqslant n$.

The following result is well-known [1,11,14,25].

Theorem 4. *The dependency graph and SCC processors are sound and complete.* □

We continue the previous example.

Example 5. The SCC processor does not make progress on the DP problem $(\mathcal{P}, \mathcal{R}, \mathsf{DG}(\mathcal{P}, \mathcal{R}))$ since the three nodes form a single SCC in the graph.

3 Tree Automata Completion

We start by recalling some basic facts and notation. A *tree automaton* $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ consists of a signature $\mathcal{F}$, a finite set of states Q, a set of final states $Q_f \subseteq Q$, and a set of transitions Δ of the form $f(q_1, \ldots, q_n) \to q$ with f an n-ary function symbol in $\mathcal{F}$ and $q, q_1, \ldots, q_n \in Q$. The language $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of ground terms $t \in \mathcal{T}(\mathcal{F})$ such that $t \to_\Delta^* q$ for some $q \in Q_f$. Let $\mathcal{R}$ be a TRS over $\mathcal{F}$. The set $\{t \in \mathcal{T}(\mathcal{F}) \mid s \to_{\mathcal{R}}^* t \text{ for some } s \in L\}$ of descendants of a set $L \subseteq \mathcal{T}(\mathcal{F})$ of ground terms is denoted by $\to_{\mathcal{R}}^*(L)$. We say that $\mathcal{A}$ is *compatible* with $\mathcal{R}$ and L if $L \subseteq \mathcal{L}(\mathcal{A})$ and for each rewrite rule $l \to r \in \mathcal{R}$ and state substitution $\sigma\colon \mathcal{V}\mathsf{ar}(l) \to Q$ such that $l\sigma \to_\Delta^* q$ it holds that $r\sigma \to_\Delta^* q$. For left-linear $\mathcal{R}$ it is known that $\to_{\mathcal{R}}^*(L) \subseteq \mathcal{L}(\mathcal{A})$ whenever $\mathcal{A}$ is compatible with $\mathcal{R}$ and L [6]. To obtain a similar result for non-left-linear TRSs, in [16] quasi-deterministic automata were introduced. Let $\mathcal{A} = (\mathcal{F}, Q, Q_f, \Delta)$ be a tree automaton. We say

that a state p *subsumes* a state q if p is final when q is final and for all transitions $f(u_1, \ldots, q, \ldots, u_n) \to u \in \Delta$, the transition $f(u_1, \ldots, p, \ldots, u_n) \to u$ belongs to Δ. For a left-hand side $l \in \mathsf{lhs}(\Delta)$ of a transition, the set $\{q \mid l \to q \in \Delta\}$ of possible right-hand sides is denoted by $Q(l)$. The automaton $\mathcal{A}$ is said to be *quasi-deterministic* if for every $l \in \mathsf{lhs}(\Delta)$ there exists a state $p \in Q(l)$ which subsumes every other state in $Q(l)$. In general, $Q(l)$ may contain more than one state that satisfies the above property. In the following we assume that there is a unique designated state in $Q(l)$, which we denote by p_l. The set of all designated states is denoted by Q_d and the restriction of Δ to transition rules $l \to q$ that satisfy $q = p_l$ is denoted by Δ_d. In [16] we showed that the tree automaton induced by Δ_d is deterministic and accepts the same language as $\mathcal{A}$. For non-left-linear TRSs $\mathcal{R}$ we modify the above definition of compatibility by demanding that the tree automaton $\mathcal{A}$ is quasi-deterministic and for each rewrite rule $l \to r \in \mathcal{R}$ and state substitution $\sigma\colon \mathcal{V}\mathsf{ar}(l) \to Q_d$ with $l\sigma \to_{\Delta_d}^* q$ it holds that $r\sigma \to_{\Delta}^* q$.

Theorem 6 ([6,16]). *Let $\mathcal{R}$ be a TRS, $\mathcal{A}$ a tree automaton, and L a set of ground terms. If $\mathcal{A}$ is compatible with $\mathcal{R}$ and L then $\to_{\mathcal{R}}^*(L) \subseteq \mathcal{L}(\mathcal{A})$.* □

For two TRSs $\mathcal{P}$ and $\mathcal{R}$ the dependency graph $\mathsf{DG}(\mathcal{P}, \mathcal{R})$ contains an arc from dependency pair α to dependency pair β if and only if there exist substitutions σ and τ such that $\mathsf{rhs}(\alpha)\sigma \to_{\mathcal{R}}^* \mathsf{lhs}(\beta)\tau$. Without loss of generality we may assume that $\mathsf{rhs}(\alpha)\sigma$ and $\mathsf{lhs}(\beta)\tau$ are ground terms. Hence there is no arc from α to β if and only if $\Sigma(\mathsf{lhs}(\beta)) \cap \to_{\mathcal{R}}^*(\Sigma(\mathsf{rhs}(\alpha))) = \varnothing$. Here $\Sigma(t)$ denotes the set of ground instances of t with respect to the signature consisting of a fresh constant $\#$ together with all function symbols that appear in $\mathcal{P} \cup \mathcal{R}$ minus the root symbols of the left- and right-hand sides of $\mathcal{P}$ that do neither occur on positions below the root in $\mathcal{P}$ nor in $\mathcal{R}$.[1] For an arbitrary term t and regular language L it is decidable whether $\Sigma(t) \cap L = \varnothing$—a result of Tison (see [22])—and hence we can check the above condition by constructing a tree automaton that accepts $\to_{\mathcal{R}}^*(\Sigma(\mathsf{rhs}(\alpha)))$. Since this set is in general not regular, we compute an over-approximation with the help of tree automata completion starting from an automaton that accepts $\Sigma(\mathsf{ren}(\mathsf{rhs}(\alpha)))$. Here ren is the function that linearizes its argument by replacing all occurrences of variables with fresh variables, which is needed to ensure the regularity of $\Sigma(\mathsf{ren}(\mathsf{rhs}(\alpha)))$.

Definition 7. *Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs, L a language, and $\alpha, \beta \in \mathcal{P}$. We say that β is* unreachable *from α with respect to L if there is a tree automaton $\mathcal{A}$ compatible with $\mathcal{R}$ and $L \cap \Sigma(\mathsf{ren}(\mathsf{rhs}(\alpha)))$ such that $\Sigma(\mathsf{lhs}(\beta)) \cap \mathcal{L}(\mathcal{A}) = \varnothing$.*

The language L in the above definition allows us to refine the set of starting terms $\Sigma(\mathsf{ren}(\mathsf{rhs}(\alpha)))$ which are considered in the computation of an arc from α to β. In Section 4 we make use of L to remove arcs from the (real) dependency graph. In the remainder of this section we always have $L = \Sigma(\mathsf{ren}(\mathsf{rhs}(\alpha)))$.

[1] The fresh constant $\#$ is added to the signature to ensure that $\Sigma(t)$ cannot be empty.

Definition 8. *The nodes of the* c-dependency graph $\mathsf{DG_c}(\mathcal{P},\mathcal{R})$ *are the rewrite rules of* $\mathcal{P}$ *and there is no arc from* α *to* β *if and only if* β *is unreachable from* α *with respect to* $\Sigma(\mathsf{ren}(\mathsf{rhs}(\alpha)))$.

The c in the above definition refers to the fact that a compatible tree automaton is constructed by tree automata completion.

Lemma 9. *Let* $\mathcal{P}$ *and* $\mathcal{R}$ *be two TRSs. Then* $\mathsf{DG_c}(\mathcal{P},\mathcal{R}) \supseteq \mathsf{DG}(\mathcal{P},\mathcal{R})$.

Proof. Easy consequence of Theorem 6. □

The general idea of tree automata completion [6,7,16] is to look for violations of the compatibility requirement: $l\sigma \to_\Delta^* q$ ($l\sigma \to_{\Delta_d}^* q$) but not $r\sigma \to_\Delta^* q$ for some rewrite rule $l \to r \in \mathcal{R}$, state substitution $\sigma\colon \mathcal{V}\mathsf{ar}(l) \to Q$ ($\sigma\colon \mathcal{V}\mathsf{ar}(l) \to Q_d$), and state q. This triggers the addition of new states and transitions to the current automaton to ensure $r\sigma \to_\Delta^* q$. There are several ways to do this, ranging from establishing a completely new path $r\sigma \to_\Delta^* q$ to adding as few as possible new transitions by reusing transitions from the current automaton. After $r\sigma \to_\Delta^* q$ has been established, we look for further compatibility violations. This process is repeated until a compatible tree automaton is obtained, which may never happen if new states are kept being added.

Example 10. We continue with our example. A tree automaton $\mathcal{A}$, with final state 2, that accepts $\Sigma(\mathsf{H}(x,y))$ is easily constructed:

$$\mathsf{a} \to 1 \qquad \mathsf{f}(1,1) \to 1 \qquad \mathsf{g}(1) \to 1 \qquad \mathsf{h}(1,1) \to 1 \qquad \mathsf{H}(1,1) \to 2$$

Because $\to_{\mathcal{R}}^*(\Sigma(\mathsf{H}(x,y))) = \Sigma(\mathsf{H}(x,y))$, the automaton is already compatible with $\mathcal{R}$ and $\Sigma(\mathsf{H}(x,y))$, so completion is trivial here. As $\mathsf{H}(\mathsf{g}(\mathsf{a}),\mathsf{a})$ is accepted by $\mathcal{A}$, $\mathsf{DG_c}(\mathcal{P},\mathcal{R})$ contains arcs from 1 and 3 to 2 and 3. Similarly, we can construct a tree automaton $\mathcal{B}$ with final state 2 that is compatible with $\mathcal{R}$ and $\Sigma(\mathsf{F}(\mathsf{g}(\mathsf{a}),\mathsf{h}(x,y)))$:

$$\begin{array}{lllll} \mathsf{a} \to 1 & \mathsf{f}(1,1) \to 1 & \mathsf{F}(3,4) \to 2 & \mathsf{g}(1) \to 4 & \mathsf{h}(1,1) \to 1 \\ \mathsf{a} \to 2 & \mathsf{f}(1,1) \to 4 & \mathsf{g}(1) \to 1 & \mathsf{g}(2) \to 3 & \mathsf{h}(1,1) \to 4 \end{array}$$

Because $\mathsf{F}(\mathsf{g}(\mathsf{a}),\mathsf{h}(\mathsf{a},\mathsf{a}))$ is accepted by $\mathcal{B}$, we obtain an arc from 2 to 1. Further arcs do not exist.

It can be argued that the use of tree automata techniques for the DP problem of Example 2 is a waste of resources because the dependency graph can also be computed by just taking the root symbols of the dependency pairs into consideration. However, in the next section we show that this radically changes when taking right-hand sides of forward closures into account.

4 Incorporating Forward Closures

Given a DP problem $(\mathcal{P},\mathcal{R},\mathcal{G})$ and $\alpha,\beta \in \mathcal{P}$, an arc from α to β in the dependency graph $\mathsf{DG}(\mathcal{P},\mathcal{R})$ is an indication that β may follow α in an *infinite*

sequence in $\mathcal{P} \cup \mathcal{R}$ but not a sufficient condition because if the problem is finite there are no infinite sequences whatsoever. What we would like to determine is whether β can follow α infinitely many times in a minimal sequence. With the existing approximations of the dependency graph, only local connections can be tested. In this section we show that by using right-hand sides of forward closures, we can sometimes delete arcs from the dependency graph which cannot occur infinitely many times in minimal sequences.

When proving the termination of a TRS $\mathcal{R}$ that is non-overlapping or right-linear it is sufficient to restrict attention to the set of right-hand sides of forward closures [4,9]. This set is defined as the closure of the right-hand sides of the rules in $\mathcal{R}$ under narrowing. Formally, given a set L of terms, $\mathsf{RFC}(L, \mathcal{R})$ is the least extension of L such that $t[r]_p\sigma \in \mathsf{RFC}(L, \mathcal{R})$ whenever $t \in \mathsf{RFC}(L, \mathcal{R})$ and there exist a non-variable position p and a fresh variant $l \to r$ of a rewrite rule in $\mathcal{R}$ with σ a most general unifier of $t|_p$ and l. In the sequel we write $\mathsf{RFC}(t, \mathcal{R})$ for $\mathsf{RFC}(\{t\}, \mathcal{R})$. Furthermore, we restrict our attention to right-linear TRSs because we cannot cope with non-right-linear TRSs during the construction of the set of right-hand sides of forward closures [8].

Dershowitz [4] showed that a right-linear TRS $\mathcal{R}$ is terminating if and only if $\mathcal{R}$ is terminating on $\mathsf{RFC}(\mathsf{rhs}(\mathcal{R}), \mathcal{R})$. In [17,18] we showed how to extend this result to the dependency pairs setting.

Lemma 11. *Let $\mathcal{P}$ and $\mathcal{R}$ be two right-linear TRSs and let $\alpha \in \mathcal{P}$. Then $\mathcal{P} \cup \mathcal{R}$ admits a minimal rewrite sequence with infinitely many α steps if and only if it admits such a sequence starting from a term in $\mathsf{RFC}(\mathsf{rhs}(\alpha), \mathcal{P} \cup \mathcal{R})$.* □

A careful inspection of the complicated proof of Lemma 11 given in [18] reveals that the statement can be adapted to our needs. We say that dependency pair β *directly follows* dependency pair α in a minimal sequence $s_1 \xrightarrow{\epsilon}_{\alpha_1} t_1 \to^*_{\mathcal{R}} s_2 \xrightarrow{\epsilon}_{\alpha_2} t_2 \to^*_{\mathcal{R}} \cdots$ if $\alpha_i = \alpha$ and $\alpha_{i+1} = \beta$ for some $i \geqslant 1$.

Lemma 12. *Let $\mathcal{P}$ and $\mathcal{R}$ be two right-linear TRSs and let $\alpha, \beta \in \mathcal{P}$. The TRS $\mathcal{P} \cup \mathcal{R}$ admits a minimal rewrite sequence in which infinitely many β steps directly follow α steps if and only if it admits such a sequence starting from a term in $\mathsf{RFC}(\mathsf{rhs}(\alpha), \mathcal{P} \cup \mathcal{R})$.* □

Definition 13. *Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs. The* improved dependency graph *of $\mathcal{P}$ and $\mathcal{R}$, denoted by $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$, has the rules in $\mathcal{P}$ as nodes and there is an arc from $s \to t$ to $u \to v$ if and only if there exist substitutions σ and τ such that $t\sigma \to^*_{\mathcal{R}} u\tau$ and $t\sigma \in \Sigma_{\#}(\mathsf{RFC}(t, \mathcal{P} \cup \mathcal{R}))$. Here $\Sigma_{\#}$ is the operation that replaces all variables by the fresh constant $\#$.*

Note that the use of $\Sigma_{\#}$ in the above definition is essential. If we would replace $\Sigma_{\#}$ by Σ then $\Sigma(\mathsf{RFC}(t, \mathcal{P} \cup \mathcal{R})) \supseteq \Sigma(t)$ because $t \in \mathsf{RFC}(t, \mathcal{P} \cup \mathcal{R})$ and hence $\mathsf{IDG}(\mathcal{P}, \mathcal{R}) = \mathsf{DG}(\mathcal{P}, \mathcal{R})$. According to the following lemma the improved dependency graph can be used whenever the participating TRSs are right-linear.

Lemma 14. *Let $\mathcal{P}$ and $\mathcal{R}$ be right-linear TRSs and $\alpha, \beta \in \mathcal{P}$. If there is a minimal sequence in $\mathcal{P} \cup \mathcal{R}$ in which infinitely many β steps directly follow α steps, then $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$ admits an arc from α to β.*

Proof. Assume that there is a minimal rewrite sequence

$$s_1 \xrightarrow{\epsilon}_{\mathcal{P}} t_1 \to^*_{\mathcal{R}} s_2 \xrightarrow{\epsilon}_{\mathcal{P}} t_2 \to^*_{\mathcal{R}} s_3 \xrightarrow{\epsilon}_{\mathcal{P}} \cdots$$

in which infinitely many β steps directly follow α steps. According to Lemma 12 we may assume without loss of generality that $s_1 \in \mathsf{RFC}(\mathsf{rhs}(\alpha), \mathcal{P} \cup \mathcal{R})$. Let $i \geqslant 1$ such that $s_i \xrightarrow{\epsilon}_{\alpha} t_i \to^*_{\mathcal{R}} s_{i+1} \xrightarrow{\epsilon}_{\beta} t_{i+1}$. Because $\mathsf{RFC}(\mathsf{rhs}(\alpha), \mathcal{P} \cup \mathcal{R})$ is closed under rewriting with respect to $\mathcal{P}$ and $\mathcal{R}$ we know that $t_i \in \mathsf{RFC}(\mathsf{rhs}(\alpha), \mathcal{P} \cup \mathcal{R})$. We have $t_i = \mathsf{rhs}(\alpha)\sigma$ and $s_{i+1} = \mathsf{lhs}(\beta)\tau$ for some substitutions σ and τ. From $t_i \in \mathsf{RFC}(\mathsf{rhs}(\alpha), \mathcal{P} \cup \mathcal{R})$ we infer that $t_i\theta \in \Sigma_{\#}(\mathsf{RFC}(\mathsf{rhs}(\alpha), \mathcal{P} \cup \mathcal{R}))$ for the substitution θ that replaces every variable by $\#$. Due to the fact that rewriting is closed under substitutions we have $t_i\theta \to^*_{\mathcal{R}} s_{i+1}\theta$. Hence $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$ contains an arc from α to β. □

The following example shows that it is essential to include $\mathcal{P}$ in the construction of the set $\Sigma_{\#}(\mathsf{RFC}(t, \mathcal{P} \cup \mathcal{R}))$ in the definition of $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$.

Example 15. Consider the TRS $\mathcal{R}$ consisting of the rewrite rules $\mathsf{f}(x) \to \mathsf{g}(x)$, $\mathsf{g}(\mathsf{a}) \to \mathsf{h}(\mathsf{b})$, and $\mathsf{h}(x) \to \mathsf{f}(\mathsf{a})$ and the TRS $\mathcal{P} = \mathsf{DP}(\mathcal{R})$ consisting of

$$\mathsf{F}(x) \to \mathsf{G}(x) \qquad \mathsf{G}(\mathsf{a}) \to \mathsf{H}(\mathsf{b}) \qquad \mathsf{H}(x) \to \mathsf{F}(\mathsf{a})$$

The DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{P} \times \mathcal{P})$ is not finite because it admits the loop

$$\mathsf{F}(\mathsf{a}) \xrightarrow{\epsilon}_{\mathcal{P}} \mathsf{G}(\mathsf{a}) \xrightarrow{\epsilon}_{\mathcal{P}} \mathsf{H}(\mathsf{b}) \xrightarrow{\epsilon}_{\mathcal{P}} \mathsf{F}(\mathsf{a})$$

Let $t = \mathsf{G}(x)$. We have $\mathsf{RFC}(t, \mathcal{R}) = \{t\}$ and hence $\Sigma_{\#}(\mathsf{RFC}(t, \mathcal{R})) = \{\mathsf{G}(\#)\}$. If we now replace $\Sigma_{\#}(\mathsf{RFC}(t, \mathcal{P} \cup \mathcal{R}))$ by $\Sigma_{\#}(\mathsf{RFC}(t, \mathcal{R}))$ in Definition 13, we would conclude that $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$ does not contain an arc from $\mathsf{F}(x) \to \mathsf{G}(x)$ to $\mathsf{G}(\mathsf{a}) \to \mathsf{H}(\mathsf{b})$ because $\mathsf{G}(\#)$ is a normal form which is different from $\mathsf{G}(\mathsf{a})$. But this makes the resulting DP problem $(\mathcal{P}, \mathcal{R}, \mathsf{IDG}(\mathcal{P}, \mathcal{R}))$ finite.

Theorem 16. *The* improved dependency graph processor

$$(\mathcal{P}, \mathcal{R}, \mathcal{G}) \mapsto \begin{cases} \{(\mathcal{P}, \mathcal{R}, \mathcal{G} \cap \mathsf{IDG}(\mathcal{P}, \mathcal{R}))\} & \text{if } \mathcal{P} \cup \mathcal{R} \text{ is right-linear} \\ \{(\mathcal{P}, \mathcal{R}, \mathcal{G} \cap \mathsf{DG}(\mathcal{P}, \mathcal{R}))\} & \text{otherwise} \end{cases}$$

is sound and complete.

Proof. Soundness is an easy consequence of Lemma 14 and Theorem 4. Completeness follows from the inclusions $\mathcal{G} \cap \mathsf{DG}(\mathcal{P}, \mathcal{R}) \subseteq \mathcal{G} \supseteq \mathcal{G} \cap \mathsf{IDG}(\mathcal{P}, \mathcal{R})$. □

Example 17. We consider again the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ of Example 2. Let $s = \mathsf{H}(x, y)$ and $t = \mathsf{F}(\mathsf{g}(\mathsf{a}), \mathsf{h}(x, y))$. We first compute $\mathsf{RFC}(s, \mathcal{P} \cup \mathcal{R})$ and $\mathsf{RFC}(t, \mathcal{P} \cup \mathcal{R})$. The former set consists of $\mathsf{H}(x, y)$ together will all terms in $\mathsf{RFC}(t, \mathcal{P} \cup \mathcal{R})$.

Each term contained in the latter set is an instance of $\mathsf{F}(\mathsf{g}(\mathsf{a}), x)$ or $\mathsf{H}(\mathsf{a}, x)$. It follows that each term in $\Sigma_{\#}(\mathsf{RFC}(s, \mathcal{P} \cup \mathcal{R}))$ is a ground instance of $\mathsf{F}(\mathsf{g}(\mathsf{a}), x)$ or $\mathsf{H}(\mathsf{a}, x)$ or equal to $\mathsf{H}(\#, \#)$. Similarly, each term in $\Sigma_{\#}(\mathsf{RFC}(t, \mathcal{P} \cup \mathcal{R}))$ is a ground instance of $\mathsf{F}(\mathsf{g}(\mathsf{a}), x)$ or $\mathsf{H}(\mathsf{a}, x)$. Hence $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$ contains an arc from 2 to 1. Further arcs do not exist because there are no substitution τ and term $u \in \Sigma_{\#}(\mathsf{RFC}(s, \mathcal{P} \cup \mathcal{R}))$ such that $u \to_{\mathcal{R}}^* \mathsf{H}(\mathsf{g}(x), y)\tau$. So $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$ looks as follows:

$$1 \longleftarrow 2 \qquad 3$$

Therefore the improved dependency graph processor produces the new DP problem $(\mathcal{P}, \mathcal{R}, \mathsf{IDG}(\mathcal{P}, \mathcal{R}))$. Since the above graph does not admit any SCCs, the SCC processor yields the finite DP problem $(\mathcal{P}, \mathcal{R}, \varnothing)$. Consequently, $\mathcal{R}$ is terminating.

Similar to $\mathsf{DG}(\mathcal{P}, \mathcal{R})$, $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$ is not computable in general. We over-approximate $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$ by using tree automata completion as described in Section 3.

Definition 18. *Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs. The nodes of the* c-improved dependency graph $\mathsf{IDG}_{\mathsf{c}}(\mathcal{P}, \mathcal{R})$ *are the rewrite rules of $\mathcal{P}$ and there is no arc from α to β if and only if β is unreachable from α with respect to $\Sigma_{\#}(\mathsf{RFC}(\mathsf{rhs}(\alpha), \mathcal{P} \cup \mathcal{R}))$.*

Lemma 19. *Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs. Then $\mathsf{IDG}_{\mathsf{c}}(\mathcal{P}, \mathcal{R}) \supseteq \mathsf{IDG}(\mathcal{P}, \mathcal{R})$.*

Proof. Assume to the contrary that the claim does not hold. Then there are rules $s \to t$ and $u \to v$ in $\mathcal{P}$ such that there is an arc from $s \to t$ to $u \to v$ in $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$ but not in $\mathsf{IDG}_{\mathsf{c}}(\mathcal{P}, \mathcal{R})$. By Definition 13 there are substitutions σ and τ such that $t\sigma \in L$ with $L = \Sigma_{\#}(\mathsf{RFC}(t, \mathcal{P} \cup \mathcal{R}))$ and $t\sigma \to_{\mathcal{R}}^* u\tau$. Since $\mathsf{IDG}_{\mathsf{c}}(\mathcal{P}, \mathcal{R})$ does not admit an arc from $s \to t$ to $u \to v$, there is a tree automaton $\mathcal{A}$ compatible with $\mathcal{R}$ and $L \cap \Sigma(\mathsf{ren}(t))$ such that $\Sigma(u) \cap \mathcal{L}(\mathcal{A}) = \varnothing$. Theorem 6 yields $\to_{\mathcal{R}}^*(L \cap \Sigma(\mathsf{ren}(t))) \subseteq \mathcal{L}(\mathcal{A})$. From $t\sigma \in L \cap \Sigma(\mathsf{ren}(t))$ and $t\sigma \to_{\mathcal{R}}^* u\tau$ we infer that $u\tau \in \mathcal{L}(\mathcal{A})$, contradicting $\Sigma(u) \cap \mathcal{L}(\mathcal{A}) = \varnothing$. □

To compute $\mathsf{IDG}_{\mathsf{c}}(\mathcal{P}, \mathcal{R})$ we have to construct an intermediate tree automaton that accepts $\mathsf{RFC}(\mathsf{rhs}(\alpha), \mathcal{P} \cup \mathcal{R})$. This can be done by using tree automata completion as described in in [8,16]. We continue our leading example.

Example 20. We construct $\mathsf{IDG}_{\mathsf{c}}(\mathcal{P}, \mathcal{R})$ for the DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ of Example 17. Let $s = \mathsf{H}(x, y)$. A tree automaton $\mathcal{A}$ that is compatible with $\mathcal{R}$ and $\Sigma_{\#}(\mathsf{RFC}(s, \mathcal{P} \cup \mathcal{R})) \cap \Sigma(s)$ consists of the transitions

$$\begin{array}{lllll} \# \to 1 & \mathsf{g}(3) \to 4 & \mathsf{f}(4,5) \to 5 & \mathsf{h}(1,1) \to 5 & \mathsf{H}(1,1) \to 2 \\ \mathsf{a} \to 3 & \mathsf{g}(6) \to 5 & & \mathsf{h}(3,5) \to 6 & \mathsf{H}(3,5) \to 2 \end{array}$$

with final state 2. Since $\mathcal{A}$ does not accept any ground instance of the term $\mathsf{H}(\mathsf{g}(x), y)$ we conclude that the rules 2 and 3 are unreachable from 1 and 3. It remains to check whether there is any outgoing arc from rule 2. Let $t =$

$\mathsf{F}(\mathsf{g}(\mathsf{a}), \mathsf{h}(x, y))$ be the right-hand side of 2. Similar as before we can construct a tree automaton $\mathcal{B}$ with final state 5 and consisting of the transitions

$$\begin{array}{llll} \# \to 1 & \mathsf{g}(2) \to 3 & \mathsf{f}(3,4) \to 4 & \mathsf{h}(1,1) \to 4 \\ \mathsf{a} \to 2 & \mathsf{g}(6) \to 4 & \mathsf{F}(3,4) \to 5 & \mathsf{h}(2,4) \to 6 \end{array}$$

which is compatible with $\mathcal{R}$ and $\Sigma_{\#}(\mathsf{RFC}(t, \mathcal{P} \cup \mathcal{R})) \cap \Sigma(t)$. Since the instance $\mathsf{F}(\mathsf{g}(\mathsf{a}), \mathsf{h}(\#, \#))$ of $\mathsf{F}(\mathsf{g}(x), y)$ is accepted by $\mathcal{B}$, $\mathsf{IDG}_{\mathsf{c}}(\mathcal{P}, \mathcal{R})$ contains an arc from 2 to 1. Further arcs do not exist. Hence $\mathsf{IDG}_{\mathsf{c}}(\mathcal{P}, \mathcal{R})$ coincides with $\mathsf{IDG}(\mathcal{P}, \mathcal{R})$.

5 Comparison

In the literature several over-approximations of the dependency graph are described [1,12,21,22,23]. In this section we compare the tree automata approach to approximate the processors of Definition 1 and Theorem 16 developed in the preceding sections with the earlier tree automata approach of [22] as well as the approximation used in tools like AProVE [10] and TTT$_2$ [19], which is a combination of ideas of [12] and [23]. We start by formally defining the latter.

Definition 21. *Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs. The nodes of the* estimated dependency graph $\mathsf{DG}_{\mathsf{e}}(\mathcal{P}, \mathcal{R})$ *are the rewrite rules of $\mathcal{P}$ and there is an arc from $s \to t$ to $u \to v$ if and only if $\mathsf{tcap}(\mathcal{R}, t)$ and u as well as t and $\mathsf{tcap}(\mathcal{R}^{-1}, u)$ are unifiable. Here $\mathcal{R}^{-1} = \{r \to l \mid l \to r \in \mathcal{R}\}$ and the function $\mathsf{tcap}(\mathcal{R}, t)$ is defined as $f(\mathsf{tcap}(\mathcal{R}, t_1), \ldots, \mathsf{tcap}(\mathcal{R}, t_n))$ if $t = f(t_1, \ldots, t_n)$ and the term $f(\mathsf{tcap}(\mathcal{R}, t_1), \ldots, \mathsf{tcap}(\mathcal{R}, t_n))$ does not unify with any left-hand side of $\mathcal{R}$. Otherwise $\mathsf{tcap}(\mathcal{R}, t)$ is a fresh variable.*

The approach described in [22] to approximate dependency graphs based on tree automata techniques relies on regularity preservation rather than completion. Below we recall the relevant definitions. An *approximation mapping* is a mapping ϕ from TRSs to TRSs such that $\to_{\mathcal{R}} \subseteq \to^*_{\phi(\mathcal{R})}$. We say that ϕ is *regularity preserving* if $\leftarrow^*_{\phi(\mathcal{R})}(L) = \{s \in \mathcal{T}(\mathcal{F}) \mid s \to^*_{\phi(\mathcal{R})} t \text{ for some } t \in L\}$ is regular for all $\mathcal{R}$ and regular L. Here $\mathcal{F}$ is the signature of $\mathcal{R}$.

The three approximation mappings s, nv, g are defined as follows: $\mathsf{s}(\mathcal{R}) = \{\mathsf{ren}(l) \to x \mid l \to r \in \mathcal{R} \text{ and } x \text{ is a fresh variable}\}$, $\mathsf{nv}(\mathcal{R}) = \{\mathsf{ren}(l) \to \mathsf{ren}(r) \mid l \to r \in \mathcal{R}\}$, and $\mathsf{g}(\mathcal{R})$ is defined as any left-linear TRS that is obtained from $\mathcal{R}$ by linearizing the left-hand sides and renaming the variables in the right-hand sides that occur at a depth greater than 1 in the corresponding left-hand sides. These mappings are known to be regularity preserving [5,24].

Definition 22. *Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs and let ϕ be an approximation mapping. The nodes of the* ϕ-approximated dependency graph $\mathsf{DG}_{\phi}(\mathcal{P}, \mathcal{R})$ *are the rewrite rules of $\mathcal{P}$ and there is an arc from $s \to t$ to $u \to v$ if and only if both $\Sigma(t) \cap \leftarrow^*_{\phi(\mathcal{R})}(\Sigma(\mathsf{ren}(u))) \neq \varnothing$ and $\Sigma(u) \cap \leftarrow^*_{\phi(\mathcal{R}^{-1})}(\Sigma(\mathsf{ren}(t))) \neq \varnothing$.*

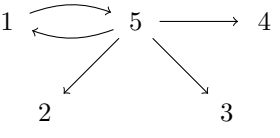
Lemma 23 ([12,22,23]). *For TRSs $\mathcal{P}$ and $\mathcal{R}$, $\mathsf{DG}_{\mathsf{e}}(\mathcal{P}, \mathcal{R}) \supseteq \mathsf{DG}(\mathcal{P}, \mathcal{R})$ and $\mathsf{DG}_{\mathsf{s}}(\mathcal{P}, \mathcal{R}) \supseteq \mathsf{DG}_{\mathsf{nv}}(\mathcal{P}, \mathcal{R}) \supseteq \mathsf{DG}_{\mathsf{g}}(\mathcal{P}, \mathcal{R}) \supseteq \mathsf{DG}(\mathcal{P}, \mathcal{R})$.* □

From Examples 2 and 20 it is obvious that neither $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$ nor $\mathsf{DG_g}(\mathcal{P},\mathcal{R})$ subsumes $\mathsf{IDG_c}(\mathcal{P},\mathcal{R})$. The converse depends very much on the approximation strategy that is used; it can always happen that the completion procedure does not terminate or that the over-approximation is too inexact. Nevertheless, there are problems where $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$ and $\mathsf{DG_s}(\mathcal{P},\mathcal{R})$ are properly contained in $\mathsf{IDG_c}(\mathcal{P},\mathcal{R})$. An example is provided by the TRS consisting of the rules $\mathsf{f}(x,x) \to \mathsf{f}(\mathsf{a},\mathsf{g}(x,\mathsf{b}))$, $\mathsf{f}(\mathsf{a},\mathsf{g}(x,x)) \to \mathsf{f}(\mathsf{a},\mathsf{a})$, and $\mathsf{g}(\mathsf{a},\mathsf{b}) \to \mathsf{b}$. The following example shows that neither $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$ nor $\mathsf{DG_g}(\mathcal{P},\mathcal{R})$ subsumes $\mathsf{DG_c}(\mathcal{P},\mathcal{R})$.

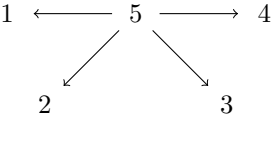
Example 24. Consider the TRSs $\mathcal{R}$ and $\mathcal{P}$ with $\mathcal{R}$ consisting of the rewrite rules $\mathsf{p}(\mathsf{p}(\mathsf{p}(x))) \to \mathsf{p}(\mathsf{p}(x))$, $\mathsf{f}(x) \to \mathsf{g}(\mathsf{p}(\mathsf{p}(\mathsf{p}(\mathsf{a}))))$, and $\mathsf{g}(\mathsf{p}(\mathsf{p}(\mathsf{s}(x)))) \to \mathsf{f}(x)$ and $\mathcal{P} = \mathsf{DP}(\mathcal{R})$ consisting of

$$\begin{array}{lll} 1\colon \mathsf{F}(x) \to \mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{p}(\mathsf{a})))) & 3\colon \mathsf{F}(x) \to \mathsf{P}(\mathsf{p}(\mathsf{a})) & 5\colon \mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{s}(x)))) \to \mathsf{F}(x) \\ 2\colon \mathsf{F}(x) \to \mathsf{P}(\mathsf{p}(\mathsf{p}(\mathsf{a}))) & 4\colon \mathsf{F}(x) \to \mathsf{P}(\mathsf{a}) & \end{array}$$

First we compute $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$. It is clear that $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$ contains arcs from 5 to 1, 2, 3, and 4. Furthermore, it contains an arc from 1 to 5 because the term $\mathsf{tcap}(\mathcal{R},\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{p}(\mathsf{a})))))) = \mathsf{G}(y)$ unifies with $\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{s}(x))))$ and $\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{p}(\mathsf{a}))))$ unifies with $\mathsf{tcap}(\mathcal{R}^{-1},\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{s}(x))))) = \mathsf{G}(y)$. Further arcs do not exist and hence $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$ looks as follows:



Next we compute $\mathsf{DG_g}(\mathcal{P},\mathcal{R})$. Similarly as $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$, $\mathsf{DG_g}(\mathcal{P},\mathcal{R})$ has arcs from 5 to 1, 2, 3, and 4. Furthermore $\mathsf{DG_g}(\mathcal{P},\mathcal{R})$ contains an arc from 1 to 5 because $\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{p}(\mathsf{a})))) \to_{\mathsf{g}(\mathcal{R})} \mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{s}(\mathsf{a})))) \in \Sigma(\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{s}(x))))))$ by applying the rewrite rule $\mathsf{p}(\mathsf{p}(\mathsf{p}(x))) \to \mathsf{p}(\mathsf{p}(y))$ and $\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{s}(x)))) \to_{\mathsf{g}(\mathcal{R}^{-1})} \mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{p}(\mathsf{a})))) \in \{\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{p}(\mathsf{a}))))\}$ using the rule $\mathsf{p}(\mathsf{p}(x)) \to \mathsf{p}(\mathsf{p}(\mathsf{p}(y)))$. Hence $\mathsf{DG_g}(\mathcal{P},\mathcal{R})$ coincides with $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$. The graph $\mathsf{DG_c}(\mathcal{P},\mathcal{R})$



does not contain an arc from 1 to 5 because 5 is unreachable from 1. This is certified by the following tree automaton $\mathcal{A}$:

$$\mathsf{a} \to 1 \qquad \mathsf{p}(1) \to 2 \qquad \mathsf{p}(2) \to 3 \qquad \mathsf{p}(2) \to 4 \qquad \mathsf{p}(3) \to 4 \qquad \mathsf{G}(4) \to 5$$

with 5 as the only final state. Note that $\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{p}(\mathsf{a})))) \in \mathcal{L}(\mathcal{A})$, $\to_{\mathcal{R}}^*(\mathcal{L}(\mathcal{A})) = \mathcal{L}(\mathcal{A})$, and $\Sigma(\mathsf{G}(\mathsf{p}(\mathsf{p}(\mathsf{s}(x))))))) \cap \mathcal{L}(\mathcal{A}) = \varnothing$.

Concerning the converse direction, there are TRSs like $\mathsf{f}(\mathsf{a},\mathsf{b},x) \to \mathsf{f}(x,x,x)$ such that $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$ and $\mathsf{DG_{nv}}(\mathcal{P},\mathcal{R})$ are properly contained in $\mathsf{DG_c}(\mathcal{P},\mathcal{R})$. We assume that this also holds for $\mathsf{DG_s}(\mathcal{P},\mathcal{R})$ although we did not succeed in finding an example.

6 Innermost Termination

In this section we sketch how the ideas presented in Sections 3 and 4 can be extended to innermost termination. Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs and $\mathcal{G} \subseteq \mathcal{P} \times \mathcal{P}$ a directed graph. A *minimal innermost* rewrite sequence is an infinite rewrite sequence of the form $s_1 \xrightarrow{\mathsf{i}}_{\mathcal{P}} t_1 \xrightarrow{\mathsf{i}}_{\mathcal{R}}^{*} s_2 \xrightarrow{\mathsf{i}}_{\mathcal{P}} t_2 \xrightarrow{\mathsf{i}}_{\mathcal{R}}^{*} \cdots$ such that $s_i \xrightarrow{\epsilon} t_i$ and $(\alpha_i, \alpha_{i+1}) \in \mathcal{G}$ for all $i \geqslant 1$. Here $\xrightarrow{\mathsf{i}}$ denotes the innermost rewrite relation of $\mathcal{P} \cup \mathcal{R}$. A DP problem $(\mathcal{P}, \mathcal{R}, \mathcal{G})$ is called *innermost finite* if there are no minimal innermost rewrite sequences.

Definition 25. *Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs. The* innermost dependency graph *of $\mathcal{P}$ and $\mathcal{R}$, denoted by $\mathsf{DG}^{\mathsf{i}}(\mathcal{P}, \mathcal{R})$, has the rules in $\mathcal{P}$ as nodes and there is an arc from $s \to t$ to $u \to v$ if and only if there exist substitutions σ and τ such that $t\sigma \xrightarrow{\mathsf{i}}_{\mathcal{R}}^{*} u\tau$ and $s\sigma$ and $u\tau$ are normal forms with respect to $\mathcal{R}$.*

By incorporating right-hand sides of forward closures, arcs of the innermost dependency graph can sometimes be eliminated. The only complication is that innermost rewriting is not closed under substitutions. To overcome this problem, we add a fresh unary function symbol besides the constant # to the signature and assume that $\Sigma_{\#}(\mathsf{RFC}(t, \mathcal{P} \cup \mathcal{R}))$ denotes the set of ground terms that are obtained from terms in $\mathsf{RFC}(t, \mathcal{P} \cup \mathcal{R})$ by instantiating the variables by terms built from # and this unary function symbol.

Definition 26. *Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs. The* improved innermost dependency graph *of $\mathcal{P}$ and $\mathcal{R}$, denoted by $\mathsf{IDG}^{\mathsf{i}}(\mathcal{P}, \mathcal{R})$, has the rules in $\mathcal{P}$ as nodes and there is an arc from $s \to t$ to $u \to v$ if and only if there exist substitutions σ and τ such that $t\sigma \xrightarrow{\mathsf{i}}_{\mathcal{R}}^{*} u\tau$, $t\sigma \in \Sigma_{\#}(\mathsf{RFC}(t, \mathcal{P} \cup \mathcal{R}))$, and $s\sigma$ and $u\tau$ are normal forms with respect to $\mathcal{R}$.*

The following result corresponds to Lemma 14.

Lemma 27. *Let $\mathcal{P}$ and $\mathcal{R}$ be right-linear TRSs and $\alpha, \beta \in \mathcal{P}$. If there is a minimal innermost sequence in $\mathcal{P} \cup \mathcal{R}$ in which infinitely many β steps directly follow α steps then $\mathsf{IDG}^{\mathsf{i}}(\mathcal{P}, \mathcal{R})$ admits an arc from α to β.* □

We approximate (improved) innermost dependency graphs as discussed in Section 3. In order to make use of the fact that $s\sigma$ and $u\tau$ are normal forms with respect to $\mathcal{R}$, we restrict $\Sigma(\mathsf{ren}(t))$ and $\Sigma_{\#}(\mathsf{RFC}(t, \mathcal{P} \cup \mathcal{R}))$ to the normalized instances (i.e., ground instances that are obtained by substituting normal forms for the variables) of $\mathsf{ren}(t)$, denoted by $\mathsf{NF}(\mathsf{ren}(t), \mathcal{R})$. This is possible because $s\sigma \to_{\mathcal{P}} t\sigma$ and σ is normalized as $s\sigma$ is a normal form.

Definition 28. *Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs. The nodes of the* c-innermost dependency graph $\mathsf{DG}_{\mathsf{c}}^{\mathsf{i}}(\mathcal{P}, \mathcal{R})$ *are the rewrite rules of $\mathcal{P}$ and there is no arc from α to β if and only if β is unreachable from α with respect to $\mathsf{NF}(\mathsf{ren}(\mathsf{rhs}(\alpha)), \mathcal{R})$. The nodes of the* c-improved innermost dependency graph $\mathsf{IDG}_{\mathsf{c}}^{\mathsf{i}}(\mathcal{P}, \mathcal{R})$ *are the rewrite rules of $\mathcal{P}$ and there is no arc from α to β if and only if β is unreachable from α with respect to $\Sigma_{\#}(\mathsf{RFC}(\mathsf{rhs}(\alpha), \mathcal{P} \cup \mathcal{R})) \cap \mathsf{NF}(\mathsf{ren}(\mathsf{rhs}(\alpha)), \mathcal{R})$.*

Lemma 29. *Let $\mathcal{P}$ and $\mathcal{R}$ be two TRSs. Then* $\mathsf{DG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P},\mathcal{R}) \supseteq \mathsf{DG}^{\mathsf{i}}(\mathcal{P},\mathcal{R})$ *and* $\mathsf{IDG}^{\mathsf{i}}_{\mathsf{c}}(\mathcal{P},\mathcal{R}) \supseteq \mathsf{IDG}^{\mathsf{i}}(\mathcal{P},\mathcal{R})$.

Proof. Straightforward adaption of the proofs of Lemmata 9 and 19. □

7 Experimental Results

The techniques described in the preceding sections are integrated in the termination prover T$_{\mathsf{T}}$T$_{\mathsf{2}}$. There are various ways to implement the (improved) dependency graph processors, ranging from checking single arcs to computing SCCs in between in order to reduce the number of arcs that have to be checked. The following procedure turned out to be the most efficient. For every term $t \in \mathsf{rhs}(\mathcal{P})$, T$_{\mathsf{T}}T_{\mathsf{2}}$ constructs a tree automaton $\mathcal{A}_t$ that is compatible with $\mathcal{R}$ and $\ell(t)$. Here $\ell(t) = \Sigma_{\#}(\mathsf{RFC}(t, \mathcal{P}\cup\mathcal{R})) \cap \Sigma(\mathsf{ren}(t))$ if $\mathcal{P}\cup\mathcal{R}$ is right-linear and $\ell(t) = \Sigma(\mathsf{ren}(t))$ otherwise. During that process it is checked if there is a term $u \in \mathsf{lhs}(\mathcal{P})$ and a substitution σ such that $u\sigma \in \mathcal{L}(\mathcal{A}_t)$. As soon as this condition evaluates to true, we add an arc from $s \to t$ to $u \to v$ for all terms s and v such that $s \to t$ and $u \to v$ are rules in $\mathcal{P}$. This procedure is repeated until for all $t \in \mathsf{rhs}(\mathcal{P})$, either $\mathcal{A}_t$ is compatible with $\mathcal{R}$ and $\ell(t)$ or an arc was added from $s \to t$ to $u \to v$ for all terms s and rules $u \to v \in \mathcal{P}$ such that $\mathsf{root}(t) = \mathsf{root}(u)$.

Another important point is the strategy used to solve compatibility violations. In T$_{\mathsf{T}}$T$_{\mathsf{2}}$ we establish paths as described in [16]. A disadvantage of this strategy is that it can happen that the completion procedure does not terminate because new states are kept being added. Hence we have to set a time limit on the involved processors to avoid that the termination proving process does not proceed beyond the calculation of (improved) dependency graphs. Alternatively one could follow the approach described in [6]. However, our experiments showed that the former approach produces better over-approximations.

Below we report on the experiments we performed with T$_{\mathsf{T}}$T$_{\mathsf{2}}$ on the 1331 TRSs in the full termination category in version 5.0 of the Termination Problem Data Base[2] that satisfy the variable condition, i.e., $\mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(l)$ for each rewrite rule $l \to r \in \mathcal{R}$. We used a workstation equipped with an Intel® Pentium™ M processor running at a CPU rate of 2 GHz and 1 GB of system memory. For all experiments we used a 60 seconds time limit.[3]

For the results in Tables 1 and 2 we used the following (improved) dependency graph processors:

- `t` A simple and fast approximation of the dependency graph processor of Theorem 4 using root comparisons to estimate the dependency graph; an arc is added from α to β if the root symbols of $\mathsf{rhs}(\alpha)$ and $\mathsf{lhs}(\beta)$ coincide.

[2] `http://www.termination-portal.org`

[3] Full experimental data, also covering the results on innermost termination in Section 6, can be found at `http://cl-informatik.uibk.ac.at/software/ttt2/experiments/bdg`.

Table 1. Dependency graph approximations I (without poly)

	without usable rules					with usable rules				
	t	e	c	r	*	t	e	c	r	*
arcs removed	55	68	68	72	73	55	67	68	73	74
# SCCs	4416	4529	4218	3786	4161	4416	4532	4179	3680	4114
# rules	24404	22198	20519	19369	21196	24404	22233	20306	19033	21093
# successes	25	60	67	176	183	25	58	67	191	195
average time	105	190	411	365	211	133	224	491	434	242
# timeouts	2	2	60	78	2	2	2	60	81	2

Table 2. Dependency graph approximations I (with poly)

	without usable rules					with usable rules				
	t	e	c	r	*	t	e	c	r	*
# successes	454	494	493	528	548	454	491	492	529	548
average time	265	194	329	139	198	262	196	352	134	191
# timeouts	14	14	71	89	14	14	14	70	91	14

- e The dependency graph processor with the estimation $\mathsf{DG}_{\mathsf{e}}(\mathcal{P},\mathcal{R})$ described in Section 5. This is the default dependency graph processor in TTT2 and AProVE.
- c The dependency graph processor with $\mathsf{DG}_{\mathsf{c}}(\mathcal{P},\mathcal{R})$ of Definition 8.
- r The improved dependency graph processor of Theorem 16 with $\mathsf{IDG}_{\mathsf{c}}(\mathcal{P},\mathcal{R})$ ($\mathsf{DG}_{\mathsf{c}}(\mathcal{P},\mathcal{R})$) for (non-)right-linear $\mathcal{P} \cup \mathcal{R}$.

After applying the above processors we use the SCC processor. In Table 2 this is additionally followed by the reduction pair processor instantiated by linear polynomial interpretations with 0/1 coefficients (poly for short) [13].

In the top half of Table 1 we list the average number of removed arcs (as percentage of the complete graph), the number of SCCs, and the number of rewrite rules in the computed SCCs. In the bottom half we list the number of successful termination attempts, the average wall-clock time needed to compute the graphs (measured in milliseconds), and the number of timeouts. In Table 2 polynomial interpretations are in effect and the average time now refers to the time to prove termination.

The power of the new processors is apparent, although the difference with e decreases when other DP processors are in place. An obvious disadvantage of the new processors is the large number of timeouts. As explained earlier, this is mostly due to the unbounded number of new states to resolve compatibility violations during tree automata completion. Modern termination tools use a variety of techniques to prove finiteness of DP problems. So it is in general more important that the graph approximations used in the (improved) dependency graph processor terminate quickly rather than that they are powerful. Since the processors c and r seem to be quite fast when they terminate, an obvious idea is to

Table 3. Dependency graph approximations II (without poly)

	without usable rules				with usable rules			
	c	s	nv	g	c	s	nv	g
arcs removed	68	63	61	48	68	63	62	48
# SCCs	4218	2294	1828	96	4179	2323	1925	270
# rules	20519	6754	5046	140	20306	6631	5133	448
# successes	67	54	67	42	67	57	64	51
average time	411	3640	3463	6734	491	3114	3817	1966
# timeouts	60	263	372	1197	60	251	349	1068

Table 4. Dependency graph approximations II (with poly)

	without usable rules				with usable rules			
	c	s	nv	g	c	s	nv	g
# successes	493	443	427	78	492	446	425	146
average time	329	2603	2143	5745	352	2396	2378	1180
# timeouts	71	264	375	1197	70	252	348	1069

equip each computation of a compatible tree automaton with a small time limit. Another natural idea is to limit the number of allowed compatibility violations. For instance, by reducing this number to 5 we can still prove termination of 141 TRSs with processor r while the number of timeouts is reduced from 78 to 21. Another strategy is to combine different graph approximations. This is shown in the columns of Tables 1 and 2 labeled $*$, which denotes the composition of t, e, c and r with a time limit of 500 milliseconds each for the latter three. We remark that the timeouts in the t and $*$ columns are solely due to the SCC processor.

A widely used approach to increase the power of DP processors is to consider only those rewrite rules of $\mathcal{R}$ which are usable [13,15]. When incorporating usable rules into the processors mentioned above, we obtain the results in the second half of Tables 1 and 2. It is interesting to observe that r (and by extension $*$) is the only processor that benefits from usable rules. This is due to the right-linearity condition in Definition 16, which obviates the addition of projection rules to DP problems.

We also implemented the approximations based on tree automata and regularity preservation described in Section 5. The results are summarized in Tables 3 and 4. It is apparent that these approximations are too time-consuming to be of any use in automatic termination provers.

One advantage of more powerful (improved) dependency graph approximations is that termination proofs can get much simpler. This is implicitly illustrated in the experiments when polynomial interpretations are in effect; using r produces the fastest termination proofs. This positive effect is also preserved if more DP processors are in effect. By incorporating the new approximations into the strategy of TTT2 used in the termination competition of 2008,[4] TTT2 can

[4] http://termcomp.uibk.ac.at

additionally prove termination of the TRS `TRCSR/ExProp7_Luc06_C.trs`: Using r, the number of arcs in the computed (improved) dependency graph is reduced from 159 to 30, resulting in a decrease of the number of SCCs from 7 to 2. This gives a speedup of about 500 milliseconds. In 2008, T$_2$ could not prove termination of this TRS because it exceeded its internal time limit of 5 seconds.

We conclude this section with the following small example.

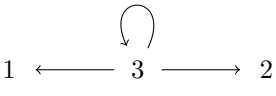
Example 30. The TRS `Endrullis/quadruple1` ($\mathcal{R}$ in the following) consists of the following rewrite rule:

$$\mathsf{p}(\mathsf{p}(\mathsf{b}(\mathsf{a}(x)),y),\mathsf{p}(z,u)) \to \mathsf{p}(\mathsf{p}(\mathsf{b}(z),\mathsf{a}(\mathsf{a}(\mathsf{b}(y)))),\mathsf{p}(u,x))$$

To prove termination of $\mathcal{R}$ using dependency pairs, we transform $\mathcal{R}$ into the initial DP problem $(\mathcal{P},\mathcal{R},\mathcal{G})$ where $\mathcal{P} = \mathsf{DP}(\mathcal{R})$ consists of the rewrite rules

$$\begin{aligned}
&1\colon \mathsf{P}(\mathsf{p}(\mathsf{b}(\mathsf{a}(x)),y),\mathsf{p}(z,u)) \to \mathsf{P}(\mathsf{p}(\mathsf{b}(z),\mathsf{a}(\mathsf{a}(\mathsf{b}(y)))),\mathsf{p}(u,x))\\
&2\colon \mathsf{P}(\mathsf{p}(\mathsf{b}(\mathsf{a}(x)),y),\mathsf{p}(z,u)) \to \mathsf{P}(\mathsf{b}(z),\mathsf{a}(\mathsf{a}(\mathsf{b}(y))))\\
&3\colon \mathsf{P}(\mathsf{p}(\mathsf{b}(\mathsf{a}(x)),y),\mathsf{p}(z,u)) \to \mathsf{P}(u,x)
\end{aligned}$$

and $\mathcal{G} = \mathcal{P} \times \mathcal{P}$. Applying the improved dependency graph processor produces the new DP problem $(\mathcal{P},\mathcal{R},\mathsf{IDG_c}(\mathcal{P},\mathcal{R}))$ where $\mathsf{IDG_c}(\mathcal{P},\mathcal{R})$ looks as follows:



After deploying the SCC processor we are left with the single DP problem $(\{3\},\mathcal{R},\{(3,3)\})$ which can easily shown to be finite by various DP processors. Using poly, T$_2$ needs about 14 milliseconds to prove termination of $\mathcal{R}$. If we use $\mathsf{DG_e}(\mathcal{P},\mathcal{R})$ instead of $\mathsf{IDG_c}(\mathcal{P},\mathcal{R})$, we do not make any progress by applying the SCC processor and thus termination of $\mathcal{R}$ cannot be shown that easily. This is reflected in the latest edition of the termination competition (2008): AProVE combined a variety of processors to infer termination within 24.31 seconds, Jambox[5] proved termination of $\mathcal{R}$ within 8.11 seconds by using linear matrix interpretations up to dimension 3, and T$_2$ used RFC match-bounds [18] to prove termination within 143 milliseconds.

References

1. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. TCS 236(1-2), 133–178 (2000)
2. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
3. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (2002), `www.grappa.univ-lille3.fr/tata`

[5] `http://joerg.endrullis.de`

4. Dershowitz, N.: Termination of linear rewriting systems (preliminary version). In: Even, S., Kariv, O. (eds.) ICALP 1981. LNCS, vol. 115, pp. 448–458. Springer, Heidelberg (1981)
5. Durand, I., Middeldorp, A.: Decidable call-by-need computations in term rewriting. I&C 196(2), 95–126 (2005)
6. Genet, T.: Decidable approximations of sets of descendants and sets of normal forms. In: Nipkow, T. (ed.) RTA 1998. LNCS, vol. 1379, pp. 151–165. Springer, Heidelberg (1998)
7. Geser, A., Hofbauer, D., Waldmann, J., Zantema, H.: Finding finite automata that certify termination of string rewriting systems. IJFCS 16(3), 471–486 (2005)
8. Geser, A., Hofbauer, D., Waldmann, J., Zantema, H.: On tree automata that certify termination of left-linear term rewriting systems. I&C 205(4), 512–534 (2007)
9. Geupel, O.: Overlap closures and termination of term rewriting systems. Report MIP-8922, Universität Passau (1989)
10. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic termination proofs in the dependency pair framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006)
11. Giesl, J., Thiemann, R., Schneider-Kamp, P.: The dependency pair framework: Combining techniques for automated termination proofs. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 301–331. Springer, Heidelberg (2005)
12. Giesl, J., Thiemann, R., Schneider-Kamp, P.: Proving and disproving termination of higher-order functions. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 216–231. Springer, Heidelberg (2005)
13. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. JAR 37(3), 155–203 (2006)
14. Hirokawa, N., Middeldorp, A.: Automating the dependency pair method. I&C 199(1-2), 172–199 (2005)
15. Hirokawa, N., Middeldorp, A.: Tyrolean termination tool: Techniques and features. I&C 205(4), 474–511 (2007)
16. Korp, M., Middeldorp, A.: Proving termination of rewrite systems using bounds. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 273–287. Springer, Heidelberg (2007)
17. Korp, M., Middeldorp, A.: Match-bounds with dependency pairs for proving termination of rewrite systems. In: Martín-Vide, C., Otto, F., Fernau, H. (eds.) LATA 2008. LNCS, vol. 5196, pp. 321–332. Springer, Heidelberg (2008)
18. Korp, M., Middeldorp, A.: Match-bounds revisited. I&C (to appear, 2009)
19. Korp, M., Sternagel, C., Zankl, H., Middeldorp, A.: Tyrolean termination tool 2. In: Proc. 20th RTA. LNCS, vol. 5595, pp. 295–304. Springer, Heidelberg (2009)
20. Kusakari, K.: Termination, AC-Termination and Dependency Pairs of Term Rewriting Systems. PhD thesis, JAIST (2000)
21. Kusakari, K., Toyama, Y.: On proving AC-termination by AC-dependency pairs. Research Report IS-RR-98-0026F, School of Information Science, JAIST (1998)
22. Middeldorp, A.: Approximating dependency graphs using tree automata techniques. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 593–610. Springer, Heidelberg (2001)
23. Middeldorp, A.: Approximations for strategies and termination. In: Proc. 2nd WRS. ENTCS, vol. 70, pp. 1–20 (2002)
24. Nagaya, T., Toyama, Y.: Decidability for left-linear growing term rewriting systems. I&C 178(2), 499–514 (2002)
25. Thiemann, R.: The DP Framework for Proving Termination of Term Rewriting. PhD thesis, RWTH Aachen (2007); available as technical report AIB-2007-17

Computing Knowledge in Security Protocols under Convergent Equational Theories*

Ştefan Ciobâcă, Stéphanie Delaune, and Steve Kremer

LSV, ENS Cachan & CNRS & INRIA, France

Abstract. In the symbolic analysis of security protocols, two classical notions of knowledge, deducibility and indistinguishability, yield corresponding decision problems. We propose a procedure for both problems under arbitrary convergent equational theories. Our procedure terminates on a wide range of equational theories. In particular, we obtain a new decidability result for a theory we encountered when studying electronic voting protocols. We also provide a prototype implementation.

1 Introduction

Cryptographic protocols are small distributed programs that use cryptographic primitives such as encryption and digital signatures to communicate securely over a network. It is essential to gain as much confidence as possible in their correctness. Therefore, symbolic methods have been developed to analyse such protocols [4,18,20]. In these approaches, one of the most important aspects is to be able to reason about the *knowledge* of the attacker.

Traditionally, the knowledge of the attacker is expressed in terms of *deducibility* (e.g. [20,10]). A message s (intuitively the secret) is said to be deducible from a set of messages φ if an attacker is able to compute s from φ. To perform this computation, the attacker is allowed, for example, to decrypt deducible messages by deducible keys.

However, deducibility is not always sufficient. Consider for example the case where a protocol participant sends over the network the encryption of one of the constants "yes" or "no" (e.g. the value of a vote). Deducibility is not the right notion of knowledge in this case, since both possible values ("yes" and "no") are indeed "known" to the attacker. In this case, a more adequate form of knowledge is *indistinguishability* (e.g. [1]): is the attacker able to distinguish between two transcripts of the protocol, one running with the value "yes" and the other one running with the value "no"?

In symbolic approaches to cryptographic protocol analysis, the protocol messages and cryptographic primitives (e.g. encryption) are generally modeled using a term algebra. This term algebra is interpreted modulo an equational theory. Using equational theories provides a convenient and flexible framework for modeling cryptographic primitives [15]. For instance, a simple equational theory for

* This work has been partly supported by the ANR SeSur AVOTÉ.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 355–370, 2009.

symmetric encryption can be specified by the equation $dec(enc(x, y), y) = x$. This equation models the fact that decryption cancels out encryption when the same key is used. Different equational theories can also be used to model randomized encryption or even more complex primitives arising when studying electronic voting protocols [16,5] or direct anonymous attestation [6]: blind signatures, trapdoor commitments, zero-knowledge proofs, ...

The two notions of knowledge that we consider do not take into account the dynamic behavior of the protocol. Nevertheless, in order to establish that two dynamic behaviors of a protocol are indistinguishable, an important subproblem is to establish indistinguishability between the sequences of messages generated by the protocol [20,2]. Indistinguishability, also called static equivalence in the applied-pi calculus framework [2], plays an important role in the study of guessing attacks (e.g. [13,7]), as well as for anonymity properties in e-voting protocols (e.g. [16,5]). This was actually the starting point of this work. During the study of e-voting protocols, we came across several equational theories for which we needed to show static equivalence while no decision procedure for deduction or static equivalence existed.

Our contributions. We provide a procedure for deduction and static equivalence which is correct, in the sense that if it terminates it gives the right answer, for any convergent equational theory. As deduction and static equivalence are undecidable for this class of equational theories [1], the procedure does not always terminate. However, we show that it does terminate for the class of *subterm convergent* equational theories (already shown decidable in [1]) and several other theories among which the theory of *trapdoor commitment* encountered in our electronic voting case studies [16].

Our second contribution is an efficient prototype implementation of this generic procedure. Our procedure relies on a simple fixed point computation based on a few saturation rules, making it convenient to implement.

Related work. Many decision procedures have been proposed for deducibility (e.g. [10,3,17]) under a variety of equational theories modeling encryption, digital signatures, exclusive OR, and homomorphic operators. Several papers are also devoted to the study of static equivalence. Most of these results introduce a new procedure for each particular theory and even in the case of the general decidability criterion given in [1,14], the algorithm underlying the proof has to be adapted for each particular theory, depending on how the criterion is fulfilled.

The first generic algorithm that has been proposed handles subterm convergent equational theories [1] and covers the classical theories for encryption and signatures. This result is encompassed by the recent work of Baudet *et al.* [9] in which the authors propose a generic procedure that works for any convergent equational theory, but which may fail or not terminate. This procedure has been implemented in the YAPA tool [8] and has been shown to terminate without failure in several cases (e.g. subterm convergent theories and blind signatures). However, due to its simple representation of deducible terms (represented by a finite set of *ground* terms), the procedure fails on several interesting equational

theories like the theory of trapdoor commitments. Our representation of deducible terms overcomes this limitation by including terms with variables which can be substituted by any deducible terms.

Due to a lack of space, the proofs are given in [12].

2 Formal Model

2.1 Term Algebras

As usual, messages will be modeled using a term algebra. Let $\mathcal{F}$ be a finite set of *function symbols* coming with an arity function $\mathrm{ar} : \mathcal{F} \to \mathbb{N}$. Function symbols of arity 0 are called *constants*. We consider several kind of *atoms*: an infinite set of *names* $\mathcal{N}$, an infinite set of *variables* $\mathcal{X}$ and a set of *parameters* $\mathcal{P}$. Names are used to represent keys, nonces and other data exchanged during a protocol run, while variables are used as usual. Parameters act as *global* variables which are used as pointers to messages exchanged during the protocol run. The essential difference between parameters and variables is that parameters can never be safely α-renamed.

The set of terms $\mathcal{T}(\mathcal{F}, \mathcal{A})$ built over $\mathcal{F}$ and the atoms in $\mathcal{A}$ is defined as

$t, t_1, \ldots ::=$	term	
$\mid$	a	atom $a \in \mathcal{A}$
$\mid$	$f(t_1, \ldots, t_k)$	application of symbol $f \in \mathcal{F}, \mathrm{ar}(f) = k$

A term t is said to be *ground* when $t \in \mathcal{T}(\mathcal{F}, \mathcal{N})$. We assume the usual definitions to manipulate terms. We write $\mathrm{fn}(t)$ (resp. $\mathrm{var}(t)$) to represent the set of (free) names (resp. variables) that occur in a term t and $\mathrm{st}(t)$ the set of its (syntactic) subterms. This notation is extended to tuples and sets of terms in the usual way. We denote by $|t|$ the *size* of t defined as the number of symbols that occur in t (variables do not count), and $\#T$ denotes the *cardinality* of the set T.

The set of positions of a term t is written $\mathrm{pos}(t) \subseteq \mathbb{N}^*$. If p is a position of t then $t|_p$ denotes the subterm of t at the position p. The term $t[u]_p$ is obtained from t by replacing the occurrence of $t|_p$ at position p with u. A *context* C is a term with (1 or more) holes and we write $C[t_1, \ldots t_n]$ for the term obtained by replacing these holes with the terms $t_1, \ldots, t_n$. A context is *public* if it only consists of function symbols and holes.

Substitutions are written $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ with $\mathrm{dom}(\sigma) = \{x_1, \ldots, x_n\}$. The application of a substitution σ to a term t is written $t\sigma$. The substitution σ is *grounding for* t if the resulting term $t\sigma$ is ground. We use the same notations for *replacements* of names and parameters by terms.

2.2 Equational Theories and Rewriting Systems

Equality between terms will generally be interpreted modulo an *equational theory*. An equational theory $\mathcal{E}$ is defined by a set of equations $M \sim N$ with $M, N \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. Equality modulo $\mathcal{E}$, written $=_{\mathcal{E}}$, is defined to be the smallest

equivalence relation on terms such that $M =_{\mathcal{E}} N$ for all $M \sim N \in \mathcal{E}$ and which is closed under substitution of terms for variables and application of contexts.

It is often more convenient to manipulate rewriting systems than equational theories. A *rewriting system* $\mathcal{R}$ is a set of rewriting rules $l \to r$ where $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $\mathrm{var}(r) \subseteq \mathrm{var}(l)$. A term t rewrites to t' by $\mathcal{R}$, denoted by $t \to_{\mathcal{R}} t'$, if there exists $l \to r \in \mathcal{R}$, a position $p \in \mathrm{pos}(t)$ and a substitution σ such that $t|_p = l\sigma$ and $t' = t[r\sigma]_p$. We denote by $\to_{\mathcal{R}}^+$ the transitive closure of $\to_{\mathcal{R}}$, $\to_{\mathcal{R}}^*$ its reflexive and transitive closure, and $=_{\mathcal{R}}$ its reflexive, symmetric and transitive closure.

A rewrite system $\mathcal{R}$ is *convergent* if it is *terminating*, i.e. there is no infinite chains $u_1 \to_{\mathcal{R}} u_2 \to_{\mathcal{R}} \ldots$, and *confluent*, i.e. for every term u such that $u \to_{\mathcal{R}}^* u_1$ and $u \to_{\mathcal{R}}^* u_2$, there exists v such that $u_1 \to_{\mathcal{R}}^* v$ and $u_2 \to_{\mathcal{R}}^* v$. A term u is in $\mathcal{R}$*-normal form* if there is no term u' such that $u \to_{\mathcal{R}} u'$. If $u \to_{\mathcal{R}}^* u'$ and u' is in $\mathcal{R}$-normal form then u' is *an* $\mathcal{R}$*-normal form of* u. When this reduced form is unique (in particular if $\mathcal{R}$ is convergent), we write $u' = u{\downarrow_{\mathcal{R}_{\mathcal{E}}}}$.

We are particularly interested in theories $\mathcal{E}$ that can be represented by a convergent rewrite system $\mathcal{R}$, i.e. theories for which there exists a convergent rewrite system $\mathcal{R}$ such that the two relations $=_{\mathcal{R}}$ and $=_{\mathcal{E}}$ coincide. Given an equational theory $\mathcal{E}$ we define the corresponding rewriting system $\mathcal{R}_{\mathcal{E}}$ by orienting all equations in $\mathcal{E}$ from left to right, i.e., $\mathcal{R}_{\mathcal{E}} = \{l \to r \mid l \sim r \in \mathcal{E}\}$. We say that $\mathcal{E}$ is *convergent* if $\mathcal{R}_{\mathcal{E}}$ is convergent.

Example 1. A classical equational theory modelling symmetric encryption is $\mathcal{E}_{enc} = \{dec(enc(x, y), y) \sim x\}$.

As a running example we consider a slight extension of this theory modelling *malleable* encryption

$$\mathcal{E}_{mal} = \mathcal{E}_{enc} \cup \{mal(enc(x, y), z) \sim enc(z, y)\}.$$

This malleable encryption scheme allows one to arbitrarily change the plaintext of an encryption. This theory certainly does not model a realistic encryption scheme but it yields a simple example of a theory which illustrates well our procedures. In particular all existing decision procedure we are aware of fail on this example. The rewriting system $\mathcal{R}_{\mathcal{E}_{mal}}$ is convergent.

From now on, we assume given a convergent equational theory $\mathcal{E}$ built over a signature $\mathcal{F}$ and represented by the convergent rewriting system $\mathcal{R}_{\mathcal{E}}$.

2.3 Deducibility and Static Equivalence

In order to describe the messages observed by an attacker, we consider the following notion of *frame* that comes from the applied-pi calculus [2].

A frame φ is a sequence of messages $u_1, \ldots, u_n$ meaning that the attacker observed each of these messages in the given order. Furthermore, we distinguish the names that the attacker knows from those that were freshly generated by others and that are *a priori* unknown by the attacker. Formally, a frame is defined as $\nu\tilde{n}.\sigma$ where $\tilde{n}$ is its set of bound names, denoted by $\mathrm{bn}(\varphi)$, and a replacement $\sigma = \{w_1 \mapsto u_1, \ldots, w_n \mapsto u_n\}$. The parameters $w_1, \ldots, w_n$ enable us to refer to $u_1, \ldots, u_n \in \mathcal{T}(\mathcal{F}, \mathcal{N})$. The *domain* $\mathrm{dom}(\varphi)$ of φ is $\{w_1, \ldots, w_n\}$.

Given terms M and N such that $\mathrm{fn}(M,N) \cap \tilde{n} = \emptyset$, we sometimes write $(M =_{\mathcal{E}} N)\varphi$ (resp. $M\varphi$) instead of $M\sigma =_{\mathcal{E}} N\sigma$ (resp. $M\sigma$).

Definition 1 (deducibility). *Let φ be a frame. A ground term t is* deducible *in $\mathcal{E}$ from φ, written $\varphi \vdash_{\mathcal{E}} t$, if there exists $M \in \mathcal{T}(\mathcal{F}, \mathcal{N} \cup \mathrm{dom}(\varphi))$, called the* recipe*, such that $\mathrm{fn}(M) \cap \mathrm{bn}(\varphi) = \emptyset$ and $M\varphi =_{\mathcal{E}} t$.*

Deducibility does not always suffice for expressing the knowledge of an attacker. For instance deducibility does not allow one to express indistinguishability between two sequences of messages. This is important when defining the confidentiality of a vote or anonymity-like properties. This motivates the following notion of static equivalence introduced in [2].

Definition 2 (static equivalence). *Let φ_1 and φ_2 be two frames such that $\mathrm{bn}(\varphi_1) = \mathrm{bn}(\varphi_2)$. They are* statically equivalent *in $\mathcal{E}$, written $\varphi_1 \approx_{\mathcal{E}} \varphi_2$, if*

- $\mathrm{dom}(\varphi_1) = \mathrm{dom}(\varphi_2)$
- *for all terms $M, N \in \mathcal{T}(\mathcal{F}, \mathcal{N} \cup \mathrm{dom}(\varphi_1))$ such that $\mathrm{fn}(M,N) \cap \mathrm{bn}(\varphi_1) = \emptyset$*

$$(M =_{\mathcal{E}} N)\varphi_1 \;\Leftrightarrow\; (M =_{\mathcal{E}} N)\varphi_2.$$

Example 2. Consider the two frames described below:

$$\varphi_1 = \nu a, k.\{w_1 \mapsto enc(a,k)\} \quad \text{and} \quad \varphi_2 = \nu a, k.\{w_1 \mapsto enc(b,k)\}.$$

We have that b and $enc(c,k)$ are deducible from φ_2 in $\mathcal{E}_{mal}$ with recipes b and $mal(w_1, c)$ respectively. We have that $\varphi_1 \not\approx_{\mathcal{E}_{mal}} \varphi_2$ since $(w_1 \neq_{\mathcal{E}_{mal}} mal(w_1, b))\varphi_1$ while $(w_1 =_{\mathcal{E}_{mal}} mal(w_1, b))\varphi_2$. Note that $\varphi_1 \approx_{\mathcal{E}_{enc}} \varphi_2$ (in the theory $\mathcal{E}_{enc}$).

3 Procedures for Deduction and Static Equivalence

In this section we describe our procedures for checking deducibility and static equivalence on convergent equational theories. After some preliminary definitions, we present the main part of our procedure, i.e. a set of saturation rules used to reach a fixed point. Then, we show how to use this saturation procedure to decide deducibility and static equivalence. Soundness and completeness of the saturation procedure are stated in Theorem 1 and detailed in Section 4.

Since both problems are undecidable for arbitrary convergent equational theories [1], our saturation procedure does not always terminate. In Section 5, we exhibit (classes of) equational theories for which the saturation terminates.

3.1 Preliminary Definitions

The main objects that will be manipulated by our procedure are *facts*, which are either *deduction facts* or *equational facts*.

Definition 3 (facts). *A* deduction fact *(resp. an* equational fact*) is an expression denoted $[U \rhd u \mid \{X_1 \rhd t_1, \ldots, X_n \rhd t_n\}]$ (resp. $[U \sim V \mid \{X_1 \rhd t_1, \ldots, X_n \rhd t_n\}]$) where $X_i \rhd t_i$ $(1 \leq i \leq n)$ are called the* side conditions *of the fact. Moreover, we assume that:*

- $u, t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{N} \cup \mathcal{X})$ *with* $\mathrm{var}(u) \subseteq \mathrm{var}(t_1, \ldots, t_n)$*;*
- $U, V \in \mathcal{T}(\mathcal{F}, \mathcal{N} \cup \mathcal{X} \cup \mathcal{P})$ *and* $X_1, \ldots, X_n \in \mathcal{X}$*;*
- $\mathrm{var}(U, V, X_1, \ldots, X_n) \cap \mathrm{var}(u, t_1, \ldots, t_n) = \emptyset$*.*

We say that a fact is solved *if* $t_i \in \mathcal{X}$ *(*$1 \leq i \leq k$*). Otherwise, it is* unsolved*. A deduction fact is* well-formed *if it is unsolved or if* $u \notin \mathcal{X}$*.*

A fact makes a statement about a frame. We read $[U \rhd u \mid \{X_1 \rhd t_1, \ldots, X_n \rhd t_n\}]$ (resp. $[U \sim V \mid \{X_1 \rhd t_1, \ldots, X_n \rhd t_n\}]$) as "$u$ is deducible with recipe U (resp. U is equal to V) if t_i is deducible with recipe X_i (for all $1 \leq i \leq n$)". For notational convenience we sometimes omit curly braces for the set of side conditions and write $[U \rhd u \mid X_1 \rhd t_1, \ldots, X_n \rhd t_n]$. When $n = 0$ we simply write $[U \rhd u]$ or $[U \sim V]$.

We say that two facts are equivalent if they are equal up to bijective renaming of variables. In the following we implicitly suppose that all operations are carried out modulo the equivalence classes. In particular set union will not add equivalent facts and inclusion will test for equivalent facts. Also, we allow *on-the-fly* renaming of variables in facts to avoid variable clashes.

We now introduce the notion of generation of a term t from a set of facts F. Intuitively, we say that a term t is generated if it can be syntactically "deduced" from F.

Definition 4 (generation). *Let* F *be a finite set of well-formed deduction facts. A term* t *is* generated by F with recipe R*, written* $\mathsf{F} \vdash^R t$*, if*

1. *either* $t = x \in \mathcal{X}$ *and* $R = x$*;*
2. *or there exist a solved fact* $[R_0 \rhd t_0 \mid X_1 \rhd x_1, \ldots, X_n \rhd x_n] \in \mathsf{F}$*, some terms* R_i *for* $1 \leq i \leq n$ *and a substitution* σ *with* $\mathrm{dom}(\sigma) \subseteq \mathrm{var}(t_0)$ *such that* $t = t_0\sigma$*,* $R = R_0[X_1 \mapsto R_1, \ldots, X_k \mapsto R_k]$*, and* $\mathsf{F} \vdash^{R_i} x_i\sigma$ *for every* $1 \leq i \leq n$*.*

A term t *is* generated by F*, written* $\mathsf{F} \vdash t$*, if there exists* R *such that* $\mathsf{F} \vdash^R t$*.*

From this definition follows a simple recursive algorithm for effectively deciding whether $\mathsf{F} \vdash t$, providing also the recipe. Termination is ensured by the fact that $|x_i\sigma| < |t|$ for every $1 \leq i \leq n$. Note that using memoization we can obtain an algorithm in polynomial time.

Given a finite set of equational facts E and terms M, N, we write $\mathsf{E} \models M \sim N$ if $M \sim N$ is a consequence, in the usual first order theory of equality, of

$$\{U\sigma \sim V\sigma \mid [U \sim V \mid X_1 \rhd x_1, \ldots, X_k \rhd x_k] \in \mathsf{E}\} \text{ where } \sigma = \{X_i \mapsto x_i\}_{1 \leq i \leq k}.$$

Note that it may be the case that $x_i = x_j$ for $i \neq j$ (whereas $X_i \neq X_j$).

3.2 Saturation Procedure

We define for each fact its *canonical form* which is obtained by first applying rule (1) and then rule (2) defined below. The idea is to ensure that each variable x_i occurs at most once in the side conditions and to get rid of those variables that do not occur in t. Unsolved deduction facts are kept unchanged.

$$(1)\quad \frac{[R \rhd t \mid X_1 \rhd x_1, \ldots, X_k \rhd x_k] \;\; \{i,j\} \subseteq \{1,\ldots,n\} \;\; j \neq i \text{ and } x_j = x_i}{[R[X_i \mapsto X_j] \rhd t \mid X_1 \rhd x_1, \ldots, X_{i-1} \rhd x_{i-1}, X_{i+1} \rhd x_{i+1}, \ldots, X_k \rhd x_k]}$$

$$(2)\quad \frac{[R \rhd t \mid X_1 \rhd x_1, \ldots, X_k \rhd x_k] \quad x_i \notin \mathrm{var}(t)}{[R \rhd t \mid X_1 \rhd x_1, \ldots, X_{i-1} \rhd x_{i-1}, X_{i+1} \rhd x_{i+1}, \ldots, X_k \rhd x_k]}$$

A *knowledge base* is a tuple (F, E) where F is a finite set of well-formed deduction facts that are in canonical form and E a finite set of equational facts.

Definition 5 (update). *Given a fact* $\mathsf{f} = [R \rhd t \mid X_1 \rhd t_1, \ldots, X_n \rhd t_n]$ *and a knowledge base* (F, E)*,* the update of (F, E) by f*, written* $(\mathsf{F}, \mathsf{E}) \oplus \mathsf{f}$*, is defined as*

$$\begin{cases} (\mathsf{F} \cup \{\mathsf{f}'\}, \mathsf{E}) & \text{if } \mathsf{f} \text{ is solved and } \mathsf{F} \not\vdash t \quad \text{useful fact} \\ \quad \text{where } \mathsf{f}' \text{ is the canonical form of } \mathsf{f} & \\ (\mathsf{F}, \mathsf{E} \cup \{[R' \sim R\{X_i \mapsto t_i\}_{1\le i\le n}]\}) & \text{if } \mathsf{f} \text{ is solved and } \mathsf{F} \vdash t \quad \text{useless fact} \\ \quad \text{where } \mathsf{F} \vdash^{R'} t & \\ (\mathsf{F} \cup \{\mathsf{f}\}, \mathsf{E}) & \text{if } \mathsf{f} \text{ is not solved} \quad \text{unsolved fact} \end{cases}$$

The choice of the recipe R' in the *useless fact* case is defined by the implementation. While this choice does not influence the correctness of the procedure, it might influence its termination as we will see later. Note that, the result of updating a knowledge base by a (possibly not well-formed and/or not canonical) fact is again a knowledge base. Facts that are not well-formed will be captured by the *useless fact* case, which adds an equational fact.

Initialisation. Given a frame $\varphi = \nu\tilde{n}.\{w_1 \mapsto t_1, \ldots, w_n \mapsto t_n\}$, our procedure starts from an *initial knowledge base* associated to φ and defined as follows:

$$\begin{array}{rll} \mathrm{Init}(\varphi) = & & (\emptyset, \emptyset) \\ & \bigoplus_{1\le i\le n} & [w_i \rhd t_i] \\ & \bigoplus_{n \in \mathrm{fn}(\varphi)} & [n \rhd n] \\ & \bigoplus_{f \in \mathcal{F}} & [f(X_1, \ldots, X_k) \rhd f(x_1, \ldots, x_k) \mid X_1 \rhd x_1, \ldots \rhd X_k \rhd x_k] \end{array}$$

Example 3. Consider the rewriting system $\mathcal{R}_{\mathcal{E}_{mal}}$ and $\varphi_2 = \nu a, k.\{w_1 \mapsto enc(b,k)\}$. The knowledge base $\mathrm{Init}(\varphi_2)$ is made up of the following deduction facts:

$$\begin{array}{llll} [\, w_1 \rhd enc(b,k) \mid \emptyset] & (\mathsf{f}_1) & [\, enc(Y_1,Y_2) \rhd enc(y_1,y_2) \mid Y_1 \rhd y_1, Y_2 \rhd y_2] & (\mathsf{f}_3) \\ [\, b \rhd b \mid \emptyset] & (\mathsf{f}_2) & [\, dec(Y_1,Y_2) \rhd dec(y_1,y_2) \mid Y_1 \rhd y_1, Y_2 \rhd y_2] & (\mathsf{f}_4) \\ & & [\, mal(Y_1,Y_2) \rhd mal(y_1,y_2) \mid Y_1 \rhd y_1, Y_2 \rhd y_2] & (\mathsf{f}_5) \end{array}$$

Saturation. The main part of our procedure consists in saturating the knowledge base $\mathrm{Init}(\varphi)$ by means of the transformation rules described in Figure 1. The rule Narrowing is designed to apply a rewriting step on an existing deduction fact. Intuitively, this rule allows us to get rid of the equational theory and nevertheless ensure that the generation of deducible terms is complete. The rule F-Solving is

Narrowing

$$\frac{\mathsf{f} = [M \rhd C[t] \mid X_1 \rhd x_1, \ldots, X_k \rhd x_k] \in \mathsf{F},\ l \to r \in \mathcal{R}_{\mathcal{E}} \quad \text{with } t \notin \mathcal{X},\ \sigma = \mathrm{mgu}(l,t) \text{ and } \mathrm{var}(\mathsf{f}) \cap \mathrm{var}(l) = \emptyset.}{(\mathsf{F},\mathsf{E}) \Longrightarrow (\mathsf{F},\mathsf{E}) \oplus \mathsf{f}_0}$$

where $\mathsf{f}_0 = [M \rhd (C[r])\sigma \mid X_1 \rhd x_1\sigma, \ldots, X_k \rhd x_k\sigma]$.

F-Solving

$$\frac{\mathsf{f}_1 = [M \rhd t \mid X_0 \rhd t_0, \ldots, X_k \rhd t_k],\ \mathsf{f}_2 = [N \rhd s \mid Y_1 \rhd y_1, \ldots, Y_\ell \rhd y_\ell] \in \mathsf{F} \quad \text{with } t_0 \notin \mathcal{X},\ \ \sigma = \mathrm{mgu}(s,t_0) \quad \text{and} \quad \mathrm{var}(\mathsf{f}_1) \cap \mathrm{var}(\mathsf{f}_2) = \emptyset.}{(\mathsf{F},\mathsf{E}) \Longrightarrow (\mathsf{F},\mathsf{E}) \oplus \mathsf{f}_0}$$

where $\mathsf{f}_0 = [M\{X_0 \mapsto N\} \rhd t\sigma \mid X_1 \rhd t_1\sigma, \ldots, X_k \rhd t_k\sigma, Y_1 \rhd y_1\sigma, \ldots, Y_\ell \rhd y_\ell\sigma]$.

Unifying

$$\frac{\mathsf{f}_1 = [M \rhd t \mid X_1 \rhd x_1, \ldots, X_k \rhd x_k],\ \mathsf{f}_2 = [N \rhd s \mid Y_1 \rhd y_1, \ldots, Y_\ell \rhd y_\ell] \in \mathsf{F} \quad \text{with } \sigma = \mathrm{mgu}(s,t) \quad \text{and} \quad \mathrm{var}(\mathsf{f}_1) \cap \mathrm{var}(\mathsf{f}_2) = \emptyset.}{(\mathsf{F},\mathsf{E}) \Longrightarrow (\mathsf{F},\mathsf{E} \cup \{\mathsf{f}_0\})}$$

where $\mathsf{f}_0 = [M \sim N \mid \{X_i \rhd x_i\sigma\}_{1 \le i \le k} \cup \{Y_i \rhd y_i\sigma\}_{1 \le i \le \ell}]$.

E-Solving

$$\frac{\mathsf{f}_1 = [U \sim V \mid Y \rhd s, X_1 \rhd t_1, \ldots, X_k \rhd t_k] \in \mathsf{E}, \mathsf{f}_2 = [M \rhd t \mid Y_1 \rhd y_1, \ldots, Y_\ell \rhd y_\ell\} \in \mathsf{F} \quad \text{with } s \notin \mathcal{X},\ \sigma = \mathrm{mgu}(s,t) \text{ and } \mathrm{var}(\mathsf{f}_1) \cap \mathrm{var}(\mathsf{f}_2) = \emptyset.}{(\mathsf{F},\mathsf{E}) \Longrightarrow (\mathsf{F},\mathsf{E} \cup \{\mathsf{f}_0\})}$$

where $\mathsf{f}_0 = [U\{Y \mapsto M\} \sim V\{Y \mapsto M\} \mid \{X_i \rhd t_i\sigma\}_{1 \le i \le k} \cup \{Y_i \rhd y_i\sigma\}_{1 \le i \le \ell}]$.

Fig. 1. Saturation rules

used to instantiate an unsolved side condition of an existing deduction fact. Unifying and E-Solving add equational facts which remember when different recipes for the same term exist.

Note that this procedure may not terminate and that the fixed point may not be unique.

We write $\Longrightarrow^*$ for the reflexive and transitive closure of $\Longrightarrow$.

Example 4. Continuing Example 3, we illustrate the saturation procedure. We can apply the rule Narrowing on f_4 and on the rewrite rule $dec(enc(x,y),y) \to x$, as well as on f_5 and the rewrite rule $mal(enc(x,y),z) \to enc(z,y)$, thereby adding the facts

$$[dec(Y_1,Y_2) \rhd x \mid Y_1 \rhd enc(x,y), Y_2 \rhd y] \quad (\mathsf{f}_6)$$
$$[mal(Y_1,Y_2) \rhd enc(z,y) \mid Y_1 \rhd enc(x,y), Y_2 \rhd z] \quad (\mathsf{f}_7)$$

The facts f_6 and f_7 are not solved and we can apply the rule F-Solving with f_1, thereby adding the facts:

$$[dec(w_1,Y_2) \rhd b \mid Y_2 \rhd k] \ (\mathsf{f}_8) \qquad [mal(w_1,Y_2) \rhd enc(z,k) \mid Y_2 \rhd z] \ (\mathsf{f}_9)$$

Rule Unifying can be used on facts $\mathsf{f}_1/\mathsf{f}_3$, $\mathsf{f}_3/\mathsf{f}_9$ as well as $\mathsf{f}_1/\mathsf{f}_9$ to add equational facts. This third case allows one to obtain $\mathsf{f}_{10} = [w_1 \sim mal(w_1, Y_2) \mid Y_2 \rhd b]$ which can be solved (using E-Solving with f_2) to obtain $\mathsf{f}_{11} = [w_1 \sim mal(w_1, b)]$. Because of lack of space we do not detail the remaining rule applications. When reaching a fixed point the knowledge base contains the solved facts f_9 and f_{11} as well as those in $\mathrm{Init}(\varphi_2)$.

We now state the soundness and completeness of our transformation rules. The technical lemmas used to prove this result are detailed in Section 4.

Theorem 1 (soundness and completeness). *Let φ be a frame and (F, E) be a saturated knowledge base such that $\mathrm{Init}(\varphi) \Longrightarrow^* (\mathsf{F}, \mathsf{E})$. Let $t \in \mathcal{T}(\mathcal{F}, \mathcal{N})$ and $\mathsf{F}^+ = \mathsf{F} \cup \{[n \rhd n] \mid n \in \mathrm{fn}(t) \setminus \mathrm{bn}(\varphi)\}$. We have that:*

1. *For all $M \in \mathcal{T}(\mathcal{F}, \mathcal{N} \cup \mathrm{dom}(\varphi))$ such that $\mathrm{fn}(M) \cap \mathrm{bn}(\varphi) = \emptyset$, we have that*

$$M\varphi =_{\mathcal{E}} t \Leftrightarrow \exists N,\ \mathsf{E} \models M \sim N \text{ and } \mathsf{F}^+ \vdash^N t{\downarrow_{\mathcal{R}_\mathcal{E}}}$$

2. *For all $M, N \in \mathcal{T}(\mathcal{F}, \mathcal{N} \cup \mathrm{dom}(\varphi))$ such that $\mathrm{fn}(M, N) \cap \mathrm{bn}(\varphi) = \emptyset$, we have*

$$(M =_{\mathcal{E}} N)\varphi \ \Leftrightarrow\ \mathsf{E} \models M \sim N.$$

3.3 Application to Deduction and Static Equivalence

Procedure for deduction. Let φ be a frame and t be a ground term. The procedure for checking $\varphi \vdash_{\mathcal{E}} t$ is described bellow. Its correctness is a direct consequence of Theorem 1, Item 1.

1. Apply the saturation rules to obtain (if any) a saturated knowledge base (F, E) such that $\mathrm{Init}(\varphi) \Longrightarrow^* (\mathsf{F}, \mathsf{E})$. Let $\mathsf{F}^+ = \mathsf{F} \cup \{[n \rhd n] \mid n \in \mathrm{fn}(t) \setminus \mathrm{bn}(\varphi)\}$.
2. Return *yes* if there exists N such that $\mathsf{F}^+ \vdash^N t{\downarrow_{\mathcal{R}_\mathcal{E}}}$ (that is, the $\mathcal{R}_\mathcal{E}$-normal form of t is generated by F with recipe N); otherwise return *no*.

Example 5. We continue our running example. Let (F, E) be the knowledge base obtained from $\mathrm{Init}(\varphi_2)$ described in Example 4. We show that $\varphi_2 \vdash enc(c, k)$ and $\varphi_2 \vdash b$. Indeed we have that $\mathsf{F} \cup \{[c \rhd c]\} \vdash^{mal(w_1, c)} enc(c, k)$ using facts f_9 and $[c \rhd c]$, and $\mathsf{F} \vdash^b b$ using fact f_2.

Procedure for static equivalence. Let φ_1 and φ_2 be two frames. The procedure for checking $\varphi_1 \approx_{\mathcal{E}} \varphi_2$ runs as follows:

1. Apply the transformation rules to obtain (if possible) two saturated knowledge bases $(\mathsf{F}_i, \mathsf{E}_i)$, $i = 1, 2$ such that $\mathrm{Init}(\varphi_i) \Longrightarrow^* (\mathsf{F}_i, \mathsf{E}_i)$, $i = 1, 2$.
2. For $\{i, j\} = \{1, 2\}$, for every solved fact $[M \sim N \mid X_1 \rhd x_1, \ldots, X_k \rhd x_k]$ in E_i, check if $(M\{X_1 \mapsto x_1, \ldots, X_k \mapsto x_k\} =_{\mathcal{E}} N\{X_1 \mapsto x_1, \ldots, X_k \mapsto x_k\})\varphi_j$.
3. If so return *yes*; otherwise return *no*.

Proof. (Sketch) If the algorithm returns *no*, then there exists an equation that holds in one frame but not in the other; therefore, the two frames are not statically equivalent.

Assume that the algorithm returns *yes*. Let $M \sim N$ be an arbitrary equation that holds in φ_1. By Theorem 1, Item 2, we have that $\mathsf{E}_1 \models M \sim N$. As all equations in E_1 also hold in φ_2, and because $\mathsf{E}_1 \models M \sim N$, it follows that $M \sim N$ holds in φ_2. We have shown that all equations that hold in φ_1 also hold in φ_2. Similarly, all equations that hold in φ_2 hold in φ_1 and therefore the two frames are statically equivalent. □

Example 6. Consider again the frames φ_1 and φ_2 which are not statically equivalent (see Example 2). Our procedure answers *no* since $[mal(w_1, b) \sim w_1] \in \mathsf{E}_2$ whereas $(mal(w_1, b) \neq_{\mathcal{E}_{mal}} w_1)\varphi_1$.

4 Soundness and Completeness

In this section we give the key results to prove Theorem 1. The soundness of our saturation procedure relies on Lemma 1 whereas its completeness is more involved: the key propositions are stated below.

Intuitively Lemma 1 states that any ground term which can be generated is indeed deducible. Similarly all equations which are consequences of the knowledge base are true equations in the initial frame. The soundness of our saturation procedure can be easily derived from this lemma.

Lemma 1 (soundness). *Let φ be a frame and (F, E) be a knowledge base such that $\mathrm{Init}(\varphi) \Longrightarrow^* (\mathsf{F}, \mathsf{E})$. Let $t \in \mathcal{T}(\mathcal{F}, \mathcal{N})$, $M, N \in \mathcal{T}(\mathcal{F}, \mathcal{N} \cup \mathrm{dom}(\varphi))$ such that $\mathrm{fn}(M, N) \cap \mathrm{bn}(\varphi) = \emptyset$, and $\mathsf{F}^+ = \mathsf{F} \cup \{[n \rhd n] \mid n \in \mathrm{fn}(t) \setminus \mathrm{bn}(\varphi)\}$. We have that:*

1. $\mathsf{F}^+ \vdash^M t \;\Rightarrow\; M\varphi =_{\mathcal{E}} t$; *and*
2. $\mathsf{E} \models M \sim N \;\Rightarrow\; (M =_{\mathcal{E}} N)\varphi$.

We now give two propositions that are used to show the completeness of the saturation rules. The first one states that whenever there exist two recipes to generate a ground term from F then the equation on the two recipes is a consequence of E.

Proposition 1 (completeness, equation). *Let (F, E) be a saturated knowledge base, and M, N be two terms such that $\mathsf{F} \vdash^M t$ and $\mathsf{F} \vdash^N t$ for some ground term t. Then, we have that $\mathsf{E} \models M \sim N$.*

Next we show that whenever a ground term (not necessarily in normal form) can be generated then its normal form can also be generated and there exists an equation on the two recipes.

Proposition 2 (completeness, reduction). *Let (F, E) be a saturated knowledge base, M a term and t a ground term such that $\mathsf{F} \vdash^M t$ and $t{\downarrow_{\mathcal{R}_\mathcal{E}}} \neq t$. Then there exists M' and t' such that $\mathsf{F} \vdash^{M'} t'$ with $t \to^+_{\mathcal{R}_\mathcal{E}} t'$ and $\mathsf{E} \models M \sim M'$.*

Relying on these propositions, we can show completeness of our saturation procedure (i.e. $\Rightarrow$ of Theorem 1).

1. To prove Item 1, we first observe that if t is deducible from φ modulo $\mathcal{E}$ then $\mathsf{F}^+ \vdash^{M'} t_0$ for some M' and t_0 such that $\mathsf{E} \models M \sim M'$ and $t_0 \to^* t{\downarrow_{\mathcal{R}_\mathcal{E}}}$. Actually M' differs from M by the fact that some public names that do not occur in the knowledge base are replaced by fresh variables. Then, we rely on Proposition 2 and we show the result by induction on t_0 equipped with the order $<$ induced by the rewriting relation ($t < t'$ iff $t \to^+ t'$).
2. Now, to prove Item 2, we apply the result shown in item 1 on $M\varphi =_\mathcal{E} t$ and $N\varphi =_\mathcal{E} t$ where $t = M\varphi{\downarrow_{\mathcal{R}_\mathcal{E}}} = N\varphi{\downarrow_{\mathcal{R}_\mathcal{E}}}$. We deduce that there exist M' and N' such that $\mathsf{E} \models M \sim M'$, $\mathsf{F}^+ \vdash^{M'} t$, $\mathsf{E} \models N \sim N'$, and $\mathsf{F}^+ \vdash^{N'} t$. Then, Proposition 1 allows one to deduce that $\mathsf{E} \models M' \sim N'$, thus $\mathsf{E} \models M \sim N$.

5 Termination

As already announced the saturation process will not always terminate.

Example 7. Consider the convergent rewriting system consisting of the single rule $f(g(x)) \to g(h(x))$ and the frame $\phi = \nu a.\{w_1 \mapsto g(a)\}$. We have that

$$\mathrm{Init}(\varphi) \supseteq \{[w_1 \rhd g(a)],\ [f(X) \rhd f(x) \mid X \rhd x]\}.$$

By Narrowing we can add the fact $\mathsf{f}_1 = [f(X) \rhd g(h(x)) \mid X \rhd g(x)]$. Then we can apply F-Solving to solve its side condition $X \rhd g(x)$ with the fact $[w_1 \rhd g(a)]$ yielding the solved fact $[f(w_1) \rhd g(h(a))]$. Now, applying iteratively F-Solving on f_1 and the newly generated fact, we generate an infinity of solved facts of the form $[f(\ldots f(w_1)\ldots) \rhd g(h(\ldots h(a)\ldots))]$. Intuitively, this happens because our symbolic representation is unable to express that the function h can be nested an unbounded number of times when it occurs under an application of g.

The same kind of limitation already exists in the procedure implemented in YAPA [9]. However, our symbolic representation, that manipulates terms that are not necessarily ground and facts with side conditions, allows us to go beyond YAPA. We are able for instance to treat equational theories such as malleable encryption and trapdoor commitment.

5.1 Generic Method for Proving Termination

We provide a generic method for proving termination, which we instantiate in the following section on several examples.

In order to prove that the saturation algorithm terminates, we require that the update function $\oplus$ be *uniform*: i.e., the same recipe R' be used for all useless solved deduction facts that have the same canonical form. Note that the soundness and completeness of the algorithm does not depend on the choice of the recipe R' when updating the knowledge base with a useless fact (*cf.* Definition 5).

Definition 6 (projection). *We define the* projection *of a deduction fact* $\mathsf{f} = [R \rhd t \mid X_1 \rhd t_1, \ldots, X_n \rhd t_n]$ *as* $\hat{\mathsf{f}} = [t \mid \{t_1, \ldots, t_n\}]$. *We extend the projection to sets of facts* F *and define* $\hat{\mathsf{F}} = \{\hat{\mathsf{f}} \mid \mathsf{f} \in \mathsf{F}\}$.

We identify projections which are equal up to bijective renaming of variables and we sometimes omit braces for the side conditions.

Proposition 3 (generic termination). *The saturation algorithm terminates if* $\oplus$ *is uniform and there exist some functions* $\mathcal{Q}$, m_f, m_e *and some well-founded orders* $<_\mathsf{f}$ *and* $<_\mathsf{e}$ *such that for all frames* φ, *and for all* (F, E) *such that* $\mathrm{Init}(\varphi) \Longrightarrow^* (\mathsf{F}, \mathsf{E})$, *we have that:*

1. $\{\hat{\mathsf{f}} \mid \mathsf{f} \in \mathsf{F}$ *and* f *is a solved deduction fact*$\} \subseteq \mathcal{Q}(\varphi)$ *and* $\mathcal{Q}(\varphi)$ *is finite;*
2. $\mathsf{m}_\mathsf{f}(\mathsf{f}_0) <_\mathsf{f} \mathsf{m}_\mathsf{f}(\mathsf{f}_1)$ *where* f_0, f_1 *are defined as in rule* F-Solving;
3. $\mathsf{m}_\mathsf{e}(\mathsf{f}_0) <_\mathsf{e} \mathsf{m}_\mathsf{e}(\mathsf{f}_1)$ *where* f_0, f_1 *are defined as in rule* E-Solving.

5.2 Applications

We now give several examples for which the saturation procedure indeed terminates. For each of these theories the definition of the function $\mathcal{Q}$ relies on the following notion of *extended subterms*.

Definition 7 (extended subterm). *Let* t *be a term; its set of* extended subterms $\mathrm{st}_{\mathcal{R}_\mathcal{E}}(t)$ *(w.r.t.* $\mathcal{E}$*) is the smallest set such that:*

1. $t \in \mathrm{st}_{\mathcal{R}_\mathcal{E}}(t)$,
2. $f(t_1, \ldots, t_k) \in \mathrm{st}_{\mathcal{R}_\mathcal{E}}(t)$ *implies* $t_1, \ldots, t_k \in \mathrm{st}_{\mathcal{R}_\mathcal{E}}(t)$,
3. $t' \in \mathrm{st}_{\mathcal{R}_\mathcal{E}}(t)$ *and* $t' \to_{\mathcal{R}_\mathcal{E}} t''$ *implies* $t'' \in \mathrm{st}_{\mathcal{R}_\mathcal{E}}(t)$.

This notation is extended to frames in the usual way.

All the examples in this section rely on the same measures m_f and m_e. Let $\{X_1 \rhd t_1, \ldots, X_n \rhd t_n\}$ be the set of side conditions of a fact f. We define $\mathsf{m}_\mathsf{f}(\mathsf{f}) = (\#\,\mathrm{var}(t_1, \ldots, t_n), \sum_{1 \leq i \leq n} |t_i|)$ and $<_\mathsf{f}$ is the lexicographical order on ordered pairs of integers. The measure m_e and the order $<_\mathsf{e}$ are defined in the same way.

We now present the class of subterm convergent equational theories as well as the theories for malleable encryption and trap-door commitment.

Subterm Convergent Equational Theories. Abadi and Cortier [1] have shown that deduction and static equivalence are decidable for *subterm convergent* equational theories in polynomial time. We retrieve the same results with our algorithm. An equational theory $\mathcal{E}$ is subterm convergent if $\mathcal{R}_\mathcal{E}$ is convergent and for every rule $l \to r \in \mathcal{R}_\mathcal{E}$, we have that either r is a strict subterm of l, or r is a ground term in $\mathcal{R}_\mathcal{E}$-normal form.

The termination proof for this class relies on the function $\mathcal{Q}$ where $\mathcal{Q}(\varphi)$ is defined as the smallest set that contains

1. $[t \mid \emptyset]$, where $t \in \mathrm{st}_{\mathcal{R}_\mathcal{E}}(\varphi)$;
2. $[f(x_1, \ldots, x_k) \mid x_1, \ldots, x_k]$, where $ar(f) = k$.

Malleable Encryption. We also obtain termination for the equational theory $\mathcal{E}_{mal}$ described in Example 1. This is a toy example that does not fall in the class studied in [1]. Indeed, this theory is not *locally stable*: the set of terms in normal form deducible from a frame φ cannot always be obtained by applying public contexts over a finite set (called $sat(\varphi)$ in [1]) of ground terms.

As a witness consider the frame $\varphi_2 = \nu a, k.\{w_1 \mapsto enc(b, k\}$ introduced in Example 2. Among the terms that are deducible from φ_2, we have those of the form $enc(t, k)$ where t represents any term deducible from φ_2. From this observation, it is easy to see that $\mathcal{E}_{mal}$ is not locally stable.

Our procedure does not have this limitation. A prerequisite for termination is that the set of terms in normal form deducible from a frame is exactly the set of terms obtained by nesting in all possible ways a finite set of contexts. The theory $\mathcal{E}_{mal}$ falls in this class. In particular, for the frame φ_2, our procedure produces the fact $\mathsf{f}_9 = [mal(w_1, Y_2) \rhd enc(z, k) \mid Y_2 \rhd z]$ allowing us to capture all the terms of the form $enc(t, k)$ by the means of a single deduction fact.

The termination proof relies on the function $\mathcal{Q}$ where $\mathcal{Q}(\varphi)$ is defined as the smallest set that contains:

1. $[t \mid \emptyset]$, for every $t \in \mathrm{st}_{\mathcal{R}_\mathcal{E}}(\varphi)$;
2. $[f(x_1, x_2) \mid x_1, x_2]$, where $f \in \{enc, dec, mal\}$;
3. $[enc(x, t) \mid x]$, if there exists t' such that $enc(t', t) \in \mathrm{st}_{\mathcal{R}_\mathcal{E}}(\varphi)$.

Trap-Door Commitment. The following convergent equational theory $\mathcal{E}_{td}$ is a model for trap-door commitment:

$$open(td(x, y, z), y) = x \qquad td(x_2, f(x_1, y, z, x_2), z) = td(x_1, y, z)$$
$$open(td(x_1, y, z), f(x_1, y, z, x_2)) = x_2 \qquad f(x_2, f(x_1, y, z, x_2), z, x_3) = f(x_1, y, z, x_3)$$

As said in the introduction, we encountered this equational theory when studying electronic voting protocols. The term $td(m, r, td)$ models the commitment of the message m under the key r using an additional trap-door td. Such a commitment scheme allows a voter who has performed a commitment to open it in different ways using its trap-door. Hence, trap-door bit commitment $td(v, r, td)$ does not bind the voter to the vote v. This is useful to ensure privacy-type properties in e-voting and in particular receipt-freeness [19]. With such a scheme, even if a coercer requires the voter to reveal his commitment, this does not give any useful information to the coercer as the commitment can be viewed as the commitment of any vote (depending on the key that will be used to open it).

For the same reason as $\mathcal{E}_{mal}$, the theory of trap-door commitment described below cannot be handled by the algorithms described in [1,9]. Our termination proof relies on the function $\mathcal{Q}$ where $\mathcal{Q}(\varphi)$ is the smallest set that contains:

1. $[t \mid \emptyset]$, for every $t \in \mathrm{st}_{\mathcal{R}_\mathcal{E}}(\varphi)$;
2. $[td(t_1, r, tp) \mid \emptyset]$ such that $f(t_1, r, tp, t_2) \in \mathrm{st}_{\mathcal{R}_\mathcal{E}}(\varphi)$ for some t_2;
3. $[g(x_1, \ldots, x_k) \mid x_1, \ldots, x_k]$, where $g \in \{open, td, f\}$ and $ar(g) = k$;
4. $[f(t_1, r, tp, x) \mid x]$, such that $f(t_1, r, tp, t_2) \in \mathrm{st}_{\mathcal{R}_\mathcal{E}}(\varphi)$ for some t_2.

Termination of our procedure is also ensured for theories such as blind signature and addition as defined in [1].

5.3 Going beyond with Fair Strategies

In [1] decidability is also shown for an equational theory modeling homomorphic encryption. For our procedure to terminate on this theory we use a particular saturation strategy.

Homomorphic Encryption. We consider the theory $\mathcal{E}_{\mathsf{hom}}$ of homomorphic encryption that has been studied in [1,9].

$$fst(pair(x,y)) = x \quad snd(pair(x,y)) = y \quad dec(enc(x,y),y) = x$$
$$enc(pair(x,y),z) = pair(enc(x,z),enc(y,z))$$
$$dec(pair(x,y),z) = pair(dec(x,z),dec(y,z))$$

In general, our algorithm does not terminate under this equational theory. Consider for instance the frame $\phi = \nu a, b.\{w_1 \mapsto pair(a,b)\}$. We have that:

$$\text{Init}(\varphi) \supseteq \{[w_1 \rhd pair(a,b)],\ [enc(X,Y) \rhd enc(x,y) \ \mid\ X \rhd x,\ Y \rhd y]\}.$$

As in Example 7 we can obtain an unbounded number of solved facts whose projections are of the form:

$$[pair(enc(\dots enc(a,z_1)\dots,z_n), enc(\dots enc(b,z_1)\dots,z_n)) \mid z_1,\dots,z_n].$$

However, we can guarantee termination by using a *fair* saturation strategy. We say that a saturation strategy is fair if whenever a rule instance is enabled it will eventually be taken.

Indeed in the above example using a fair strategy we will eventually add the facts $[fst(w_1) \rhd a]$ and $[snd(w_1) \rhd b]$. Now the "problematic" facts described above become useless and are not added to the knowledge base anymore. One may note that a fair strategy does not guarantee termination in Example 7 (intuitively, because the function g is one-way and a is not deducible in that example).

The proof of termination will as for the previous theories define functions $\mathcal{Q}$, $\mathsf{m_f}$ and $\mathsf{m_e}$. The main argument of the proof is the observation that due to fairness only a finite number of solved fact not in $\mathcal{Q}(\varphi)$ can be added.

6 Implementation

A C++ implementation of the procedures described in this paper is provided in the KiSs (Knowledge in Security protocols) tool [11]. The tool implements a uniform $\oplus$ and contains several optimizations. First, as the order of solving side conditions is not important, we always solve the first unsolved side condition rather than considering all the combinations. We also use DAG representation of terms and specialized F-Solving and E-Solving rules for solving ground side conditions. Indeed, by checking whether the side condition is generated or not we know whether solving it will eventually produce a solved fact. Checking generation takes only polynomial time. This makes the procedure terminate in

polynomial time for subterm convergent equational theories, and the theories $\mathcal{E}_{blind}$, $\mathcal{E}_{mal}$ and $\mathcal{E}_{td}$.

The performance of the tool is comparable to the YAPA tool [8,9] and on most examples the tool terminates in less than a second. In [9] a family of contrived examples is presented to diminish the performance of YAPA, exploiting the fact that YAPA does not implement DAG representations of terms and recipes, as opposed to KiSs. As expected, KiSs indeed performs better on these examples.

Regarding termination, our procedure terminates on all examples of equational theories presented in [9]. In addition, our tool terminates on the theories $\mathcal{E}_{mal}$ and $\mathcal{E}_{td}$ whereas YAPA does not. In [9] a class of equational theories for which YAPA terminates is identified and it is not known whether our procedure terminates. YAPA may also terminate on examples outside this class. Hence the question whether termination of our procedures encompasses termination of YAPA is still open.

7 Conclusion

We have proposed a procedure for deduction and for static equivalence for convergent equational theories. Our procedure terminates for a wide range of equational theories. In particular, we obtain a new decidability result for the theory of trapdoor commitment.

As future work, we indent to extend our approach in order to handle associative commutative operators (like xor) and the active case of the two problems.

References

1. Abadi, M., Cortier, V.: Deciding knowledge in security protocols under equational theories. Theoretical Computer Science 387(1-2), 2–32 (2006)
2. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: Proc. 28th ACM Symp. on Principles of Programming Languages (2001)
3. Anantharaman, S., Narendran, P., Rusinowitch, M.: Intruders with caps. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 20–35. Springer, Heidelberg (2007)
4. Armando, A., et al.: The AVISPA Tool for the automated validation of internet security protocols and applications. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 281–285. Springer, Heidelberg (2005)
5. Backes, M., Hritcu, C., Maffei, M.: Automated verification of remote electronic voting protocols in the applied pi-calculus. In: Proc. 21st IEEE Computer Security Foundations Symposium (CSF 2008) (2008)
6. Backes, M., Maffei, M., Unruh, D.: Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In: IEEE Symposium on Security and Privacy (S&P 2008). IEEE Comp. Soc. Press, Los Alamitos (2008)
7. Baudet, M.: Deciding security of protocols against off-line guessing attacks. In: 12th ACM Conference on Computer and Communications Security (CCS 2005) (2005)
8. Baudet, M.: YAPA (Yet Another Protocol Analyzer) (2008), `http://www.lsv.ens-cachan.fr/~baudet/yapa/index.html`

9. Baudet, M., Cortier, V., Delaune, S.: YAPA: A generic tool for computing intruder knowledge. In: Treinen, R. (ed.) Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA 2009), Brasília, Brazil, June-July 2009. LNCS. Springer, Heidelberg (to appear, 2009)
10. Chevalier, Y.: Résolution de problèmes d'accessibilité pour la compilation et la validation de protocoles cryptographiques. PhD thesis, Univ. Henri Poincaré (2003)
11. Ciobâcă, Ş.: KiSs (2009), `http://www.lsv.ens-cachan.fr/~ciobaca/kiss`
12. Ciobâcă, Ş., Delaune, S., Kremer, S.: Computing knowledge in security protocol under convergent equational theories. Research Report LSV-09-05, Laboratoire Spécification et Vérification, ENS Cachan, France, March 2009, 42 p. (2009)
13. Corin, R., Doumen, J., Etalle, S.: Analysing password protocol security against off-line dictionary attacks. In: Proc. 2nd International Workshop on Security Issues with Petri Nets and other Computational Models (WISP 2004). ENTCS (2004)
14. Cortier, V., Delaune, S.: Deciding knowledge in security protocols for monoidal equational theories. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 196–210. Springer, Heidelberg (2007)
15. Cortier, V., Delaune, S., Lafourcade, P.: A survey of algebraic properties used in cryptographic protocols. Journal of Computer Security 14(1), 1–43 (2006)
16. Delaune, S., Kremer, S., Ryan, M.D.: Verifying privacy-type properties of electronic voting protocols. Journal of Computer Security (to appear, 2008)
17. Lafourcade, P., Lugiez, D., Treinen, R.: Intruder deduction for the equational theory of Abelian groups with distributive encryption. Information and Computation 205(4), 581–623 (2007)
18. Millen, J., Shmatikov, V.: Constraint solving for bounded-process cryptographic protocol analysis. In: Proc. 8th ACM Conference on Computer and Communications Security (CCS 2001) (2001)
19. Okamoto, T.: Receipt-free electronic voting schemes for large scale elections. In: Christianson, B., Lomas, M. (eds.) Security Protocols 1997. LNCS, vol. 1361. Springer, Heidelberg (1998)
20. Rusinowitch, M., Turuani, M.: Protocol insecurity with finite number of sessions and composed keys is NP-complete. Theoretical Computer Science 299, 451–475 (2003)

Complexity of Fractran and Productivity

Jörg Endrullis[1], Clemens Grabmayer[2], and Dimitri Hendriks[1]

[1] Vrije Universiteit Amsterdam, Department of Computer Science
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
joerg@few.vu.nl, diem@cs.vu.nl
[2] Universiteit Utrecht, Department of Philosophy
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands
clemens@phil.uu.nl

Abstract. In functional programming languages the use of infinite structures is common practice. For total correctness of programs dealing with infinite structures one must guarantee that every finite part of the result can be evaluated in finitely many steps. This is known as productivity. For programming with infinite structures, productivity is what termination in well-defined results is for programming with finite structures.

Fractran is a simple Turing-complete programming language invented by Conway. We prove that the question whether a Fractran program halts on all positive integers is Π^0_2-complete. In functional programming, productivity typically is a property of individual terms with respect to the inbuilt evaluation strategy. By encoding Fractran programs as specifications of infinite lists, we establish that this notion of productivity is Π^0_2-complete even for some of the most simple specifications. Therefore it is harder than termination of individual terms. In addition, we explore generalisations of the notion of productivity, and prove that their computational complexity is in the analytical hierarchy, thus exceeding the expressive power of first-order logic.

1 Introduction

For programming with infinite structures, productivity is what termination is for programming with finite structures. In lazy functional programming languages like Haskell, Miranda or Clean the use of data structures, whose intended semantics is an infinite structure, is common practice. Programs dealing with such infinite structures can very well be terminating. For example, consider the Haskell program implementing a version of Eratosthenes' sieve:

```
prime n = primes !! (n-1)
primes = sieve [2..]
sieve (n:xs) = n:(sieve (filter (\m -> m `mod` n /= 0) xs))
```

where `prime n` returns the n-th prime number for every $n \geq 1$. The function `prime` is terminating, despite the fact that it contains a call to the non-terminating function `primes` which, in the limit, rewrites to the infinite list of

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 371–387, 2009.

prime numbers in ascending order. To make this possible, the strategy with respect to which the terms are evaluated is crucial. Obviously, we cannot fully evaluate `primes` before extracting the n-th element. For this reason, lazy functional languages typically use a form of outermost-needed rewriting where only needed, finite parts of the infinite structure are evaluated, see for example [13].

Productivity captures the intuitive notion of unlimited progress, of 'working' programs producing values indefinitely, programs immune to livelock and deadlock, like `primes` above. A recursive specification is called productive if not only can the specification be evaluated continually to build up an infinite normal form, but this infinite expression is also meaningful in the sense that it represents an infinite object from the intended domain. The study of productivity (of stream specifications in particular) was pioneered by Sijtsma [15]. More recently, a decision algorithm for productivity of stream specifications from an expressive syntactic format has been developed [6] and implemented [4].

We consider various variants of the notion of productivity and pinpoint their computational complexity in the arithmetical and analytical hierarchy. In functional programming, expressions are evaluated according to an inbuilt evaluation strategy. This gives rise to *productivity with respect to an evaluation strategy*. We show that this property is Π^0_2-complete (for individual terms) using a standard encoding of Turing machines into term rewriting systems. Next, we explore two generalisations of this concept: *strong* and *weak productivity*. Strong productivity requires every outermost-fair rewrite sequence to 'end in' a constructor normal form, whereas weak productivity demands only the existence of a rewrite sequence to a constructor normal form. As it turns out, these properties are of analytical complexity: Π^1_1 and Σ^1_1-complete, respectively.

Finally, we encode Fractran programs into stream specifications. In contrast to the encoding of Turing machines, the resulting specifications are of a very simple form and do not involve any computation on the elements of the stream. We show that the uniform halting problem of Fractran programs is Π^0_2-complete. (Although Turing-completeness of Fractran is folklore, the exact complexity has not yet been investigated before.) Consequently we obtain a strengthening of the earlier mentioned Π^0_2-completeness result for productivity.

Fractran [2] is a remarkably simple Turing-complete programming language invented by the mathematician John Horton Conway. A Fractran program is a finite list of fractions $\frac{p_1}{q_1}, \ldots, \frac{p_k}{q_k}$. Starting with a positive integer n_0, the algorithm successively calculates n_{i+1} by multiplying n_i with the first fraction that yields an integer again. The algorithm halts if there is no such fraction.

To illustrate the algorithm we consider an example of Conway from [2]:

$$\frac{17}{91}, \frac{78}{85}, \frac{19}{51}, \frac{23}{38}, \frac{29}{33}, \frac{77}{29}, \frac{95}{23}, \frac{77}{19}, \frac{1}{17}, \frac{11}{13}, \frac{13}{11}, \frac{15}{14}, \frac{15}{2}, \frac{55}{1}$$

We start with $n_0 = 2$. The leftmost fraction which yields an integer product is $\frac{15}{2}$, and so $n_1 = 2 \cdot \frac{15}{2} = 15$. Then we get $n_2 = 15 \cdot \frac{55}{1} = 825$, etcetera. By successive application of the algorithm, we obtain the following infinite sequence:

$$2, 15, 825, 725, 1925, 2275, 425, 390, 330, 290, 770, \ldots$$

Apart from 2^1, the powers of 2 occurring in this infinite sequence are $2^2, 2^3, 2^5, 2^7, 2^{11}, 2^{13}, 2^{17}, 2^{19}, \ldots$, where the exponents form the sequence of primes.

We translate Fractran programs to streams specifications in such a way that the specification is productive if and only if the program halts on all $n_0 > 1$. Let us define the target format of this translation: the *lazy stream format* (LSF). LSF consists of stream specifications of the form $\mathsf{M} \to C[\mathsf{M}]$ where C is a context built solely from: one data element $\bullet$, the stream constructor ':', the functions $\mathsf{head}(x : \sigma) \to x$ and $\mathsf{tail}(x : \sigma) \to \sigma$, unary stream functions mod_n, and k-ary stream functions zip_k with the following defining rules, for every $n, k \geq 1$:

$$\begin{aligned} \mathsf{mod}_n(\sigma) &\to \mathsf{head}(\sigma) : \mathsf{mod}_n(\mathsf{tail}^n(\sigma)) \\ \mathsf{zip}_k(\sigma_1, \sigma_2 \ldots, \sigma_k) &\to \mathsf{head}(\sigma_1) : \mathsf{zip}_k(\sigma_2, \ldots, \sigma_k, \mathsf{tail}(\sigma_1)) \end{aligned} \tag{LSF}$$

By reducing the uniform halting problem of Fractran programs to productivity of LSF, we get that productivity for LSF is Π^0_2-complete.

This undecidability result stands in sharp contrast to the decidability of productivity for the *pure stream format* (PSF, [6]). Let us elaborate on the difference between these two formats. Examples of specifications in PSF are:

$$\mathsf{J} \to 0 : 1 : \mathsf{even}(\mathsf{J}) \qquad \text{and} \qquad \mathsf{Z} \to 0 : \mathsf{zip}(\mathsf{even}(\mathsf{Z}), \mathsf{odd}(\mathsf{Z}))\,,$$

including the defining rules for the stream functions involved:

$$\mathsf{even}(x : \sigma) \to x : \mathsf{odd}(\sigma)\,, \quad \mathsf{odd}(x : \sigma) \to \mathsf{even}(\sigma)\,, \quad \mathsf{zip}(x : \sigma, \tau) \to x : \mathsf{zip}(\tau, \sigma)\,,$$

where zip 'zips' two streams alternatingly into one, and even (odd) returns a stream consisting of the elements at its even (odd) positions. The specification for Z produces the stream $0 : 0 : 0 : \ldots$ of zeros, whereas the infinite normal form of J is $0 : 1 : 0 : 0 : \mathsf{even}^\omega$, which is not a constructor normal form.

Excluded from PSF is the observation function on streams $\mathsf{head}(x : \sigma) \to x$. This is for a good reason, as we shall see shortly. PSF is essentially layered: data terms (terms of sort `data`) cannot be built using stream terms (terms of sort `stream`). As soon as *stream dependent* data functions are admitted, the complexity of the productivity problem of such an extended format is increased. Indeed, as our Fractran translation shows, productivity of even the most simple stream specifications is undecidable and Π^0_2-hard. The problem with stream dependent data functions is that they possibly create 'look-ahead': the evaluation of the 'current' stream element may depend on the evaluation of 'future' stream elements. To see this, consider an example from [15]:

$$\mathsf{S}_n \to 0 : \mathsf{S}_n(n) : \mathsf{S}_n$$

where for a term t of sort stream and $n \in \mathbb{N}$, we write $t(n)$ as a shorthand for $\mathsf{head}(\mathsf{tail}^n(t))$. If we take n to be an even number, then S_n is productive, whereas it is unproductive for odd n.

A hint for the fact that it is Π^0_2-hard to decide whether a lazy specification is productive already comes from a simple encoding of the Collatz conjecture (also known as the '$3x$+1-problem' [12]) into a productivity problem. Without proof we state: *the Collatz conjecture is true if and if only the following specification produces the infinite chain* $\bullet : \bullet : \bullet : \ldots$ *of data elements* $\bullet$*:*

$$\mathsf{collatz} \to \bullet : \mathsf{zip}_2(\mathsf{collatz}, \mathsf{mod}_6(\mathsf{tail}^9(\mathsf{collatz}))) \tag{1}$$

In order to understand the operational difference between rules in PSF and rules in LSF, consider the following two rules:

$$\mathsf{read}(\sigma) \to \mathsf{head}(\sigma) : \mathsf{read}(\mathsf{tail}(\sigma)) \tag{2}$$

$$\mathsf{read}'(x : \sigma) \to x : \mathsf{read}'(\sigma) \tag{3}$$

The functions defined by these rules are extensionally equivalent: they both implement the identity function on fully developed streams. However, intensionally, or operationally, there is a difference. A term $\mathsf{read}'(s)$ is a redex only in case s is of the form $u : t$, whereas $\mathsf{read}(s)$ constitutes a redex for *every* stream term s, and so $\mathsf{head}(s)$ can be undefined. The 'lazy' rule (2) *postpones* pattern matching. Although in PSF we can define functions mod'_n and zip'_k extensionally equivalent to mod_n and zip_k, a pure version $\mathsf{collatz}'$ of $\mathsf{collatz}$ in (1) above (using mod'_6 and zip'_2 instead) can easily be seen to be not productive (it produces two data elements only), and to have no bearing on the Collatz conjecture.

Contribution and Overview. In Section 2 we show that the uniform halting problem of Fractran programs is Π^0_2-complete. This is the problem of determining whether a program terminates on all positive integers. Turing-completeness of a computational model does not imply that the uniform halting problem in the strong sense of termination on *all configurations* is Π^0_2-complete. For example, assume that we extend Turing machines with a special non-terminating state. Then the computational model obtained can still compute every recursive function. However, the uniform halting problem becomes trivial.

Our result is a strengthening of the result in [11] where it has been shown that the generalised Collatz problem (GCP) is Π^0_2-complete. This is because every Fractran program P can easily be translated into a Collatz function f such that the uniform halting problem for P is equivalent to the GCP for f. The other direction is not immediate, since Fractran programs form a strict subset of Collatz functions. We discuss this in more detail in Section 2.

In Section 3 we explore alternative definitions of productivity and make them precise in the framework of term rewriting. These can be highly undecidable: 'strong productivity' turns out to be Π^1_1-complete and 'weak productivity' is Σ^1_1-complete. Productivity of individual terms with respect to a computable strategy, which is the notion used in functional programming, is Π^0_2-complete.

In Section 4 we prove that productivity Π^0_2-complete even for specifications of the restricted LSF format. The new proof uses a simple encoding of Fractran programs P into stream specifications of the form $\mathsf{M}_P \to C[\mathsf{M}_P]$, in such a way that M_P is productive if and only if the program P halts on all inputs. The resulting stream specifications are very simple compared to the ones resulting from encoding of Turing machines employed in Section 3. Whereas the Turing machine encoding essentially uses calculations on the elements of the list, the specifications obtained from the Fractran encoding contain no operations on the list elements. In particular, the domain of data elements is a singleton.

Related Work. In [3] undecidability of different properties of first-order TRSs is analysed. While the standard properties of TRSs turn out to be either Σ^0_1- or Π^0_2-complete, the complexity of the dependency pair problems [1] is essentially analytical: it is shown to be Π^1_1-complete. We employ the latter result as a basis for our Π^1_1- and Σ^1_1-completeness results for productivity.

Roşu [14] shows that equality of stream specifications is Π^0_2-complete. We remark that this result can be obtained as a corollary of our translation of Fractran programs P to stream specifications M_P. Stream specifications M_P have the stream $\bullet : \bullet : \ldots$ as unique solutions if and only if they are productive. Thus Π^0_2-completeness of productivity of these specifications implies Π^0_2-completeness of the stream equality problem $\mathsf{M}_P = \bullet : \bullet : \ldots$.

One of the reviewers pointed us to recent work [7] of Grue Simonsen (not available at the time of writing) where Π^0_2-completeness of productivity of orthogonal stream specifications is shown. Theorem 3.5 below can be seen as a sharpening of that result in that we consider general TRSs and productivity with respect to arbitrary evaluation strategies. For orthogonal systems the evaluation strategy is irrelevant as long as it is outermost-fair. Moreover we further strengthen the result on orthogonal stream specifications by restricting the format to LSF.

2 Fractran

The one step computation of a Fractran program is a partial function.

Definition 2.1. Let $P = \frac{p_1}{q_1}, \ldots, \frac{p_k}{q_k}$ be a Fractran program. The partial function $f_P : \mathbb{N} \rightharpoonup \mathbb{N}$ is defined for all $n \in \mathbb{N}$ by:

$$f_P(n) = \begin{cases} n \cdot \frac{p_i}{q_i} & \text{where } \frac{p_i}{q_i} \text{ is the first fraction of } P \text{ such that } n \cdot \frac{p_i}{q_i} \in \mathbb{N}, \\ \text{undefined} & \text{if no such fraction exists.} \end{cases}$$

We say that P *halts* on $n \in \mathbb{N}$ if there exists $i \in \mathbb{N}$ such that $f^i_P(n) = \text{undefined}$. For $n, m \in \mathbb{N}$ we write $n \to_P m$ whenever $m = f_P(n)$.

The Fractran program for generating prime numbers, that we discussed in the introduction, is non-terminating for all starting values n_0, because the product of any integer with $\frac{55}{1}$ is an integer again. However, in general, termination of Fractran programs is undecidable.

Theorem 2.2. *The uniform halting problem for Fractran programs, that is, deciding whether a program halts for every starting value $n_0 \in \mathbb{N}_{>0}$, is Π^0_2-complete.*

A related result is obtained in [11] where it is shown that the generalised Collatz problem (GCP) is Π^0_2-complete, that is, the problem of deciding for a Collatz function f whether for every integer $x > 0$ there exists $i \in \mathbb{N}$ such that $f^i(x) = 1$. A Collatz function f is a function $f : \mathbb{N} \to \mathbb{N}$ of the form:

$$f(n) = \begin{cases} a_0 \cdot n + b_0, & \text{if } n \equiv 0 \pmod{p} \\ \vdots & \vdots \\ a_{p-1} \cdot n + b_{p-1}, & \text{if } n \equiv p-1 \pmod{p} \end{cases}$$

for some $p \in \mathbb{N}$ and rational numbers a_i, b_i such that $f(n) \in \mathbb{N}$ for all $n \in \mathbb{N}$.

The result of [11] is an immediate corollary of Theorem 2.2. Every Fractran program P is a Collatz function f'_P where f'_P is obtained from f_P (see Definition 2.1) by replacing undefined with 1. We obtain the above representation of Collatz functions simply by choosing for p the least common multiple of the denominators of the fractions of P. We call a Fractran program P *trivially immortal* if P contains a fraction with denominator 1 (an integer). Then for all not trivially immortal P, P halts on all inputs if and only for all $x > 0$ there exists $i \in \mathbb{N}$ such that $f^i_P(x) = 1$. Using our result, this implies that GCP is Π^0_2-hard.

Theorem 2.2 is a strengthening of the result in [11] since Fractran programs are a strict subset of Collatz functions. If Fractran programs are represented as Collatz functions directly, for all $0 \leq i < p$ it holds either $b_i = 0$, or $a_i = 0$ and $b_i = 1$. Via such a translation Fractran programs are, e.g., not able to implement the famous Collatz function $C(2n) = n$ and $C(2n+1) = 6n+4$ (for all $n \in \mathbb{N}$), nor an easy function like $f(2n) = 2n+1$ and $f(2n+1) = 2n$ (for all $n \in \mathbb{N}$).

For the proof of Theorem 2.2 we devise a translation from Turing machines to Fractran programs ([11] uses register machines) such that the resulting Fractran program halts on all positive integers ($n_0 \geq 1$) if and only if the Turing machine is terminating on all configurations. That is, we reduce the uniform halting problem of Turing machines to the uniform halting problem of Fractran programs.

We briefly explain why we employ the uniform halting problem instead of the problem of totality (termination on all inputs) of Turing machines, also known as the initialised uniform halting problem. When translating a Turing machine M to a Fractran program P_M, start configurations (initialised configurations) are mapped to a subset $I_M \subseteq \mathbb{N}$ of Fractran inputs. Then from Π^0_2-hardness of the totality problem one can conclude Π^0_2-hardness of the question whether P_M terminates on all numbers from I_M. But this does not imply that the uniform halting problem for Fractran programs is Π^0_2-hard (termination on all natural numbers $n \in \mathbb{N}$). The numbers not in the target of the translation could make the problem both harder as well as easier. A situation where extending the domain of inputs makes the problem easier is: local confluence of TRSs is Π^0_2-complete for the set of ground terms, but only Σ^0_1-complete for the set of all terms [3].

To keep the translation as simple we restrict to unary Turing machines having only two symbols $\{0, 1\}$ in their tape alphabet, 0 being the blank symbol.

Definition 2.3. A unary *Turing machine* M is a triple $\langle Q, q_0, \delta\rangle$, where Q is a finite set of states, $q_0 \in Q$ the initial state, and $\delta : Q \times \{0,1\} \rightharpoonup Q \times \{0,1\} \times \{L, R\}$ a (partial) *transition function*. A *configuration* of M is a pair $\langle q, \mathit{tape}\rangle$ consisting of a state $q \in Q$ and the tape content $\mathit{tape} : \mathbb{Z} \to \{0,1\}$ such that the support $\{n \in \mathbb{Z} \mid \mathit{tape}(n) \neq 0\}$ is finite. The set of all configurations is denoted by $\mathcal{C}\mathit{onf}_M$. We define the relation $\to_M$ on the set of configurations $\mathcal{C}\mathit{onf}_M$ as follows: $\langle q, \mathit{tape}\rangle \to_M \langle q', \mathit{tape}'\rangle$ whenever:

- $\delta(q, \mathit{tape}(0)) = \langle q', f, L\rangle$, $\mathit{tape}'(1) = f$ and $\forall n \neq 0.\ \mathit{tape}'(n+1) = \mathit{tape}(n)$, or
- $\delta(q, \mathit{tape}(0)) = \langle q', f, R\rangle$, $\mathit{tape}'(-1) = f$ and $\forall n \neq 0.\ \mathit{tape}'(n-1) = \mathit{tape}(n)$.

We say that M *halts (or terminates) on a configuration* $\langle q, \mathit{tape}\rangle$ if the configuration $\langle q, \mathit{tape}\rangle$ does not admit infinite $\to_M$ rewrite sequences.

The *uniform halting problem* of Turing machines is the problem of deciding whether a given Turing machine M halts on all (initial or intermediate) configurations. The following theorem is a result of [8]:

Theorem 2.4. *The uniform halting problem for Turing machines is* Π_2^0*-complete.*

This result carries over to unary Turing machines using a simulation based on a straightforward encoding of tape symbols as blocks of zeros and ones (of equal length), which are admissible configurations of unary Turing machines.

We now give a translation of Turing machines to Fractran programs. Without loss of generality we restrict in the sequel to Turing machines $M = \langle Q, q_0, \delta\rangle$ for which $\delta(q,x) = \langle q', s', d'\rangle$ implies $q \neq q'$. In case M does not fulfil this condition then we can find an equivalent Turing machine $M' = \langle Q \cup Q_\#, q_0, \delta'\rangle$ where $Q_\# = \{q_\# \mid q \in Q\}$ and δ' is defined by $\delta'(q,x) = \langle p_\#, s, d\rangle$ and $\delta'(q_\#, x) = \langle p, s, d\rangle$ for $\delta(q,x) = \langle p, s, d\rangle$.

Definition 2.5. Let $M = \langle Q, q_0, \delta\rangle$ be a Turing machine. Let $tape_\ell$, h, $tape_r$, $tape'_\ell$, h', $tape'_r$, $m_{L,x}$, $m_{R,x}$, $copy_x$ and p_q for every $q \in Q$ and $x \in \{0,1\}$ be pairwise distinct prime numbers. The intuition behind these primes is:

- $tape_\ell$ and $tape_r$ represent the tape left and right of the head, respectively,
- h is the tape symbol in the cell currently scanned by the tape head,
- $tape'_\ell$, h', $tape'_r$ store temporary tape content (when moving the head),
- $m_{L,x}$, $m_{R,x}$ execute a left or right move of the head on the tape, respectively,
- $copy_x$ copies the temporary tape content back to the primary tape, and
- p_q represent the states of the Turing machine.

The subscript $x \in \{0,1\}$ is used to have two primes for every action: in case an action p takes more than one calculation step we cannot write $\frac{p\cdot\ldots}{p\cdot\ldots}$ since then p in numerator and denominator would cancel itself out. We define the Fractran program P_M to consist of the following fractions (listed in program order):

$$\frac{1}{p\cdot p'} \qquad \begin{array}{c}\text{for every } p,p' \in \{m_{L,0},\, m_{L,1},\, m_{R,0},\, m_{R,1},\, copy_0,\, copy_1\}\\ \text{every } p,p' \in \{p_q \mid q \in Q\} \text{ and } p,p' \in \{h, h'\}\end{array} \tag{4}$$

to get rid of illegal configurations,

$$\frac{m_{L,1-x}\cdot tape'_\ell}{m_{L,x}\cdot tape_\ell^2} \quad \frac{m_{L,1-x}\cdot tape_r'^2}{m_{L,x}\cdot tape_r} \quad \frac{m_{L,1-x}\cdot tape'_r}{m_{L,x}\cdot h'} \quad \frac{m_{L,1-x}\cdot h}{m_{L,x}\cdot tape_\ell} \quad \frac{copy_0}{m_{L,x}} \tag{5}$$

with $x \in \{0,1\}$, for moving the head left on the tape,

$$\frac{m_{R,1-x}\cdot tape'_r}{m_{R,x}\cdot tape_r^2} \quad \frac{m_{R,1-x}\cdot tape_\ell'^2}{m_{R,x}\cdot tape_\ell} \quad \frac{m_{R,1-x}\cdot tape'_\ell}{m_{R,x}\cdot h'} \quad \frac{m_{R,1-x}\cdot h}{m_{R,x}\cdot tape_r} \quad \frac{copy_0}{m_{R,x}} \tag{6}$$

with $x \in \{0,1\}$, for moving the head right on the tape,

$$\frac{copy_{1-x}\cdot tape_\ell}{copy_x\cdot tape'_\ell} \qquad \frac{copy_{1-x}\cdot tape_r}{copy_x\cdot tape'_r} \qquad \frac{1}{copy_x} \tag{7}$$

with $x \in \{0,1\}$, for copying the temporary tape back to the primary tape,

$$\frac{p_{q'} \cdot h'^{s'} \cdot m_{d,0}}{p_q \cdot h} \qquad \text{whenever } \delta(q,1) = \langle q', s', d\rangle \tag{8}$$

$$\frac{1}{p_q \cdot h} \qquad \text{(for termination) for every } q \in Q \tag{9}$$

$$\frac{p_{q'} \cdot h'^{s'} \cdot m_{d,0}}{p_q} \qquad \text{whenever } \delta(q,0) = \langle q', s', d\rangle \tag{10}$$

for the transitions of the Turing machine. Whenever we use variables in the rules, e.g. $x \in \{0,1\}$, then it is to be understood that instances of the same rule are immediate successors in the sequence of fractions (the order of the instances among each other is not crucial).

Example 2.6. Let $M = \langle Q, a_0, \delta\rangle$ be a Turing machine where $Q = \{a_0, a_1, b\}$, and the transition function is defined by $\delta(a_0,0) = \langle b,1,R\rangle$, $\delta(a_1,0) = \langle b,1,R\rangle$, $\delta(a_0,1) = \langle a_1,0,R\rangle$, $\delta(a_1,1) = \langle a_0,0,R\rangle$, $\delta(b,1) = \langle a_0,0,R\rangle$, and we leave $\delta(b,0)$ undefined. That is, M moves to the right, converting zeros into ones and vice versa, until it finds two consecutive zeros and terminates. Assume that M is started on the configuration $1b1001$, that is, the tape content 11001 in state b with the head located on the second 1. In the Fractran program P_M this corresponds to $n_0 = p_b \cdot tape_\ell^1 \cdot h^1 \cdot tape_r^{100}$ as the start value where we represent the exponents in binary notation for better readability. Started on n_0 we obtain the following calculation in P_M:

$$\begin{aligned}
&p_b \cdot tape_\ell^1 \cdot h^1 \cdot tape_r^{100} \qquad \text{(configuration } 1b1001\text{)}\\
\to_{(8)}\ & m_{R,0} \cdot p_{a_0} \cdot tape_\ell^1 \cdot tape_r^{100} \to^2_{(6;1^{\text{st}})} m_{R,0} \cdot p_{a_0} \cdot tape_\ell^1 \cdot tape_r'^{10}\\
\to_{(6;2^{\text{nd}})}\ & m_{R,1} \cdot p_{a_0} \cdot tape_\ell'^{10} \cdot tape_r'^{10} \to_{(6;5^{\text{th}})} copy_0 \cdot p_{a_0} \cdot tape_\ell'^{10} \cdot tape_r'^{10}\\
\to^2_{(7;1^{\text{st}})} \to^2_{(7;2^{\text{nd}})} \to_{(7;3^{\text{rd}})}\ & p_{a_0} \cdot tape_\ell^{10} \cdot tape_r^{10} \qquad \text{(configuration } 10a001\text{)}\\
\to_{(10)}\ & m_{R,0} \cdot p_b \cdot tape_\ell^{10} \cdot h'^1 \cdot tape_r^{10} \to_{(6;1^{\text{st}})} m_{R,1} \cdot p_b \cdot tape_\ell^{10} \cdot h'^1 \cdot tape_r'^1\\
\to^2_{(6;2^{\text{nd}})}\ & m_{R,1} \cdot p_b \cdot tape_\ell'^{100} \cdot h'^1 \cdot tape_r'^1 \to_{(6;3^{\text{rd}}+5^{\text{th}})} copy_0 \cdot p_b \cdot tape_\ell'^{101} \cdot tape_r'^1\\
\to^5_{(7;1^{\text{st}})} \to_{(7;2^{\text{nd}})} \to_{(7;3^{\text{rd}})}\ & p_b \cdot tape_\ell^{101} \cdot tape_r^1 \qquad \text{(configuration } 101b01\text{)}
\end{aligned}$$

reaching a configuration where the Fractran program halts.

Definition 2.7. We translate configurations $c = \langle q, tape\rangle$ of Turing machines $M = \langle Q, q_0, \delta\rangle$ to natural numbers (input values for Fractran programs). We reuse the notation of Definition 2.5 and define:

$$n_c = tape_\ell^L \cdot p_q \cdot h^H \cdot tape_r^R$$

$$L = \sum_{i=0}^{\infty} 2^i \cdot tape(-1-i) \qquad H = tape(0) \qquad R = \sum_{i=0}^{\infty} 2^i \cdot tape(1+i)$$

Lemma 2.8. *For every Turing machine M and configurations c_1, c_2 we have:*

(i) if $c_1 \to_M c_2$ then $n_{c_1} \to^*_{P_M} n_{c_2}$, and
(ii) if c_1 is a $\to_M$ normal form then $n_{c_1} \to^*_{P_M}$ undefined.

Proofs of Lemma 2.8 and Theorem 2.2 can be found in [5].

3 What Is Productivity?

A program is productive if it evaluates to a finite or infinite constructor normal form. This rather vague description leaves open several choices that can be made to obtain a more formal definition. We explore several definitions and determine the degree of undecidability for each of them. See [6] for more pointers to the literature on productivity.

The following is a productive specification of the (infinite) stream of zeros:

$$\mathsf{zeros} \to 0 : \mathsf{zeros}$$

Indeed, there exists only one maximal rewrite sequence from zeros and this ends in the infinite constructor normal form $0 : 0 : 0 : \ldots$. Here and later we say that a rewrite sequence $\rho : t_0 \to t_1 \to t_2 \to \ldots$ *ends in* a term s if either ρ is finite with its last term being s, or ρ is infinite and then s is the limit of the sequence of terms t_i, i.e. $s = \lim_{i\to\infty} t_i$. We consider only rewrite sequences starting from finite terms, thus all terms occurring in ρ are finite. Nevertheless, the limit s of the terms t_i may be an infinite term. Note that, if ρ ends in a constructor normal form, then every finite prefix will be evaluated after finitely many steps.

The following is a slightly modified specification of the stream of zeros:

$$\mathsf{zeros} \to 0 : \mathsf{id}(\mathsf{zeros}) \qquad\qquad \mathsf{id}(\sigma) \to \sigma$$

This specification is considered productive as well, although there are infinite rewrite sequences that do not even end in a normal form, let alone in a constructor normal form: e.g. by unfolding zeros only we get the limit term $0 : \mathsf{id}(0 : \mathsf{id}(0 : \mathsf{id}(\ldots)))$. In general, normal forms can only be reached by outermost-fair rewriting sequences. A rewrite sequence $\rho : t_0 \to t_1 \to t_2 \to \ldots$ is *outermost-fair* [16] if there is no t_n containing an outermost redex which remains an outermost redex infinitely long, and which is never contracted. For this reason it is natural to consider productivity of terms with respect to outermost-fair strategies.

What about stream specifications that admit rewrite sequences to constructor normal forms, but that also have divergent rewrite sequences:

$$\mathsf{maybe} \to 0 : \mathsf{maybe} \qquad \mathsf{maybe} \to \mathsf{sink} \qquad \mathsf{sink} \to \mathsf{sink}$$

This example illustrates that, for non-orthogonal stream specifications, reachability of a constructor normal form depends on the evaluation strategy. The term maybe is only productive with respect to strategies that always apply the first rule.

For this reason we propose to think of productivity as a property of individual terms *with respect to* a given rewrite strategy. This reflects the situation in functional programming, where expressions are evaluated according to an inbuilt strategy. These strategies are usually based on a form of outermost-needed rewriting with a priority order on the rules.

3.1 Productivity with Respect to Strategies

For term rewriting systems (TRSs) [16] we now fix definitions of the notions of (history-free) strategy and history-aware strategy. Examples for the latter notion are outermost-fair strategies, which typically have to take history into account.

Definition 3.1. Let R be a TRS with rewrite relation $\to_R$.

A *strategy for* $\to_R$ is a relation $\rightsquigarrow \subseteq \to_R$ with the same normal forms as $\to_R$.

The *history-aware rewrite relation* $\to_{\mathcal{H},R}$ *for* R is the binary relation on $\mathit{Ter}(\Sigma) \times (R \times \mathbb{N}^*)^*$ that is defined by:

$$\langle s, h_s\rangle \to_{\mathcal{H},R} \langle t, h_s : \langle \rho, p\rangle\rangle \iff s \to t \text{ via rule } \rho \in R \text{ at position } p\,.$$

We identify $t \in \mathit{Ter}(\Sigma)$ with $\langle t, \epsilon\rangle$, and for $s, t \in \mathit{Ter}(\Sigma)$ we write $s \to_{\mathcal{H},R} t$ whenever $\langle s, \epsilon\rangle \to_{\mathcal{H},R} \langle t, h\rangle$ for some history $h \in (R \times \mathbb{N}^*)^*$. A *history-aware strategy for* R is a strategy for $\to_{\mathcal{H},R}$.

A strategy $\rightsquigarrow$ is *deterministic* if $s \rightsquigarrow t$ and $s \rightsquigarrow t'$ implies $t = t'$. A strategy $\rightsquigarrow$ is *computable* if the function mapping a term (a term/history pair) to its set of $\rightsquigarrow$-successors is a total recursive function, after coding into natural numbers.

Remark 3.2. Our definition of strategy for a rewrite relation follows [17]. For abstract rewriting systems, in which rewrite steps are first-class citizens, a definition of strategy is given in [16, Ch. 9]. There, history-aware strategies for a TRS R are defined in terms of 'labellings' for the 'abstract rewriting system' underlying R. While that approach is conceptually advantageous, our definition of history-aware strategy is equally expressive.

Definition 3.3. A *(TRS-indexed) family of strategies* $\mathcal{S}$ is a function that assigns to every TRS R a set $\mathcal{S}(R)$ of strategies for R. We call such a family $\mathcal{S}$ of strategies *admissible* if $\mathcal{S}(R)$ is non-empty for every orthogonal TRS R.

Now we give the definition of productivity with respect to a strategy.

Definition 3.4. A term t is called *productive with respect to a strategy* $\rightsquigarrow$ if all maximal $\rightsquigarrow$ rewrite sequences starting from t end in a constructor normal form.

In the case of non-deterministic strategies we require here that all maximal rewrite sequences end in a constructor normal form. Another possible choice could be to require only the existence of one such rewrite sequence (see Section 3.2). However, we think that productivity should be a practical notion. Productivity of a term should entail that arbitrary finite parts of the constructor normal form can indeed be evaluated. The mere requirement that a constructor

normal form exists leaves open the possibility that such a normal form cannot be approximated to every finite precision in a computable way.

For orthogonal TRSs outermost-fair (or fair) rewrite strategies are the natural choice for investigating productivity because they guarantee to find (the unique) infinitary constructor normal form whenever it exists (see [16]).

Pairs and finite lists of natural numbers can be encoded using the well-known Gödel encoding. Likewise terms and finite TRSs over a countable set of variables can be encoded. A TRS is called *finite* if its signature and set of rules are finite. In the sequel we restrict to (families of) computable strategies, and assume that strategies are represented by appropriate encodings.

Now we define the productivity problem in TRSs with respect to families of computable strategies, and prove a Π^0_2-completeness result.

PRODUCTIVITY PROBLEM with respect to a family $\mathcal{S}$ of computable strategies.

Instance: Encodings of a finite TRS R, a strategy $\rightsquigarrow \in \mathcal{S}(R)$ and a term t.
Answer: 'Yes' if t is productive with respect to $\rightsquigarrow$, and 'No', otherwise.

Theorem 3.5. *For every family of admissible, computable strategies $\mathcal{S}$, the productivity problem with respect to $\mathcal{S}$ is Π^0_2-complete.*

Proof. A Turing machine is called *total* (encodes a total function $\mathbb{N} \to \mathbb{N}$) if it halts on all inputs encoding natural numbers. The problem of deciding whether a Turing machine is *total* is well-known to be Π^0_2-complete, see [9]. Let M be an arbitrary Turing machine. Employing the encoding of Turing machines into orthogonal TRSs from [10], we can define a TRS R_M that simulates M such that for every $n \in \mathbb{N}$ it holds: every reduct of the term $\mathsf{M}(\mathsf{s}^n(0))$ contains at most one redex occurrence, and the term $\mathsf{M}(\mathsf{s}^n(0))$ rewrites to 0 if and only if the Turing machine M halts on the input n. Note that the rewrite sequence starting from $\mathsf{M}(\mathsf{s}^n(0))$ is deterministic. We extend the TRS R_M to a TRS R'_M with the following rules:

$$\mathsf{go}(0, x) \to 0 : \mathsf{go}(\mathsf{M}(x), \mathsf{s}(x))$$

and choose the term $t = \mathsf{go}(0, 0)$. Then R'_M is orthogonal and by construction every reduct of t contains at most one redex occurrence (consequently all strategies for R coincide on every reduct of t). The term t is productive if and only if $\mathsf{M}(\mathsf{s}^n(0))$ rewrites to 0 for every $n \in \mathbb{N}$ which in turn holds if and only if the Turing machine M is total. This concludes Π^0_2-hardness.

For Π^0_2-completeness let $\mathcal{S}$ be a family of computable strategies, R a TRS, $\rightsquigarrow \in \mathcal{S}(R)$ and t a term. Then productivity of t can be characterised as:

$$\forall d \in \mathbb{N}.\, \exists n \in \mathbb{N}.\, \text{every } n\text{-step } \rightsquigarrow\text{-reducts of } t \text{ is a constructor normal form up to depth } d \qquad (\star)$$

Since the strategy $\rightsquigarrow$ is computable and finitely branching, all n-step reducts of t can be computed. Obviously, if the formula $(\star)$ holds, then t is productive w.r.t. $\rightsquigarrow$. Conversely, assume that t is productive w.r.t. $\rightsquigarrow$. For showing $(\star)$, let $d \in \mathbb{N}$

be arbitrary. By productivity of t w.r.t. $\rightsquigarrow$, on every path in the reduction graph of t w.r.t. $\rightsquigarrow$ eventually a term with a constructor normal form up to depth d is encountered. Since reduction graphs in TRSs always are finitely branching, Koenig's lemma implies that there exists an $n \in \mathbb{N}$ such that all terms on depth greater or equal to n in the reduction graph of t are constructor prefixes of depth at least d. Since d was arbitrary, $(\star)$ has been established. Because $(\star)$ is a Π^0_2-formula, the productivity problem with respect to $\mathcal{S}$ also belongs to Π^0_2. □

Theorem 3.5 implies that productivity is Π^0_2-complete for orthogonal TRSs with respect to outermost-fair rewriting. To see this, apply the theorem to the family of strategies that assigns to every orthogonal TRS R the set of computable, outermost-fair rewriting strategies for R, and $\varnothing$ to non-orthogonal TRSs.

The definition of productivity with respect to computable strategies reflects the situation in functional programming. Nevertheless, we now investigate variants of this notion, and determine their respective computational complexity.

3.2 Strong Productivity

As already discussed, only outermost-fair rewrite sequences can reach a constructor normal form. Dropping the fine tuning device 'strategies', we obtain the following stricter notion of productivity.

Definition 3.6. A term t is called *strongly productive* if all maximal outermost-fair rewrite sequences starting from t end in a constructor normal form.

The definition requires all outermost-fair rewrite sequences to end in a constructor normal form, including non-computable rewrite sequences. This catapults productivity into a much higher class of undecidability: Π^1_1, a class of the analytical hierarchy. The analytical hierarchy continues the classification of the arithmetical hierarchy using second order formulas. The computational complexity of strong productivity therefore exceeds the expressive power of first-order logic to define sets from recursive sets.

A well-known result of recursion theory states that for a given computable relation $> \subseteq \mathbb{N} \times \mathbb{N}$ it is Π^1_1-hard to decide whether $>$ is well-founded, see [9]. Our proof is based on a construction from [3]. There a translation from Turing machines M to TRSs $\mathcal{R}oot_M$ (which we explain below) together with a term t_M is given such that: t_M is root-terminating (i.e., t_M admits no rewrite sequences containing an infinite number of root steps) if and only if the binary relation $>_M$ encoded by M is well-founded. The TRS $\mathcal{R}oot_M$ consists of rules for simulating the Turing machine M such that $M(x, y) \to^* \mathsf{T}$ iff $x >_M y$ holds (which basically uses a standard encoding of Turing machines, see [10]), a rule:

$$\mathsf{run}(\mathsf{T}, \mathsf{ok}(x), \mathsf{ok}(y)) \to \mathsf{run}(M(x, y), \mathsf{ok}(y), \mathsf{pickn})$$

and rules for randomly generating a natural number:

$$\mathsf{pickn} \to \mathsf{c}(\mathsf{pickn}) \qquad \mathsf{pickn} \to \mathsf{ok}(0(\triangleright)) \qquad \mathsf{c}(\mathsf{ok}(x)) \to \mathsf{ok}(\mathsf{S}(x))\,.$$

The term $t_M = \mathsf{run}(\mathsf{T}, \mathsf{pickn}, \mathsf{pickn})$ admits a rewrite sequence containing infinitely many root steps if and only if $>_M$ is not well-founded. More precisely, whenever there is an infinite decreasing sequence $x_1 >_M x_2 >_M x_3 >_M \ldots$, then t_M admits a rewrite sequence $\mathsf{run}(\mathsf{T}, \mathsf{pickn}, \mathsf{pickn}) \to^* \mathsf{run}(\mathsf{T}, \mathsf{ok}(x_1), \mathsf{ok}(x_2)) \to \mathsf{run}(M(x_1, x_2), \mathsf{ok}(x_2), \mathsf{pickn}) \to^* \mathsf{run}(\mathsf{T}, \mathsf{ok}(x_2), \mathsf{ok}(x_3)) \to^* \ldots$. We further note that t_M and all of its reducts contain exactly one occurrence of the symbol run, namely at the root position.

Theorem 3.7. *Strong productivity is* Π^1_1*-complete.*

Proof. For the proof of Π^1_1-hardness, let M be a Turing machine. We extend the TRS $\mathcal{R}oot_M$ from [3] with the rule $\mathsf{run}(x, y, z) \to 0{:}\mathsf{run}(x, y, z)$. As a consequence the term $\mathsf{run}(\mathsf{T}, \mathsf{pickn}, \mathsf{pickn})$ is strongly productive if and only if $>_M$ is well-founded (which is Π^1_1-hard to decide). If $>_M$ is not well-founded, then by the result in [3] t_M admits a rewrite sequence containing infinitely many root steps which obviously does not end in a constructor normal form. On the other hand if $>_M$ is well-founded, then t_M admits only finitely many root steps with respect to $\mathcal{R}oot_M$, and thus by outermost-fairness the freshly added rule has to be applied infinitely often. This concludes Π^1_1-hardness.

Rewrite sequences of length ω can be represented by functions $r : \mathbb{N} \to \mathbb{N}$ where $r(n)$ represents the n-th term of the sequence together with the position and rule applied in step n. Then for all r (one universal $\forall r$ function quantifier) we have to check that r converges towards a constructor normal form whenever r is outermost-fair; this can be checked by a first order formula. We refer to [3] for the details of the encoding. Hence strong productivity is in Π^1_1. □

3.3 Weak Productivity

A natural counterpart to strong productivity is the notion of 'weak productivity': the existence of a rewrite sequence to a constructor normal form. Here outermost-fairness does not need to be required, because rewrite sequences that reach normal forms are always outermost-fair.

Definition 3.8. A term t is called *weakly productive* if there exists a rewrite sequence starting from t that ends in a constructor normal form.

For non-orthogonal TRSs the practical relevance of this definition is questionable since, in the absence of a computable strategy to reach normal forms, mere knowledge that a term t is productive does typically not help to find a constructor normal form of t. For orthogonal TRSs computable, normalising strategies exist, but then also all of the variants of productivity coincide (see Section 3.4).

Theorem 3.9. *Weak productivity is* Σ^1_1*-complete.*

Proof. For the proof of Σ^1_1-hardness, let M be a Turing machine. We exchange the rule $\mathsf{run}(\mathsf{T}, \mathsf{ok}(x), \mathsf{ok}(y)) \to \mathsf{run}(M(x, y), \mathsf{ok}(y), \mathsf{pickn})$ in the TRS $\mathcal{R}oot_M$ from [3] by the rule $\mathsf{run}(\mathsf{T}, \mathsf{ok}(x), \mathsf{ok}(y)) \to 0{:}\mathsf{run}(M(x, y), \mathsf{ok}(y), \mathsf{pickn})$. Then we

obtain that the term $\mathsf{run}(\mathsf{T}, \mathsf{pickn}, \mathsf{pickn})$ is weakly productive if and only if $>_M$ is not well-founded (which is Σ^1_1-hard to decide). This concludes Π^1_1-hardness.

The remainder of the proof proceeds analogously to the proof of Theorem 3.7, except that we now have an existential function quantifier $\exists r$ to quantify over all rewrite sequences of length ω. Hence weak productivity is in Σ^1_1. □

3.4 Discussion

For orthogonal TRSs all of the variants of productivity coincide. That is, if we restrict the first variant to computable outermost-fair strategies; as already discussed, other strategies are not very reasonable. For orthogonal TRSs there always exist computable outermost-fair strategies, and whenever for a term there exists a constructor normal form, then it is unique and all outermost-fair rewrite sequences will end in this unique constructor normal form.

This raises the question whether uniqueness of the constructor normal forms should be part of the definition of productivity. We consider a specification of the stream of random bits:

$$\mathsf{random} \to 0 : \mathsf{random} \qquad\qquad \mathsf{random} \to 1 : \mathsf{random}$$

Every rewrite sequence starting from random ends in a normal form. However, these normal forms are not unique. In fact, there are uncountably many of them. We did not include uniqueness of normal forms into the definition of productivity since non-uniqueness only arises in non-orthogonal TRSs when using non-deterministic strategies. However, one might want to require uniqueness of normal forms even in the case of non-orthogonal TRSs.

Theorem 3.10. *The problem of determining, for TRSs R and terms t in R, whether t has a unique (finite or infinite) normal form is Π^1_1-complete.*

Proof. For Π^1_1-hardness, we extend the TRS constructed in the proof of Theorem 3.9 by the rules: $\mathsf{start} \to \mathsf{run}(\mathsf{T}, \mathsf{pickn}, \mathsf{pickn})$, $\mathsf{run}(x, y, z) \to \mathsf{run}(x, y, z)$, $\mathsf{start} \to \mathsf{ones}$, and $\mathsf{ones} \to 1 : \mathsf{ones}$. Then start has a unique normal form if and only if $>_M$ is well-founded. For Π^1_1-completeness, we observe that the property can be characterised by a Π^1_1-formula: we quantify over two infinite rewrite sequences, and, in case both of them end in a normal form, we compare them. Note that consecutive universal quantifiers can be compressed into one. □

Let us consider the impact on computational complexity of taking up the condition of uniqueness of normal forms into the definition of productivity. Including uniqueness of normal forms without considering the strategy would increase the complexity of productivity with respect to a family of strategies to Π^1_1. However, we think that doing so would be contrary to the spirit of the notion of productivity. Uniqueness of normal forms should only be required for the normal forms reachable by the given (non-deterministic) strategy. But then the complexity of productivity remains unchanged, Π^0_2-complete. The complexity of strong productivity remains unaltered, Π^1_1-complete, when including uniqueness

of normal forms. However, the degree of undecidability of weak productivity increases. From the proofs of Theorems 3.9 and 3.10 it follows that the property would then both be Σ^1_1-hard and Π^1_1-hard, then being in Δ^1_1.

4 Productivity for Lazy Stream Specifications Is Π^0_2

In this section we strengthen the undecidability result of Theorem 3.5 by showing that the productivity problem is Π^0_2-complete already for a very simple format of stream specifications, namely the lazy stream format (LSF) introduced on page 373. We do so by giving a translation from Fractran programs into LSF and applying Theorem 2.2.

Definition 4.1. Let $P = \frac{p_1}{q_1}, \ldots, \frac{p_k}{q_k}$ be a Fractran program. Let d be the least common multiple of the denominators of P, that is, $d := \mathrm{lcm}(q_1, \ldots, q_k)$. Then for $n = 1, \ldots, d$ define $p'_n = p_i \cdot (d/q_i)$ and $b_n = n \cdot \frac{p_i}{q_i}$ where $\frac{p_i}{q_i}$ is the first fraction of P such that $n \cdot \frac{p_i}{q_i}$ is an integer, and we let p'_n and b_n be undefined if no such fraction exists. Then, the *stream specification induced by* P is a term rewriting system $\mathcal{R}_P = \langle \Sigma_P, R_P \rangle$ with:

$$\Sigma_P = \{\bullet, :, \mathsf{head}, \mathsf{tail}, \mathsf{zip}_d, \mathsf{M}_P\} \cup \{\mathsf{mod}_{p'_n} \mid p'_n \text{ is defined}\}$$

and with R_P consisting of the following rules:

$$\mathsf{M}_P \to \mathsf{zip}_d(\mathsf{T}_1, \ldots, \mathsf{T}_d), \text{ where, for } 1 \le n \le d, \mathsf{T}_n \text{ is shorthand for:}$$

$$\mathsf{T}_n = \begin{cases} \mathsf{mod}_{p'_n}(\mathsf{tail}^{b_n - 1}(\mathsf{M}_P)) & \text{if } p'_n \text{ is defined,} \\ \bullet : \mathsf{mod}_d(\mathsf{tail}^{n-1}(\mathsf{M}_P)) & \text{if } p'_n \text{ is undefined.} \end{cases}$$

$$\mathsf{head}(x : \sigma) \to x \qquad \mathsf{mod}_k(\sigma) \to \mathsf{head}(\sigma) : \mathsf{mod}_k(\mathsf{tail}^k(\sigma))$$

$$\mathsf{tail}(x : \sigma) \to \sigma \qquad \mathsf{zip}_d(\sigma_1, \sigma_2 \ldots, \sigma_d) \to \mathsf{head}(\sigma_1) : \mathsf{zip}_d(\sigma_2, \ldots, \sigma_d, \mathsf{tail}(\sigma_1))$$

where x, σ, σ_i are variables.[1]

The rule for mod_n defines a stream function which takes from a given stream σ all elements $\sigma(i)$ with $i \equiv 0 \pmod n$, and results in a stream consisting of those elements in the original order. As we only need rules $\mathsf{mod}_{p'_n}$ whenever p'_n is defined we need d such rules at most.

If p'_n is undefined then it should be understood that $m \cdot p'_n$ is undefined. For $n \in \mathbb{N}$ let $\varphi(n)$ denote the number from $\{1, \ldots, d\}$ with $n \equiv \varphi(n) \pmod d$.

Lemma 4.2. *For every $n > 0$ we have $f_P(n) = \lfloor (n-1)/d \rfloor \cdot p'_{\varphi(n)} + b_{\varphi(n)}$.*

Proof. Let $n > 0$. For every $i \in \{1, \ldots, k\}$ we have $n \cdot \frac{p_i}{q_i} \in \mathbb{N}$ if and only if $\varphi(n) \cdot \frac{p_i}{q_i} \in \mathbb{N}$, since $n \equiv \varphi(n) \bmod d$ and d is a multiple of q_i. Assume that $f_P(n)$

[1] Note that $\mathsf{mod}_d(\mathsf{tail}^{n-1}(\mathsf{zip}_d(\mathsf{T}_1, \ldots, \mathsf{T}_d)))$ equals T_n, and so, in case p'_n is undefined, we just have $\mathsf{T}_n = \bullet : \mathsf{T}_n$. In order to have the simplest TRS possible (for the purpose at hand), we did not want to use an extra symbol ($\bullet$) and rule $(\bullet) \to \bullet : (\bullet)$.

is defined. Then $f_P(n) = n{\cdot}p'_{\varphi(n)}/d = (\lfloor(n{-}1)/d\rfloor{\cdot}d{+}((n{-}1) \bmod d){+}1){\cdot}p'_{\varphi(n)}/d$ $= \lfloor(n{-}1)/d\rfloor{\cdot}p'_{\varphi(n)}{+}\varphi(n){\cdot}p_i/q_i = \lfloor(n{-}1)/d\rfloor{\cdot}p'_{\varphi(n)}{+}b_{\varphi(n)}$. Otherwise whenever $f_P(n)$ is undefined then $p'_{\varphi(n)}$ is undefined. □

Lemma 4.3. *Let P be a Fractran program. Then $\mathcal{R}_P$ is productive for M_P if and only if P is terminating on all integers $n > 0$.*

Proof. Let $\sigma(n)$ be shorthand for $\mathsf{head}(\mathsf{tail}^n(\sigma))$. It suffices to show for all $n \in \mathbb{N}$: $\mathsf{M}_P(n) \to^* \bullet$ if and only if P halts on n. For this purpose we show $\mathsf{M}_P(n) \to^+ \bullet$ whenever $f_P(n+1)$ is undefined, and $\mathsf{M}_P(n) \to^+ \mathsf{M}_P(f_P(n+1)-1)$, otherwise. We have $\mathsf{M}_P(n) \to^* \mathsf{T}_{\varphi(n+1)}(\lfloor n/d\rfloor)$.

Assume that $f_P(n+1)$ is undefined. By Lemma 4.2 $p'_{\varphi(n+1)}$ is undefined, thus thus $\mathsf{M}_P(n) \to^* \bullet$ whenever $\lfloor n/d\rfloor = 0$, and otherwise we have:

$$\mathsf{M}_P(n) \to^* \mathsf{T}_{\varphi(n+1)}(\lfloor n/d\rfloor) \to^* \mathsf{mod}_d(\mathsf{tail}^{\varphi(n+1)-1}(\mathsf{M}_P))(\lfloor n/d\rfloor - 1) \to^* \mathsf{M}_P(n')$$

where $n' = (\lfloor n/d\rfloor - 1)\cdot d + \varphi(n+1) - 1 = n - d$. Clearly $n \equiv n' \pmod d$, and then $\mathsf{M}_P(n) \to^* \bullet$ follows by induction on n.

Assume that $f_P(n+1)$ is defined. By Lemma 4.2 $p'_{\varphi(n+1)}$ is defined and:

$$\mathsf{M}_P(n) \to^* \mathsf{T}_{\varphi(n+1)}(\lfloor n/d\rfloor) \to^* \mathsf{mod}_{p'_{\varphi(n+1)}}(\mathsf{tail}^{b_{\varphi(n+1)}-1}(\mathsf{M}_P))(\lfloor n/d\rfloor)$$

and hence $\mathsf{M}_P(n) \to^+ \mathsf{M}_P(n')$ with $n' = \lfloor n/d\rfloor \cdot p'_{\varphi(n+1)} + b_{\varphi(n+1)} - 1$. Then we have $n' = f_P(n+1) - 1$ by Lemma 4.2. □

Theorem 4.4. *The restriction of the productivity problem to stream specifications induced by Fractran programs and outermost-fair strategies is Π^0_2-complete.*

Proof. Since by Lemma 4.3 the uniform halting problem for Fractran programs can be reduced to the problem here, Π^0_2-hardness is a consequence of Theorem 2.2. Π^0_2-completeness follows from membership of the problem in Π^0_2, which can be established analogously as in the proof of Theorem 3.5. □

Note that Theorem 4.4 also gives rise to an alternative proof for the Π^0_2-hardness part of Theorem 3.5, the result concerning the computational complexity of productivity with respect to strategies.

References

1. Arts, T., Giesl, J.: Termination of Term Rewriting Using Dependency Pairs. Theoretical Computer Science 236, 133–178 (2000)
2. Conway, J.H.: Fractran: A Simple Universal Programming Language for Arithmetic. In: Open Problems in Communication and Computation, pp. 4–26. Springer, Heidelberg (1987)
3. Endrullis, J., Geuvers, H., Zantema, H.: Degrees of Undecidabililty of TRS Properties. In: CSL 2009 (to appear, 2009)
4. Endrullis, J., Grabmayer, C., Hendriks, D.: ProPro: an Automated Productivity Prover (2008), `http://infinity.few.vu.nl/productivity/`

5. Endrullis, J., Grabmayer, C., Hendriks, D.: Complexity of Fractran and Productivity (2009), http://arxiv.org/abs/0903.4366
6. Endrullis, J., Grabmayer, C., Hendriks, D., Isihara, A., Klop, J.W.: Productivity of Stream Definitions. In: Csuhaj-Varjú, E., Ésik, Z. (eds.) FCT 2007. LNCS, vol. 4639, pp. 274–287. Springer, Heidelberg (2007)
7. Grue Simonsen, J.: The Π^0_2-Completeness of Most of the Properties of Rewriting Systems You Care About (and Productivity). In: RTA 2009 (to appear, 2009)
8. Herman, G.T.: Strong Computability and Variants of the Uniform Halting Problem. Zeitschrift für Math. Logik und Grundlagen der Mathematik 17(1), 115–131 (1971)
9. Hinman, P.G.: Recursion-Theoretic Hierarchies. Springer, Heidelberg (1978)
10. Klop, J.W.: Term Rewriting Systems. In: Handbook of Logic in Computer Science, vol. 2, pp. 1–116. Oxford University Press, Oxford (1992)
11. Kurtz, S.A., Simon, J.: The Undecidability of the Generalized Collatz Problem. In: Cai, J.-Y., Cooper, S.B., Zhu, H. (eds.) TAMC 2007. LNCS, vol. 4484, pp. 542–553. Springer, Heidelberg (2007)
12. Lagarias, J.C.: The $3x + 1$ Problem and its Generalizations. AMM 92(1), 3–23 (1985)
13. Peyton Jones, S.: The Implementation of Functional Programming Languages. Prentice-Hall, Englewood Cliffs (1987)
14. Roşu, G.: Equality of Streams is a Π^0_2-complete Problem. In: ICFP, pp. 184–191 (2006)
15. Sijtsma, B.A.: On the Productivity of Recursive List Definitions. ACM Transactions on Programming Languages and Systems 11(4), 633–649 (1989)
16. Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press, Cambridge (2003)
17. Toyama, Y.: Strong Sequentiality of Left-Linear Overlapping Term Rewriting Systems. In: LICS, pp. 274–284. IEEE Computer Society Press, Los Alamitos (1992)

Automated Inference of Finite Unsatisfiability

Koen Claessen and Ann Lilliestr\"om

Chalmers University of Technology, Gothenburg, Sweden
{koen,annl}@chalmers.se

Abstract. We present Infinox, an automated tool for analyzing first-order logic problems, aimed at showing *finite unsatisfiability*, i.e. the absence of models with finite domains. Finite satisfiability is a semi-decidable problem, which means that such a tool can never be complete. Nevertheless, our hope is that Infinox be a complement to finite model finders in practice. The implementation consists of several different proof techniques for showing infinity of a set, each of which requires the identification of a function or a relation with particular properties. Infinox enumerates candidates to such functions and relations, and subsequently uses an automated theorem prover as a sub-procedure to try to prove the resulting proof obligations. We have evaluated Infinox on the relevant problems from the TPTP benchmark suite, with very promising results.

1 Introduction

Background and motivation. A typical situation where automated reasoning tools for first-order logic are used is the following: A number of first-order proof obligations are generated (either automatically or by a human), and need to be validated by an automated theorem prover. An example of such a situation is formal verification, where a list of proof obligations is generated from a model of a system and a number of specifications.

Although the ultimate goal of such a project is to establish the validity of the proof obligations, it is equally important to be able to *disprove* the obligations, such that useful feedback can be provided and appropriate measures can be taken.

In order to disprove first-order proof obligations, automated (counter-)model finders can be used. Examples of such model finders are saturation-based model finders (such as E [4] and SPASS [8]) and finite model finders (such as MACE [3] and Paradox [1]). A problem with this approach is that, because of the semi-decidability of first-order logic, for any choice of model finder, only a particular class of models can be found. Thus, an undesirable *gap* is created by any combination of a theorem prover and a model finder; there exist problems where neither the theorem prover nor the model finder is able to give an answer (see Fig. 1).

It is clear that the choice of model finder depends heavily on the kind of application; for example, a finite model finder is useless on problems that are satisfiable, but have no finite (counter-)models[1].

[1] By "finite model" we mean "model with a finite domain".

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 388–403, 2009.

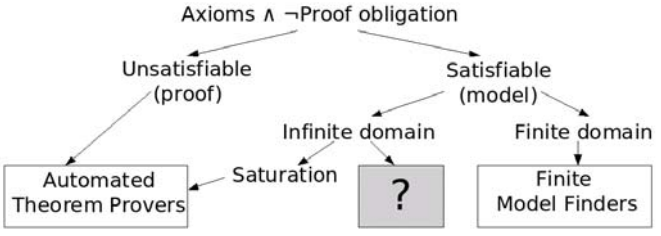


Fig. 1. Decidability of sub-classes and the corresponding tools

Thus, we believe that it is desirable to have methods that for a given problem can positively show that a given model finder can never find a model. Since finite model finders are currently one of the most widely used tools for this purpose, methods to disprove the existence of finite models are of particular interest.

The problem of showing that a given problem cannot have a finite model is called *finite unsatisfiability*. Alas, as Trakhtenbrot showed [9], the problem of finite satisfiability is semi-decidable, which means that we should never hope for a complete solution to finite unsatisfiability, and thus the gap mentioned earlier still exists (and will always exist). Nevertheless, we believe that methods that solve this problem in practice are still possible and desirable to develop.

Other Applications. Other applications of a tool that is able to show finite unsatisfiability are more direct. One example is group theory; some mathematicians are interested in finding out if for certain given properties, there exist finite groups. A finite model finder can positively establish the existence of such groups, a tool for finite unsatisfiability can answer the question negatively.

Our Approach. We have built an automated tool, called Infinox, for showing finite unsatisfiability of a given first-order theory. As far as we know, we are the first to design and build a tool of this kind.

Based on a manually chosen proof principle, the tool generates a number of candidate proof obligations, each of which implies the non-existence of a finite model. An automated theorem-prover (with a time-out) is used on each such candidate. If it succeeds, finite unsatisfiability is established; if it cannot prove the validity of any of the candidates, the tool gives up.

Note that finite unsatisfiability means that *if* a model exists, its domain must be infinite. We do not say anything about the actual existence of models. In particular, any unsatisfiable theory is also finitely unsatisfiable.

Summary. The rest of the paper is organized as follows. In section 2 we introduce a number of different proof principles for showing infinity of sets. In section 3, we show how these principles are used in the implementation of our tool. In section 4 we present our experimental results and comparisons of the different methods. Section 5 discusses future work, and section 6 concludes.

2 Proof Principles for Showing Infinite Domains

In this section, we present several proof principles that can be used to show that a given first-order theory T can only have models with infinite domains. Each of these principles is later developed into an (automatic) algorithm, the details of which are explained in the next section.

Each principle we present here is based on well-known methods for showing that a given set D must be infinite. We adapt such a standard method to our purpose by assuming that our theory T has a model, which consists of a domain D and an interpretation I for each function and predicate symbol in the theory.

2.1 Functions That Are Injective and Not Surjective

Let us start with a simple example. Throughout the paper, we show examples in clause-style, with capital letters (X, Y, Z, ...) meaning universally quantified variables, and we assume that existentially quantified variables have been skolemized.

Example 1. Consider the following theory $T1$.

$$\mathsf{suc}(X) \neq 0 \tag{1}$$

$$\mathsf{suc}(X) = \mathsf{suc}(Y) \;\Rightarrow\; X = Y \tag{2}$$

This theory is finitely unsatisfiable.

In order to realize why the theory $T1$ cannot have a finite model consider the following well-known lemma.

Lemma 1. *Given a set D, if there exists a function $f : D \to D$ that is injective and not surjective, then D must be infinite.*

Since f is not surjective, there exists an $a \in D$ such that $f(x) \neq a$ for any $x \in D$. To show that D is infinite, we now construct the infinite sequence $a, f(a), f(f(a)), \ldots, f^i(a), \ldots$. Since f is injective, no two elements in the sequence can be equal to each other, because this would imply $f^k(a) = a$ for some $k > 0$.

To connect the lemma to the example, we simply assume that there is a model $\langle D, I\rangle$ of the theory $T1$. The interpretation for the symbol suc must be a function $I(\mathsf{suc}) : D \to D$ satisfying the axioms in $T1$. According to axiom (2) this function is injective, and according to axiom (1), this function is not surjective. Therefore, the domain D must be infinite.

The proof principle used here is the basis for the first method presented in section 3.

2.2 Generalizing Equality to Reflexive Relations

It is possible to generalize the above lemma, as the following variant on example 1 shows.

Example 2. Consider the following theory $T2$.

$$\mathsf{lte}(X, X)$$
$$\neg\mathsf{lte}(\mathsf{suc}(X), 0)$$
$$\mathsf{lte}(\mathsf{suc}(X), \mathsf{suc}(Y)) \;\Rightarrow\; \mathsf{lte}(X, Y)$$

This theory is finitely unsatisfiable.

To show that the theory $T2$ cannot have finite models, we generalize lemma 1. The key insight is that not all properties of equality are needed in order to construct the infinite sequence in the proof of lemma 1.

Lemma 2. *Given a set D, if there exists a function $f : D \to D$ and a reflexive relation $R \subseteq D \times D$, such that f is injective w.r.t. R and not surjective w.r.t. R, then D must be infinite.*

A function $f : D \to D$ is injective w.r.t. a relation R iff. for all x and $y \in D$, if $R(f(x), f(y))$ then also $R(x, y)$. Similarly, a function f is surjective w.r.t. a relation R iff. for all $y \in D$, there exists an $x \in D$ such that $R(f(x), y)$.

The proof of the lemma is very similar to the proof of lemma 1. Since f is not surjective w.r.t. R, there exists an $a \in D$ such that $\neg R(f(x), a)$ for any $x \in D$. We now construct the infinite sequence $a, f(a), f(f(a)), \ldots, f^i(a), \ldots$. No two elements in the sequence can be related to each other by R, because this would imply $R(f^k(a), a)$ for some $k > 0$. And since R is reflexive, this means that no two elements in the sequence can be equal to each other.

Here, to connect the lemma to the example, we again look at properties of possible models $\langle D, I\rangle$ of the theory $T2$. The symbol interpretations $I(\mathsf{suc})$ and $I(\mathsf{lte})$ must respectively be a function and a relation that satisfy the conditions of the lemma. Therefore, the domain D must be infinite for any model.

2.3 Functions That Are Surjective and Not Injective

There are other properties that functions can have that force infinity of models, as the following example shows.

Example 3. In 1933, E.V. Huntington, presented the following axioms, as a basis for boolean algebra:

$$\mathsf{plus}(X, Y) = \mathsf{plus}(Y, X) \tag{3}$$
$$\mathsf{plus}(\mathsf{plus}(X, Y), Z) = \mathsf{plus}(X, \mathsf{plus}(Y, Z)) \tag{4}$$
$$\mathsf{plus}(\mathsf{neg}(\mathsf{plus}(\mathsf{neg}(X), Y))), \mathsf{neg}(\mathsf{plus}(\mathsf{neg}(X), \mathsf{neg}(Y)))) = X \tag{5}$$

Huntington's student Herbert Robbins conjectured that axiom (5) can be replaced with the following equation:

$$\mathsf{neg}(\mathsf{plus}(\mathsf{neg}(\mathsf{plus}(X, Y)), \mathsf{neg}(\mathsf{plus}(X, \mathsf{neg}(Y))))) = X \tag{6}$$

Neither Huntington nor Robbins was able to find a proof or a counter-example, a counter-example being a model of the theory $T3$, consisting of the the axioms

(3), (4), (5) of boolean algebra and the negation of Robbin's conjecture (6). The problem remained open until 1996, when EQP, an automated theorem prover for equational logic, found a proof after eight days of computation [2]. Thus, after great effort, (general) unsatisfiability of the theory $T3$ was established. As it turns out, showing finite unsatisfiability of this theory is much easier.

The theory $T3$ does not contain any (obvious) functions represented by terms from the signature that are injective but not surjective. However, we can find functions with a different property.

Lemma 3. *Given a set D, if there exists a function $f : D \to D$, such that f is surjective and not injective, then D must be infinite.*

The proof goes as follows: Since f is surjective, there exist right-inverse functions g (i.e. $f(g(x)) = x$ for all $x \in D$). By construction, g must be injective. Also, g cannot be surjective, because that would imply that g is also a left-inverse of f, and that f thus is injective. So, by applying lemma 1 to the function g, D must be infinite.

It turns out to be easy to show that, for any model $\langle D, I\rangle$ of theory $T3$, the function $I(\mathsf{neg}) : D \to D$ would have to be surjective but not injective, implying that D must be infinite.

Unlike for the lemma about injective and non-surjective functions, there is no obvious way to generalize equality to a more general relation in lemma 3. For example, a reflexive relation is not enough, as shown by the following counter-example:

Example 4. Consider the following theory $T4$.

$$\begin{gathered} R(X, X) \\ R(f(g(X)), X) \\ R(f(a), c) \\ R(f(b), c) \\ a \neq b \end{gathered}$$

This theory is finitely satisfiable, even though f is a function that is surjective and not injective w.r.t. the reflexive relation R.

2.4 Relations That Are Irreflexive, Transitive and Serial

The next proof principle we use in this paper only requires the existence of a relation with certain properties.

Example 5. Consider the following theory $T5$.

$$\begin{gathered} \neg \mathsf{lt}(X, X) \\ \mathsf{lt}(X, Y) \wedge \mathsf{lt}(Y, Z) \;\Rightarrow\; \mathsf{lt}(X, Z) \\ \mathsf{lt}(X, \mathsf{suc}(X)) \end{gathered}$$

This theory is finitely unsatisfiable.

To realize why $T5$ can not have finite models, consider the following lemma.

Lemma 4. *Given a non-empty set D. If there exists a relation $R : D \times D$, such that R is irreflexive, transitive and serial, then D must be infinite.*

A relation is serial if every point is related to some (other) point. To prove the lemma, we can start in any point a_0, and construct an infinite sequence $a_0, a_1, a_2, \ldots$ such that $R(a_i, a_{i+1})$ for all $i \geq 0$, which we know we can do because of seriality. Since R is transitive, we also know that $R(a_i, a_j)$ for all $i < j$. Because of irreflexivity, all a_i must be different.

The lemma shows that for all models $\langle D, I \rangle$ of $T5$, D must be infinite, because $I(\mathsf{lt})$ must be a relation on $D \times D$ that is irreflexive, transitive and serial, and D must be non-empty.

2.5 Infinite Subdomains

The following example shows how all proof principles we have seen up to now can be generalized, leading to wider applicability.

Example 6. Consider the following theory $T6$.

$$\begin{aligned} & & \mathsf{nat}(0) & \\ \mathsf{nat}(X) & \Rightarrow & \mathsf{nat}(\mathsf{suc}(X)) & \\ \mathsf{nat}(X) & \Rightarrow & \mathsf{suc}(X) \neq 0 & \\ \mathsf{nat}(X) \wedge \mathsf{nat}(Y) & \Rightarrow & \mathsf{suc}(X) = \mathsf{suc}(Y) & \Rightarrow \; X = Y \end{aligned}$$

This theory is finitely unsatisfiable.

In order to show infinity of a set, it is sufficient to show infinity of a subset. In the example, none of the proof principles we have discussed so far are directly applicable. But, given a model $\langle D, I \rangle$ of $T6$, lemma 1 can be used to show infinity of a subset of the domain D, namely the subset $D' = \{x | I(\mathsf{nat})(x)\}$. On the subset D', $I(\mathsf{suc})$ must be an injective, non-surjective function (but not necessarily on the full domain D).

The subset generalization can be used to adapt any of the proof principles we have seen in this section.

2.6 Summing Up

In this section, we have presented a number of proof principles for showing infinity of a set, and how they can be applied to show that models of first-order theories must be infinite. It turns out that these actually are all mathematically equivalent. However, since we are interested in automating the process of showing finite unsatisfiability, we are going to be forced to restrict the context in which each principle can be applied, which leads to a range of complementary methods.

This list of chosen proof principles may seem, and indeed is, arbitrarily chosen. There are many other proof principles that we could have added, or may add in the future. We come back to this in section 5.

3 Automating Finite Unsatisfiability

In this section, we show how the proof principles presented in the previous section can be turned into the automated method that we have implemented in our tool Infinox.

3.1 The Main Architecture

Our method works as follows. Given the choice of a proof principle, we have to search for a function, a relation, or a combination thereof, with some desired properties. For example, the second method from the previous section requires us to find a function f and a relation R such that f is both injective and non-surjective with respect to R.

The key idea behind realizing this as a implementable method, is that we enumerate *candidate* combinations for these functions and relations (f, R), and use an automated theorem prover (ATP) (in our case we chose E [4]) to try to prove the desired properties (see figure 2). For example, using the second method, the proof obligation we generate for a given theory T, and candidate (f, R) is:

$$\begin{aligned}(\bigwedge T) \Longrightarrow\ & (\forall x.\ R(x,x)) \\ \wedge\ & (\forall xy.\ R(f(x), f(y)) \Rightarrow R(x,y)) \\ \wedge\ & \neg(\forall x \exists y.\ R(f(x), y)\end{aligned}$$

If the above formula can be proven, we know that the theory T cannot have any finite models.

ATP systems are necessarily incomplete in the negative case, so in order to use them as a sub-procedure, we need to set a time-out. The time-out needs to

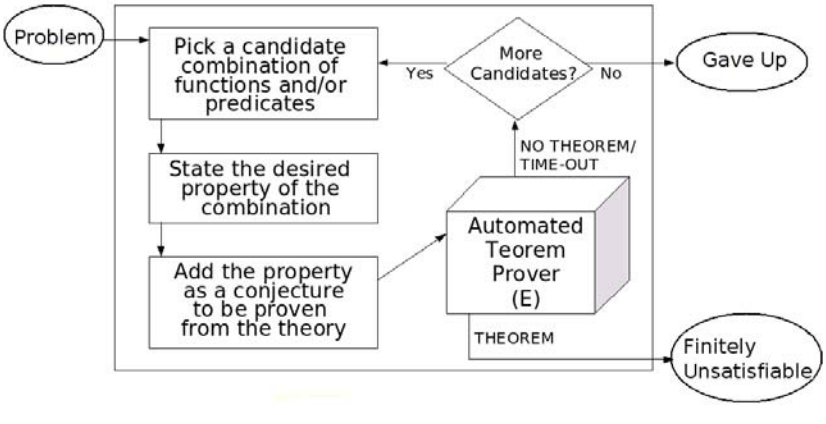


Fig. 2. The architecture of the automated methods

be long enough to produce meaningful results, and short enough to process the relevant test cases within a reasonable time.

Using the key idea, moves us from mathematical reasoning about functions and relations on the *domain* of the models, to automated reasoning about the symbols in the theory, which is a syntactic process. Thus, we need to have a syntactic representation for the functions and relations we are considering. A natural choice is to use *terms* (with one variable X) to represent functions, and *literals* (with two variables X and Y) to represent relations.

Even though the lemmas in the previous section all can be read as if-and-only-if, which suggests that a method based on them would be complete, the method we actually implement is incomplete. The incompleteness stems from the fact that we only consider functions, relations and subsets that can be syntactically represented. This is the first (and main) choice we make in our implementation where incompleteness is introduced.

3.2 Enumerating Functions and Relations

In the search for terms and literals that possess a certain property, the choice of candidates is of utmost importance. In theory, there is an unbounded number of possibilities that should be tested. In practice, in order to keep the process managable, we restrict ourselves to a limited, finite subset of candidates. This is the second choice that leads to incompleteness.

A suitable basis for candidate generation is the terms and literals that already occur as subterms and literals in the theory. These provide a natural limitation, and are intuitively more likely to match a specific pattern, compared to an arbitrarily chosen term or literal.

We define the *arity* of a term or a literal to be the number of variable occurrences in the term or literal. For example, the term $f(g(X), a)$ has arity 1, and the literal $P(f(X, Z))$ has arity 2.

Simple Symbol Search. One basic enumeration method for terms that represent functions f is to take all subterms $t(X)$ from the theory that have arity 1. Similarly, enumerating literals that represent relations can be done by taking all literals $l(X, Y)$ from the theory that have arity 2. (Variable occurrences other than X or Y are appropriately renamed to either X or Y.)

Although this method works, it is very restrictive and can not solve many problems (see section 4).

Greater Arities. We can augment the set of candidate terms and literals by including the ones with greater arities, and using existential quantification to quantify away the excessive variable occurrences to reduce the arity of the term or the literal.

For example, when looking for a function with arity 1 that satisfies a property ϕ, we might want to consider a term $t(X, Y)$ (containing two variable occurrences). The list of terms we consider in such a case is denoted $t(*, X)$, $t(X, *)$ and $t(X, X)$. Here, $*$ means an existentially quantified variable, so that when

we want to ask if the function represented by the term $t(*, X)$ satisfies ϕ, we actually check:

$$\exists A.\ \phi[t(A, X)]$$

This idea can be generalized to terms and literals of greater arity. For example, the terms we check when considering a term $t(X, Y, Z)$ are $t(X, *, *)$, $t(*, X, *)$, $t(*, *, X)$, $t(X, X, *)$, $t(X, *, X)$, $t(X, X, *)$, and $t(X, X, X)$. Each $*$ occurrence refers to a unique existentially quantified variable. For example, checking if $t(*, *, X)$ satisfies ϕ amounts to checking:

$$\exists AB.\ \phi[t(A, B, X)]$$

This technique greatly improves the applicability of our methods.

3.3 Infinite Sub-domains

As explained in the previous section, any of our methods can be generalized by searching for a subset of the domain that is infinite. Subsets can be syntactically represented by literals with arity 1. Thus, the most general method based on injective and non-surjective functions will have to find a combination of a function f, a relation R and a subset S such that the right properties hold.

So far, we have implemented and evaluated the subset generalization only for the method that looks for injective and non-surjective functions. To implement this, we need to show the following property:

$$\begin{aligned} & \forall X.\ S(X) \Rightarrow S(f(X)) \\ \wedge\ & \forall X.\ S(X) \Rightarrow R(X, X) \\ \wedge\ & \forall XY.\ S(X) \wedge S(Y) \wedge R(f(X), f(Y)) \Rightarrow R(X, Y) \\ \wedge\ & \neg(\forall X.\ S(X) \Rightarrow \exists Y.S(Y) \wedge R(f(X), Y)) \end{aligned}$$

In other words, we have to show that the subset S is closed under the function f, and whenever we quantify in the requirements for reflexitivity, injectivity and non-surjectivity, we do so only over the elements of the subset S.

3.4 Filtering

By blindly generating test cases, the search space grows very quickly. It is therefore desirable to filter out the combinations that, for some reason, are unsuitable.

For example, in the method that searches for injective and non-surjective functions with respect to a relation, we can filter away all relations that are not reflexive, before we even choose what function to combine it with. This reduces the search space and makes a practical difference.

Similarly, when applying the above method to a subset of the domain, it is not necessary to check all combinations of functions, relations and subsets. Instead, we generate the candidate combinations in two steps; first, we collect all pairs of functions f and subsets S, such that S is closed under f. Typically, only a few subsets will be closed under some function, and thus a majority of the subsets

will not be represented among these pairs. Second, we collect all pairs of relations R and subsets S such that R is reflexive in S, for the S that are still left. In this way, we reduce the candidate combinations to include only the triples (f, R, S) where S is closed under f and R is reflexive in S.

3.5 Zooming

Often, it is not feasible to check all of the generated combinations of terms and literals. The use of existential quantification extends the search space significantly, making an exhaustive search impossible within a reasonable time limit. This is especially the case for large theories containing many literals and subterms.

Often, a very small subset of the axioms is responsible for the finite unsatisfiability of the whole theory. We use a method that we call *zooming* to approximate this subset. The aim is to be able to "zoom" in on the relevant part of the theory, so that we only need to check the combinations of terms and literals found in this smaller theory. In many cases, this significantly reduces the number of candidate terms and literals to check. (We still use the whole theory to establish the desired properties in the proving process.)

We start by creating a simple procedure to approximate if a given subtheory is finitely satisfiable. We do this by using a finite model finder (in our case we chose Paradox [1]); if we fail to find a finite model within a certain time limit, we simply assume that there is no finite model for that subtheory.

Now, a given theory for which we have failed to find a finite model, can be weakened by the removal of some of the axioms. If we cannot find a finite model of this new theory, we can assume that the removed axioms were not responsible for the absence of a finite model. We can continue removing axioms until we reach a smallest axiom set for which a finite model cannot be found.

There are two risks involved using zooming. Firstly, it relies on the assumption that if no finite model has been found within a set time limit, then no finite model exists. Thus, we might zoom in on a part of the theory that does have a finite model. The time limit used in the search for a finite model is thus a factor that needs careful consideration! Secondly, we might successfully zoom in on a small part of the theory without finite models, but it might be much harder to show finite unsatisfiability for this subtheory. This typically happens when the original theory belongs to the class of problems that Infinox can solve, but the subtheory does not. Despite the risks involved, using zooming enables Infinox to solve a large set of problems which would have been impossible to solve without.

4 Results

Infinox has been evaluated using problems from the *TPTP Problem Library* [6] that we considered relevant. In this section, we will explain what problems we chose, how Infinox performed overall on these problems, and how each of the individual methods compared to the rest.

4.1 Relevant Problems and Overall Results

To create the set of evaluation problems, we started with all problems from the TPTP of April 2008. We excluded problems already identified as *unsatisfiable*, since these are by definition finitely unsatisfiable. Problems identified as *theorems* were also excluded, since these have unsatisfiable negations, and thus lack counter-models. In the experimental evaluation, we also left out problems identified as *satisfiable* with known finite models, and *countersatisfiable* with known finite countermodels, although we used these internally to test soundness.

The remaining 1272 problems have the following classification: *Open* (the abstract problem has never been solved): 27 problems; *Unknown* (the problem has never been solved by an ATP system): 1075 problems; *Satisfiable* (there exist models of the axioms): 122 problems with no known finite model; and *Countersatisfiable* (there exist models of the negation of the conjecture): 48 problems with no known finite counter-model.

The experiments were performed on a 2x Dual Core processor operating at 1 GHz, with a time-out of 15 minutes per proof method and problem, and a time-out of two seconds for each call to E and Paradox. The exact proof methods we used are listed in the next subsection. In total, Infinox classified 413 out of the 1272 test problems as finitely unsatisfiable. (Note that the actual number of the test problems that are finitely unsatisfiable is unknown!) The success rate is thus at least 32%, divided over the following categories: *Unknown*: 388 (out of 1075); *Open*: 3 (out of 27); *Satisfiable*: 21 (out of 122); and *CounterSatisfiable*: 1 (out of 48).

4.2 Comparisons

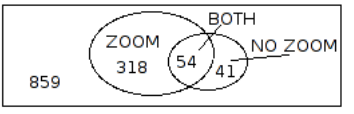
Since there is no other tool similar to Infinox, there are no previous results to compare with. Instead, we compare the performance of the different methods. We are not only interested in the total number of successful tests of a method, but also in what the given method contributes to the overall result; here we are looking for the so-called State Of The Art Contributors (SOTAC) in the terminology of CASC [5]; the methods that were able to classify problems that no other method could.

For the purpose of readability, we introduce the following abbreviations for the methods:

R. Search for functions that are injective and non-surjective w.r.t. a reflexive relation.
RP. Search for functions that are injective and non-surjective w.r.t. a reflexive relation, on a subdomain defined by a predicate.
SNI. Search for surjective and non-injective functions.
SR. Search for relations that are irreflexive, transitive and serial.

The letter "**Z**" appended to the above codes indicate the added use of zooming to select test functions, relations and predicates.

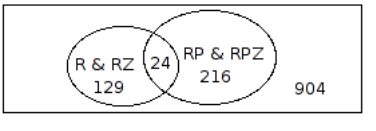
Zooming. We first investigate how zooming, as a way to select candidate functions and relations, affects the result. The diagram below shows that zooming significantly increases the ratio of classified problems. We compare the sets of all problems classified by any of the listed methods, with and without the use of zooming.



Using zooming, 318 additional problems are classified that could otherwise not be classified within the time limit of 15 minutes. The zoomed versions fail on 41 out of the 95 problems that the plain methods classify successfully. The results indicate that the use of zooming is in most cases preferrable. Still, its benefits are not guaranteed, and it may be of value try the plain method when zooming fails. As explained in section 3.5, a too short time-out setting for Paradox may cause it to focus on the "wrong" sub-theory. On the other hand, with a too long time-out, there might not be enough time to test all candidates.

Another interesting observation is that many problems with shared axiom sets reduce to a common subset of axioms after zooming. When using zooming, it is therefore necessary only to test one problem out of all problems that reduce to the same "zoomed" theory. In this way, the number of tests performed involving zooming could be reduced by over 75%.

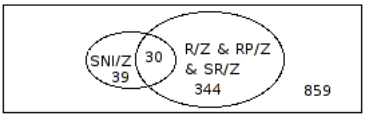
Infinite Subsets. The next comparison involves the effect of using the infinite subset principle. The diagram below shows how the use of subset predicates affects the result. Since the subset principle has only been implemented for the R method, we compare the set of problems classified by either of the methods R and RZ with the set of problems classified by either of RP and RPZ.



We see that the intersection of the two sets is relatively small, and thus that the two methods complement each other well. Ideally, they should be combined, (which is easily done by adding the predicate p such that $p(X) = true$ for all X), however, combining several methods will generally require a longer global time-out. The use of limiting predicates produced a somewhat better result, with a total of 240 classified problems, compared to R and RZ, that classified 153 problems. Together, the two methods account for almost 90% of the total number of classified problems.

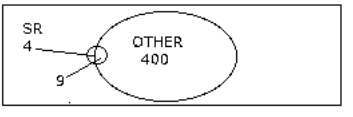
Injective or Surjective Functions. The diagram below shows the contribution of the methods SNI and SNIZ to the overall result. In total, 69 problems are

classified with these methods, out of which 39 were not classified by any other method.



A reason that surjective and non-injective functions do not occur as frequently as injective and non-surjective functions may be that we are limited to using standard equality. With fewer test cases, we are less likely to come across a term that fits our description. Another reason may be that the test problems are initially constructed by people. It may be more intuitive to deal with a certain kind of functions (injective and non-surjective) rather than the reverse (non-injective and surjective), and thus these functions occur more frequently.

Serial Relations. In total SR and SRZ classify only 13 of the test problems. Despite this, they classify 4 problems on which all other methods failed.



The reasons for the poor results of this method are unclear. Further generalizations of this method are desirable, possibly with longer time-outs for E. This method should also be evaluated in combination with limiting predicates, which is future work.

Reflexive Predicates. The added use of reflexive predicates as equality relation has shown to be a useful generalization. It accounts for 41 out of the 146 problems classified by RZ, and 13 out of the 235 problems classifed by RPZ.

Existentially Quantified Variables. Using existential quantification to generate terms and predicates has proved to be of major significance to the results. In 104 out of the 146 problems classified by RZ, and 219 out of the 235 problems classified by RPZ, the identified terms and/or predicates include variables that are existentially quantified.

Summary. According to our results, the most successful of the described methods are the ones based on the search for functions that are injective and not surjective, or surjective and not injective. Existential quantification has made a major contribution to the results of all of the presented methods. While some techniques have been more successful than others, all methods that we describe here are SOTAC; they contribute to the overall result.

5 Future Work

Naturally, since any automated solution to finite unsatisfiability will always be incomplete, the room for improvement is without limit.

One possible enhancement to our methods is to refine the techniques of function and relation selection. One way in which this could be done is by introducing new symbols with new axioms. For example, if it is known that a function f must always have an inverse, one could introduce a new symbol f_{inv}, together with axioms. The new symbol can then be used to construct more interesting terms. For relations (and subsets) in particular, one could generalize the search from literals to formulas with a boolean structure and even quantification.

In order to be able to manage complexity, with any extension of the search space, limitations on the search space should also be introduced. Possible future improvements include a more efficient and reliable zooming algorithm, and a thorough analysis of what candidates are meaningful to test.

Another opportunity for improvement is the addition of new proof principles. Related to this is our desire to generalize the existing ones into fewer, wider applicable principles.

We are working on a domain-specific language to specify proof principles and term and literal selection methods externally from the tool. This will allow users to experiment with their own specialized principles.

Finally, we are also working on an extensive evaluation of each method, something from which future implementations would benefit greatly. This kind of information might for example be used to come up with a sensible automatic strategy selection or orchestration. Part of this evaluation is an analysis of how the different time-outs should be balanced for the best possible results. Right now, Infinox has three different time-out settings: The global time-out (when does Infinox give up?), the theorem proving time-out (how long time do we spend on each candidate?), and the finite model finder time-out (used in zooming). Related to these is the choice of how many candidate combinations do we check. Choosing these values is a delicate process that we have not sufficiently mastered yet and something for which further study is needed.

6 Conclusions

We have designed methods for automatic, but incomplete, detection of finite unsatisfiability, and have implemented these methods in a tool called Infinox. Our methods are a combination of a *proof principle*, which identifies properties of a function (or a relation) that imply infinity of its domain, and a *term selection strategy* which specifies what kinds of functions and relations we search for.

We have found 6 practically useful distinct proof principles, that turned out to be complementary to each other. Also, each of these principles is a State Of The Art Contributor when evaluated on the TPTP, although some more so than

others. Furthermore, for the general success of the methods it turns out to be important to use *existential quantification* to extend the search for functions and relations, and to use *zooming* to limit the search.

Out of the relevant problems in the TPTP (the ones where it was not already known whether or not a finite model exists), Infinox could automatically classify over 32% as finitely unsatisfiable. Some of these were previously classified as "Open" and many as "Unknown", demonstrating that it is often easier to show finite unsatisfiability than general unsatisfiability. One remarkable consequence of Infinox is that "FinitelyUnsatisfiable" has now been added as a problem status to the TSTP [7]. Since Infinox is the first tool of its kind, it is hard to compare against alternative methods.

Limitations. Not surprisingly, there are many problems that are finitely unsatisfiable where Infinox is incapable of giving an answer. These fall into three different categories:

(1) Problems where we need to identify a function or a relation that cannot be represented by a term or a literal. This limitation can be remedied (but never fully) in two ways: (a) By extending the number of proof principles for showing infinity; although mathematically these principles are all equivalent, in practice there might be a term or a literal that fits one but not the other; (b) By augmenting the representations for functions or literals, for example by some of the methods mentioned in section 5.

(2) Problems where there exists a term or literal for the chosen proof principle, but where this term is not among the set of candidates we try. The only remedy here is to augment the set of candidates to try. However, augmenting the set of candidates is often paired with a search space explosion that has to be dealt with.

(3) Problems where we try the right term or literal for the chosen proof principle, but where the theorem prover fails to validate the proof obligation. This can be remedied by allowing more resources for the theorem prover, or by trying different theorem provers.

Acknowledgments. We thank the anonymous referees for their helpful comments.

References

1. Claessen, K., Sörensson, N.: New techniques that improve MACE-style model finding. In: Proc. of Workshop on Model Computation (MODEL) (2003)
2. McCune, W.: Solution of the Robbins problem. Journal of Automated Reasoning 19(3), 263–276 (1997)
3. McCune, W.: Prover9 and Mace4 (2006), `http://www.cs.unm.edu/~mccune/mace4/`
4. Schulz, S.: E - a brainiac theorem prover. AI Commun. 15(2-3), 111–126 (2002)
5. Sutcliffe, G., Suttner, C.: The State of CASC. AI Communications 19(1), 35–48 (2006)

6. Sutcliffe, G., Suttner, C.B.: The TPTP Problem Library: CNF Release v1.2.1. Journal of Automated Reasoning 21(2), 177–203 (1998)
7. Sutcliffe, G.: TSTP – thousands of solutions of theorem provers (2008), http://www.tptp.org/TSTP
8. The SPASS Team. The Spass theorem prover (2007), http://www.spass-prover.org/
9. Vännänen, J.: A Short course on finite model theory, Department of Mathematics, University of Helsinki (2006)

Decidability Results for Saturation-Based Model Building

Matthias Horbach and Christoph Weidenbach

Max-Planck-Institut für Informatik
Saarbrücken, Germany
{horbach,weidenb}@mpi-inf.mpg.de

Abstract. Saturation-based calculi such as superposition can be successfully instantiated to decision procedures for many decidable fragments of first-order logic. In case of termination without generating an empty clause, a saturated clause set implicitly represents a minimal model for all clauses, based on the underlying term ordering of the superposition calculus. In general, it is not decidable whether a ground atom, a clause or even a formula holds in this minimal model of a satisfiable saturated clause set.

We extend our superposition calculus for fixed domains with syntactic disequality constraints in a non-equational setting. Based on this calculus, we present several new decidability results for validity in the minimal model of a satisfiable finitely saturated clause set that in particular extend the decidability results known for ARM (Atomic Representations of term Models) and DIG (Disjunctions of Implicit Generalizations) model representations.

1 Introduction

Saturation-based calculi such as ordered resolution [2] or superposition [16] can be successfully instantiated to decision procedures for many decidable fragments of first-order logic [8,14,11]. Given a set N of clauses, saturation means the exhaustive recursive application of all calculus inference rules up to redundancy resulting in a potentially infinite clause set N^*. If the calculus is complete, either N^* contains the empty clause, meaning that N is unsatisfiable, or the set N^* implicitly represents a unique minimal Herbrand model $N^*_{\mathcal{I}}$ produced by a model generating operator out of N^*. The model generating operator is based on a reduction ordering $\prec$ that is total on ground terms and also used to define the redundancy notion and inference restrictions underlying a superposition calculus.

Given a model representation formalism for some clause set N, according to [7,4], each model representation should ideally represent a *unique* single interpretation, provide an *atom test* deciding ground atoms, support a *formula evaluation* procedure deciding arbitrary formulae, and an algorithm deciding *equivalence* of two model representations.

By definition, the superposition model generating operator produces a unique minimal model $N^*_{\mathcal{I}}$ out of N^* according to $\prec$. This satisfies the above uniqueness

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 404–420, 2009.

postulate. As first-order logic is semi-decidable, the saturated set N^* may be infinite and hence decision procedures for properties of $N^*_{\mathcal{I}}$ are hard to find. Even if N^* is finite, any other properties like the ground atom test, formula evaluation, and equivalence of models are still undecidable in general, showing the expressiveness of the saturation concept. For particular cases, more is known. The atom test is decidable if N^* is finite and all clauses $\Gamma \rightarrow \Delta, s{\approx}t \in N^*$ are universally reductive, i.e., $\mathrm{vars}(\Gamma, \Delta, t) \subseteq \mathrm{vars}(s)$ and s is the strictly maximal term in $\Gamma \rightarrow \Delta, s{\approx}t$ or a literal in Γ is selected [10]. This basically generalizes the well-known decidability result of the word problem for convergent rewrite systems to full clause representations. Even for a finite universally reductive clause set N^*, clause evaluation (and therefore formula evaluation) and the model equivalence of two such clause sets remain undecidable.

More specific resolution strategies produce forms of universally reductive saturated clause sets with better decidability properties. An eager selection strategy results in a hyper-resolution style saturation process where, starting with a Horn clause set N, eventually all clauses contributing to the model $N^*_{\mathcal{I}}$ are positive units. Such strategies decide, e.g., the clause classes $\mathcal{VED}$ and $\mathcal{PVD}$ [7,4]. The positive unit clauses in N^* represent so-called ARMs (Atomic Representations of term Models). Saturations of resolution calculi with constraints [16,4] produce in a similar setting positive unit clauses with constraints. Restricted to syntactic disequality constraints, the minimal model of the saturated clause set can be represented as a DIG (Disjunctions of Implicit Generalizations). DIGs generalize ARMs in that positive units may be further restricted by syntactic disequations. In [9] it was shown that the expressive power of DIGs corresponds to the one of so-called *contexts* used in the model evolution calculus [3] and that the ground atom test as well as the clause evaluation test and the equivalence test are decidable.

We extend the results of [9] for DIGs and ARMs to more expressive formulae with quantifier alternations using saturation-based techniques. We first enhance the non-equational part of our superposition calculus for fixed domains [12] with syntactic disequations (Section 3). The result is an ordered resolution calculus for fixed domains with syntactic disequations that is sound (Proposition 1) and complete (Theorem 1). Given an ARM representation N^*, we show that

$$N^*_{\mathcal{I}} \models \forall \vec{x}.\exists \vec{y}.\phi \text{ and } N^*_{\mathcal{I}} \models \exists \vec{x}.\forall \vec{y}.\phi$$

are both decidable, where ϕ is an arbitrary quantifier-free formula (Theorem 4). For more expressive DIG representations N^*, we show among other results that

$$N^*_{\mathcal{I}} \models \forall \vec{x}.\exists \vec{y}.C \text{ and } N^*_{\mathcal{I}} \models \exists \vec{x}.\forall \vec{y}.C'$$

are decidable for any clause C, and for any clause C' in which no predicate occurs both positively and negatively (Theorem 3). In order to cope with existential quantifiers in a minimal model semantics, we do not Skolemize but treat existential quantifiers by additional constraints.

Missing proofs can be found in a technical report [13].

2 Preliminaries

We build on the notions of [1,17] and shortly recall here the most important concepts as well as the specific extensions needed for the new calculus.

Terms and Clauses. Let $\Sigma = (\mathcal{P}, \mathcal{F})$ be a *signature* consisting of a finite set $\mathcal{P}$ of predicate symbols of fixed arity and a finite set $\mathcal{F}$ of function symbols of fixed arity, and let $X \cup V$ be an infinite set of variables such that X, V and $\mathcal{F}$ are disjoint and V is finite. Elements of X are called *universal variables* and denoted as x, y, z, and elements of V are called *existential variables* and denoted as v, possibly indexed.

We denote by $\mathcal{T}(\mathcal{F}, X')$ the set of all *terms* over $\mathcal{F}$ and $X' \subseteq X \cup V$ and by $\mathcal{T}(\mathcal{F})$ the set of all *ground terms* over $\mathcal{F}$. Throughout this article, we assume that $\mathcal{T}(\mathcal{F})$ is non-empty. An equation or disequation is a multiset of two terms, usually written as $s \simeq t$ or $s \not\simeq t$, respectively. A multiset $s_1 \simeq t_1, \ldots, s_n \simeq t_n$ of equations is often written as $\vec{s} \simeq \vec{t}$. An *atom* over Σ is an expression of the form $P(t_1, \ldots, t_n)$, where $P \in \mathcal{P}$ is a predicate symbol of arity n and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, X)$ are terms. To improve readability, atoms $P(t_1, \ldots, t_n)$ will often be denoted $P(\vec{t})$.

A *clause* is a pair of multisets of atoms, written $\Gamma \to \Delta$, interpreted as the conjunction of all atoms in the *antecedent* Γ implying the disjunction of all atoms in the *succedent* Δ. A clause is *Horn* if Δ contains at most one atom, and a *unit* if $\Gamma \to \Delta$ contains exactly one atom. The *empty clause* is denoted by $\square$.

Constrained Clauses. A *constraint* α over $\Sigma = (\mathcal{P}, \mathcal{F})$ (and V) is a multiset of equations $v \simeq t$ and disequations $s \not\simeq t$ where $v \in V$ and $s, t \in \mathcal{T}(\mathcal{F}, X)$. We denote the equations in α by $\alpha^{\simeq}$ and the disequations by $\alpha^{\not\simeq}$. We call $\alpha^{\simeq}$ the *positive part* of α and say that α is *positive* if $\alpha = \alpha^{\simeq}$. A constraint is *ground* if it does not contain any universal variables. A ground constraint is *satisfiable* if (i) all of its equations are of the form $v \simeq t$ or $t \simeq t$ and (ii) it does not contain a disequation of the form $t \not\simeq t$. This means that we interpret α as a conjunction and $\simeq$ and $\not\simeq$ as syntactic equality and disequality, respectively.

Let $V = \{v_1, \ldots, v_n\}$ with $v_i \neq v_j$ for $i \neq j$. A *constrained clause* $\alpha \parallel C$ *over* Σ *(and* V*)* consists of a constraint α and a clause C over Σ, such that each v_i appears exactly once in α and $\alpha^{\simeq} = v_1 \simeq t_1, \ldots, v_n \simeq t_n$. The set of all variables occurring in a constrained clause $\alpha \parallel C$ is denoted by $\mathrm{vars}(\alpha \parallel C)$. A constrained clause $\alpha \parallel C$ is called *ground* if it does not contain any universal variables, i.e. if $\mathrm{vars}(\alpha \parallel C) = V$. We abbreviate $\alpha \parallel C$ as $\alpha^{\not\simeq} \parallel C$ if $\mathrm{vars}(\alpha^{\simeq}) \cap (\mathrm{vars}(\alpha^{\not\simeq}) \cup \mathrm{vars}(C)) = \emptyset$ and no variable appears twice in $\alpha^{\simeq}$. Such a constrained clause is called *unconstrained* if $\alpha^{\not\simeq}$ is empty. We regard clauses as a special case of constrained clauses by identifying a clause C with $\parallel C$.

Substitutions. A *substitution* σ is a map from $X \cup V$ to $\mathcal{T}(\mathcal{F}, X)$ that acts as the identity map on all but a finite number of variables. We (non-uniquely) write $\sigma : Y \to Z$ if σ maps every variable in $Y \subseteq X \cup V$ to a term in $Z \subseteq \mathcal{T}(\mathcal{F}, X)$ and σ is the identity map on $(X \cup V) \setminus Y$. A substitution is identified with

its extensions to terms, equations, atoms, and constrained clauses, where it is applied to both constraint and clausal part.

The most general unifier of two terms s, t or two atoms A, B is denoted by $\mathrm{mgu}(s,t)$ or $\mathrm{mgu}(A,B)$, respectively. If α_1 and α_2 are positive constraints of the form $\alpha_1 = v_1 \simeq s_1, \ldots, v_n \simeq s_n$ and $\alpha_2 = v_1 \simeq t_1, \ldots, v_n \simeq t_n$, then we write $\mathrm{mgu}(\alpha_1, \alpha_2)$ for the most general simultaneous unifier of $(s_1, t_1), \ldots, (s_n, t_n)$.

Orderings. Any ordering $\prec$ on atoms can be extended to clauses in the following way. We consider clauses as multisets of occurrences of atoms. The occurrence of an atom A in the antecedent is identified with the multiset $\{A, A\}$; the occurrence of an atom A in the succedent is identified with the multiset $\{A\}$. Now we lift $\prec$ to atom occurrences as its multiset extension, and to clauses as the multiset extension of this ordering on atom occurrences.

An occurrence of an atom A is *maximal* in a clause C if there is no occurrence of an atom in C that is strictly greater with respect to $\prec$ than the occurrence of A. It is *strictly maximal* in C if there is no other occurrence of an atom in C that is greater than or equal to the occurrence of A with respect to $\prec$. Constrained clauses are ordered by their clausal part, i.e. $\alpha \parallel C \prec \beta \parallel D$ iff $C \prec D$.[1]

Throughout this paper, we will assume a well-founded reduction ordering $\prec$ on atoms over Σ that is total on ground atoms.

Herbrand Interpretations. A *Herbrand interpretation* over the signature Σ is a set of atoms over Σ. We recall from [1] the construction of the special Herbrand interpretation $N_{\mathcal{I}}$ derived from an unconstrained (and non-equational) clause set N. If N is consistent and saturated with respect to a complete inference system (possibly using a literal selection function), then $N_{\mathcal{I}}$ is a minimal model of N with respect to set inclusion. Let $\prec$ be a well-founded reduction ordering that is total on ground terms. We use induction on the clause ordering $\prec$ to define sets of atoms $\mathrm{Prod}(C), R(C)$ for ground clauses C over Σ. Let $\mathrm{Prod}(C) = \{A\}$ (and we say that C *produces* A), if $C = \Gamma \to \Delta, A$ is a ground instance of a clause $C' \in N$ such that (i) no literal in Γ is selected, (ii) A is a strictly maximal occurrence of an atom in C, (iii) A is not an element of $R(C)$, (iv) $\Gamma \subseteq R(C)$, and (v) $\Delta \cap R(C) = \emptyset$. Otherwise $\mathrm{Prod}(C) = \emptyset$. In both cases, $R(C) = \bigcup_{C \succ C'} \mathrm{Prod}(C')$. We define the interpretation $N_{\mathcal{I}}$ as $N_{\mathcal{I}} = \bigcup_C \mathrm{Prod}(C)$. We will extend this construction of $N_{\mathcal{I}}$ to constrained clauses in Section 3.

Constrained Clause Sets and Their Models. Given a constrained clause set N, a Herbrand interpretation $\mathcal{M}$ over $\Sigma = (\mathcal{P}, \mathcal{F})$ is a *model* of N, written $\mathcal{M} \models N$, if and only if there is a substitution $\sigma : V \to \mathcal{T}(\mathcal{F})$ such that for every constrained clause $\alpha \parallel C \in N$ and every substitution $\tau : \mathrm{vars}(\alpha \parallel C) \setminus V \to \mathcal{T}(\mathcal{F})$,

[1] This ordering on constrained clauses differs from the one in [12]. Atoms there are equational, requiring superposition into constraints and that constrained clauses must also be ordered by their constraints. Constrained clauses here do not contain equational atoms but only syntactic (dis-)equations, so this is not necessary.

$\alpha\sigma\tau$ being valid (i.e. satisfiable) implies that $C\tau$ is true in $\mathcal{M}$.[2] If N is finite, this is equivalent to the formula $\exists\vec{v}.\bigwedge_{\alpha \parallel C \in N} \forall\vec{x}.\alpha \rightarrow C$ being true in $\mathcal{M}$, where $\vec{x}$ are the universal variables in $\alpha \parallel C$ and $\simeq$ is interpreted as syntactic equality.[3] N is *Herbrand-satisfiable over* Σ if it has a Herbrand model over Σ.

Let M and N be two (constrained or unconstrained) clause sets. We write $N \models_\Sigma M$ if each Herbrand model of N over Σ is also a model of M, and we write $N \models_{Ind} M$ if $N_\mathcal{I} \models N$ and $N_\mathcal{I} \models M$.

Inferences, Redundancy and Derivations. An *inference rule* is a relation on constrained clauses. Its elements are called *inferences* and written as

$$\frac{\alpha_1 \parallel C_1 \quad \dots \quad \alpha_k \parallel C_k}{\alpha \parallel C} \quad .$$

The constrained clauses $\alpha_1 \parallel C_1, \dots, \alpha_k \parallel C_k$ are called the *premises* and $\alpha \parallel C$ the *conclusion* of the inference. An *inference calculus* is a set of inference rules.

A ground constrained clause $\alpha \parallel C$ is called *redundant* with respect to a set N of constrained clauses if α is unsatisfiable or if there are ground instances $\alpha_1 \parallel C_1, \dots, \alpha_k \parallel C_k$ of constrained clauses in N with satisfiable constraints and the common positive constraint part $\alpha_1^\simeq = \dots = \alpha_k^\simeq = \alpha^\simeq$ such that $\alpha_i \parallel C_i \prec \alpha \parallel C$ for all i and $C_1, \dots, C_k \models C$.[4] A non-ground constrained clause is redundant if all its ground instances are redundant.

Given an inference system, a ground inference is *redundant* with respect to N if some premise is redundant, or if the conclusion is redundant, where the maximal premise is used instead of the conclusion in the ordering constraint. A non-ground inference is redundant if all its ground instances are redundant. A constrained clause set N is *saturated* with respect to the inference system if each inference with premises in N is redundant wrt. N.

A *derivation* is a finite or infinite sequence $N_0, N_1, \dots$ of constrained clause sets such that for each i, there is an inference with premises in N_i and conclusion $\alpha \parallel C$ that is not redundant wrt. N_i, such that $N_{i+1} = N_i \cup \{\alpha \parallel C\}$. A derivation $N_0, N_1, \dots$ is *fair* if every inference with premises in the constrained clause set $N_\infty = \bigcup_j \bigcap_{k \geq j} N_k$ is redundant with respect to $\bigcup_j N_j$.

3 A Constrained Ordered Resolution Calculus

In [12], we introduced a superposition-based calculus to address the problem whether $N \models_\Sigma \forall\vec{x}.\exists\vec{y}.\phi$, where N is a set of unconstrained clauses and ϕ a formula over Σ. There, both N and ϕ may contain equational atoms. The basic idea is to express the negation $\exists\vec{x}.\forall\vec{y}.\neg\phi$ of the query as a constrained

[2] When considering constrained clauses, the usual definition of the semantics of a clause $\alpha \parallel C$ (where all variables are universally quantified) in the literature is simply the set of all ground instances $C\sigma$ such that σ is a solution of α (cf. [2,16]). This definition does not meet our needs because we have existentially quantified variables, and these interconnect all clauses in a given constrained clause set.

[3] Finiteness of N is only required to ensure that the formula is also finite.

[4] Note that $\models$ and $\models_\Sigma$ agree on ground clauses.

clause set N' where all constraints are positive and the constraint part enables a special treatment of the existential variables without Skolemization. For example, $\exists x.\forall y.\neg P(x,y)$ corresponds to the constrained clause set $N' = \{v \simeq x \parallel P(x,y) \rightarrow\}$. If the saturation of $N \cup N'$ terminates, it is decidable whether $N \cup N'$ has a Herbrand model over Σ, i.e. whether $N \models_\Sigma \forall \vec{x}.\exists \vec{y}.\phi$.

In our current setting with constrained clauses that contain only predicative atoms, only two rules from the original calculus are needed. Extending these to clauses with constraints containing both equations and disequations yields the constrained ordered resolution calculus COR given by the following two inference rules. The rules are defined with respect to a *selection function* that assigns to each clause a possibly empty set of atom occurrences in its antecedent. Such occurrences are called *selected* and every inference with this clause has to use a selected atom.

Ordered Resolution:

$$\frac{\alpha_1 \parallel \Gamma_1 \rightarrow \Delta_1, A_1 \quad \alpha_2 \parallel \Gamma_2, A_2 \rightarrow \Delta_2}{(\alpha_1, \alpha_2^{\not\simeq} \parallel \Gamma_1, \Gamma_2 \rightarrow \Delta_1, \Delta_2)\sigma_1\sigma_2}$$

where (i) $\sigma_1 = \mathrm{mgu}(A_1, A_2)$, $\sigma_2 = \mathrm{mgu}(\alpha_1^{\simeq}\sigma_1, \alpha_2^{\simeq}\sigma_1)$,[5] (ii) no atom is selected in $\Gamma_1 \rightarrow \Delta_1, A_1$ and $A_1\sigma_1\sigma_2$ strictly maximal in $(\Gamma_1 \rightarrow \Delta_1, A_1)\sigma_1\sigma_2$, and (iii) either A_2 is selected in $\Gamma_2, A_2 \rightarrow \Delta_2$, or no atom occurrence is selected in $\Gamma_2, A_2 \rightarrow \Delta_2$ and $A_2\sigma_1\sigma_2$ is maximal in $(\Gamma_2, A_2 \rightarrow \Delta_2)\sigma_1\sigma_2$.

Ordered Factoring:

$$\frac{\alpha \parallel \Gamma \rightarrow \Delta, A, A'}{(\alpha \parallel \Gamma \rightarrow \Delta, A)\sigma}$$

where (i) $\sigma = \mathrm{mgu}(A, A')$ and (ii) no occurrence is selected in $\Gamma \rightarrow \Delta, A, A'$ and $A\sigma$ is maximal in $(\Gamma \rightarrow \Delta, A, A')\sigma$.

Where not explicitly stated otherwise, we assume a selection function that selects no occurrences at all.

We will now show that we can decide whether a finite constrained clause set N that is saturated by COR has a Herbrand model over Σ and, if so, how to construct such a model. As constrained clauses are an extension of unconstrained clauses, the construction of a Herbrand model of N is strongly related to the one from [1] for unconstrained clause sets as recalled in Section 2. The main difference is that we now have to account for constraints before starting the construction. To define a Herbrand interpretation $N_\mathcal{I}$ of a set N of constrained clauses over Σ, we proceed in two steps:

(1) Let $V = \{v_1, \ldots, v_n\}$ and let $A_N = \{\alpha \mid (\alpha \parallel \Box) \in N\}$ be the set of all constraints of constrained clauses in N with empty clausal part. A_N is *covering*

[5] We unify $\alpha_1^{\simeq}$ and $\alpha_2^{\simeq}$ for efficiency reasons: Since equations are syntactic, if $\alpha_1^{\simeq}$ and $\alpha_2^{\simeq}$ are not unifiable then any variable-free instance of α_1, α_2 is unsatisfiable anyway.

(for Σ) if for every positive ground constraint $\beta = \vec{v} \simeq \vec{t}$ over Σ there is a satisfiable ground instance α of a constraint in A_N such that $\beta = \alpha^{\simeq}$.

We define a constraint α_N as follows: If A_N is not covering, let $\alpha_N = \vec{v} \simeq \vec{t}$ be a positive ground constraint that is not equal to the equational part of any satisfiable ground instance of a constraint in A_N.[6] If A_N is covering, let α_N be an arbitrary ground constraint. We will show that N is Herbrand-unsatisfiable over Σ in this case.

(2) $N_{\mathcal{I}}$ is defined as the Herbrand interpretation $N'_{\mathcal{I}}$ associated to the (unconstrained) ground clause set

$$N' = \{C\sigma \mid (\alpha \parallel C) \in N \text{ and } \sigma : \mathrm{vars}(\alpha \parallel C) \setminus V \to \mathcal{T}(\mathcal{F}) \\ \text{and } \alpha_N \sigma = \alpha^{\simeq}\sigma \text{ and } \alpha\sigma \text{ is satisfiable}\} \ .$$

As an example consider the signature $\Sigma = (\emptyset, \{0, s\})$, $V = \{v\}$ and the two constrained clause sets $M = \{v \simeq 0 \parallel \Box,\ v \simeq s(0) \parallel \Box,\ v \simeq s(s(0)) \parallel \Box,\ \ldots\}$ and $N = \{v \simeq s(s(x)) \parallel \Box\}$. Then A_M is covering but A_N is not, and we may choose either $\{v \simeq 0\}$ or $\{v \simeq s(0)\}$ for α_N.

One easily sees that $N_{\mathcal{I}}$ is independent of α_N if all clauses in N are unconstrained. If moreover $N_{\mathcal{I}} \models N$, then $N_{\mathcal{I}}$ one of the minimal models of N wrt. set inclusion and we call $N_{\mathcal{I}}$ *the minimal model* of N.

While it is well known how the second step in the construction of $N_{\mathcal{I}}$ works, it is not obvious that one can decide whether A_N is covering and, if it is not, effectively compute some α_N. This is, however, possible for finite A_N: Let $\{x_1, \ldots, x_m\} \subseteq X$ be the set of universal variables appearing in A_N. A_N is covering if and only if the formula $\forall x_1, \ldots, x_m . \bigwedge_{\alpha \in A_N} \neg\alpha$ is satisfiable in $\mathcal{T}(\mathcal{F})$. Such so-called disunification problems have been studied among others by Comon and Lescanne [5], who gave a terminating algorithm that eliminates the universal quantifiers from this formula and transforms the initial problem into an equivalent formula from which the set of solutions can easily be read off.

We will now show that Herbrand-satisfiability or unsatisfiability over Σ is invariant under the application of inferences in COR and that a saturated constrained clause set N has a Herbrand model over Σ (namely $N_{\mathcal{I}}$) if and only if A_N is not covering.

Proposition 1 (Soundness). *Let $\alpha \parallel C$ be the conclusion of a COR inference with premises in a set N of constrained clauses over Σ. Then $N \models_\Sigma N \cup \{\alpha \parallel C\}$.*

As usual, the fairness of a derivation can be ensured by systematically adding conclusions of non-redundant inferences, making these inferences redundant.

The following theorem relies on soundness, redundancy, and fairness rather than on a concrete inference system. Hence its proof is exactly as in the unconstrained case or in the case of the original fixed domain calculus (cf. [1,12]):

Proposition 2 (Saturation). *Let $N_0, N_1, \ldots$ be a fair COR derivation. Then the set $N^* = \bigcup_j \bigcap_{k \geq j} N_k$ is saturated. N_0 has a Herbrand model over Σ if and only if N^* does.*

[6] In contrast to [12], we do not have to impose any minimality requirements on α_N.

Now we express the Herbrand-satisfiability of N^* over Σ in terms of the coverage of A_{N^*}.

Proposition 3. *Let N be a set of constrained clauses such that A_N is covering. Then N does not have any Herbrand model over Σ.*

Proof. Let $\mathcal{M}$ be a Herbrand model of N over $\Sigma = (\mathcal{P}, \mathcal{F})$. Then there is a substitution $\sigma : V \to \mathcal{T}(\mathcal{F})$ such that for every constrained clause $\alpha \parallel C \in N$ and every substitution $\tau : \mathrm{vars}(\alpha \parallel C) \setminus V \to \mathcal{T}(\mathcal{F})$, if $\alpha\sigma\tau$ is satisfiable then $\mathcal{M} \models C\sigma\tau$. Then the same holds for all constrained clauses $\alpha \parallel \Box \in N$. Since $\mathcal{M} \not\models \Box$, this means that for all such constrained clauses and all τ, $\alpha\sigma\tau$ is not satisfiable, and so $\alpha\sigma$ is not satisfiable. Since every positive ground constraint over Σ is of the form $v_1 \simeq v_1\sigma, \ldots, v_n \simeq v_n\sigma$ for some substitution $\sigma : V \to \mathcal{T}(\mathcal{F})$, this means that A_N is not covering. ■

Proposition 4 (Σ-Completeness for Saturated Clause Sets). *Let N be a saturated set of constrained clauses over Σ such that A_N is not covering for Σ. Then $N_{\mathcal{I}} \models N$ for any choice of α_N.*

Proof. Let $\alpha_N = v_1 \simeq t_1, \ldots, v_n \simeq t_n$ and assume, contrary to the proposition, that $N_{\mathcal{I}}$ is not a model of N. Then there are $\alpha \parallel C \in N$ and $\sigma : \mathrm{vars}(\alpha \parallel C) \to \mathcal{T}(\mathcal{F})$ such that $\sigma(v_i) = t_i$ for all i, $\alpha\sigma$ is satisfiable and $N_{\mathcal{I}} \not\models C\sigma$. Let $C\sigma$ be minimal with these properties. We will refute this minimality. We proceed by a case analysis of the position of selected or maximal literal occurrences in $C\sigma$.

- $C\sigma$ does not contain any literal at all, i.e. $C = \Box$. Then the satisfiability of $\alpha\sigma$ contradicts the choice of α_N.
- $C = \Gamma, A \to \Delta$ and $A\sigma$ is selected or $A\sigma$ is maximal and no literal is selected in $C\sigma$. Since $N_{\mathcal{I}} \not\models C\sigma$, we know that $A\sigma \in N_{\mathcal{I}}$. The literal A must be produced by a ground instance $(\beta \parallel \Lambda \to \Pi, B)\sigma'$ of a constrained clause in N in which no literal is selected. Note that both ground constrained clauses $(\alpha \parallel C)\sigma$ and $(\beta \parallel \Lambda \to \Pi, B)\sigma'$ are not redundant with respect to N.

  Because $\alpha^{\simeq}\sigma = \beta^{\simeq}\sigma' = \alpha_N\sigma$ and because σ is a unifier of A and B, i.e. an instance of $\sigma_1 := \mathrm{mgu}(A, B)$, there is an inference by ordered resolution as follows:

$$\frac{\beta \parallel \Lambda \to \Pi, B \quad \alpha \parallel \Gamma, A \to \Delta}{(\alpha, \beta^{\not\simeq} \parallel \Lambda, \Gamma \to \Pi, \Delta)\sigma_1\sigma_2}\ \sigma_2 = \mathrm{mgu}(\beta^{\simeq}\sigma_1, \alpha^{\simeq}\sigma_1)$$

  Looking at the ground instance $\delta \parallel D = (\alpha, \beta^{\not\simeq} \parallel \Lambda, \Gamma \to \Pi, \Delta)\sigma$ of the conclusion, we see that δ is satisfiable and $N_{\mathcal{I}} \not\models D$.

  On the other hand, as the inference is redundant, so is the constrained clause $\delta \parallel D$, i.e. D follows from ground instances $\delta \parallel C_i$ of constrained clauses of N all of which are smaller than $(\alpha \parallel C)\sigma$. Because of the minimality of $C\sigma$, all C_i hold in $N_{\mathcal{I}}$. So $N_{\mathcal{I}} \models D$, which contradicts $N_{\mathcal{I}} \not\models D$.
- $C = \Gamma \to \Delta, A$ and $A\sigma$ is strictly maximal in $C\sigma$. This is not possible, since then either $C\sigma$ or a smaller clause must have produced $A\sigma$, and hence $N_{\mathcal{I}} \models C\sigma$, which contradicts the choice of $C\sigma$.

– No literal in $C = \Gamma \to \Delta, A$ is selected and $A\sigma$ is maximal but not strictly maximal in $C\sigma$. Then $\Delta = \Delta', A'$ such that $A'\sigma = A\sigma$. So there is an inference by ordered factoring as follows:

$$\frac{\alpha \parallel \Gamma \to \Delta', A, A'}{(\alpha \parallel \Gamma \to \Delta', A')\sigma_1} \; \sigma_1 = \mathrm{mgu}(s, s')$$

As above, $\alpha\sigma$ is satisfiable and we can derive both $N_{\mathcal{I}} \models (\Gamma \to \Delta', A')\sigma$ and $N \not\models (\Gamma \to \Delta', A')\sigma$, which is a contradiction. ■

Combining Propositions 2, 3 and 4, we can conclude:

Theorem 1. *Let $N_0, N_1, \ldots$ be a fair COR derivation. Then $N^* = \bigcup_j \bigcap_{k \geq j} N_k$ is saturated. Moreover, N_0 has a Herbrand model over Σ if and only if A_{N^*} is not covering.*

4 Decidability of Model Equivalence and Formula Entailment for DIGs

In Section 3, we showed how saturated sets of constrained clauses can be regarded as (implicitly) representing certain Herbrand models. Other representations of Herbrand models include sets of non-ground atoms or the more flexible so-called disjunctions of implicit generalizations of Lassez and Marriott [15]. We show how both types of representation can be seen as special cases of the representation by saturated constrained clause sets.

Based on this view, we reprove that the equivalence of any given pair of representations by disjunctions of implicit generalizations is decidable, and we extend the known results on the decidability of clause and formula entailment (cf. [9]). To do so, we translate a query $\mathcal{M} \models \phi$ into a constrained clause set that is Herbrand-unsatisfiable over a certain signature iff $\mathcal{M} \models \phi$ holds. The Herbrand-unsatisfiability can then be decided using the calculus COR.

Definition 1. *An* implicit generalization *G over Σ is an expression of the form $A/\{A_1, \ldots, A_n\}$, where $A, A_1, \ldots, A_n$ are atoms over Σ. A set D of implicit generalizations over Σ is called a* DIG (disjunction of implicit generalizations)*. A DIG D is called an* atomic representation of a term model (ARM)*, if all implicit generalizations in D are of the form $A/\{\}$.*

The Herbrand model $\mathcal{M}(\{A/\{A_1, \ldots, A_n\}\})$ represented by a DIG consisting of a single implicit generalization $A/\{A_1, \ldots, A_n\}$ is exactly the set of all atoms that are instances of the atom A but not of any A_i. The model $\mathcal{M}(D)$ represented by a general DIG $D = \{G_1, \ldots, G_m\}$ is the union of the models $\mathcal{M}(\{G_1\}), \ldots, \mathcal{M}(\{G_m\})$.

Example 1. Let $D = \{G_1, G_2\}$ be a DIG over $\Sigma = (\{P\}, \{s, 0\})$, where 0 is a constant, s is a unary function symbol, and the two implicit generalizations in D are $G_1 = P(s(x), s(y))/\{P(x, x)\}$ and $G_2 = P(0, y)/\{P(x, 0)\}$. The model represented by D is $\mathcal{M}(D) = \{P(t, t') \mid t, t' \in \mathcal{T}(\{s, 0\})$ and $t \neq t'$ and $t' \neq 0\}$.

Note that, without loss of generality, we may assume for each implicit generalization $G = A/\{A_1, \ldots, A_n\}$ that all atoms $A_1, \ldots, A_n$ are instances of A. If A

is of the form $P(\vec{t})$, then we say that G is an implicit generalization *over* P. If $G_1, \ldots, G_n$ are implicit generalizations over $P_1, \ldots, P_n$, respectively, we say that $\{G_1, \ldots, G_n\}$ is a *DIG over* $\{P_1, \ldots, P_n\}$.

We will now translate each DIG D into a set $R_0(D)$ of constrained clauses whose minimal model is $\mathcal{M}(D)$. The set $R_0(D)$ may also have other Herbrand models, which means that in general $\models_{Ind}$ and $\models_{\Sigma}$ do not agree for $R_0(D)$. Hence the calculus COR is not complete for $\mathcal{M}(D)$ based on $R_0(D)$ alone. We use a predicate completion procedure that was proposed by Comon and Nieuwenhuis [6] to enrich $R_0(D)$ by additional constrained clauses, such that the resulting clause set $R(D)$ has exactly one Herbrand model over the given signature.

In general, the completion procedure works as follows:[7]

(1) Let P be a predicate and let N_P be a finite and saturated set of clauses over $\Sigma = (\mathcal{P}, \mathcal{F})$ such that all clauses in N are of the form $\Gamma \to \Delta, P(\vec{t})$, where $P(\vec{t})$ is a strictly maximal literal occurrence. Combine all these clauses into a single formula $\phi_P \to P(\vec{x})$ where

$$\phi_P = \exists \vec{y}. \bigvee_{\Gamma \to \Delta, P(\vec{t}) \in N_P} (\vec{x} \simeq \vec{t} \wedge \bigwedge_{A \in \Gamma} A \wedge \bigwedge_{B \in \Delta} \neg B) \ ,$$

the y_i are the variables appearing in N_P, and the x_j are fresh variables.

(2) In the minimal model $(N_P)_{\mathcal{I}}$, this formula is equivalent to $\neg\phi_P \to \neg P(\vec{x})$. If all variables appearing in $\Gamma \to \Delta, P(\vec{t})$ also appear in $P(\vec{t})$, $\neg\phi_P$ can be transformed using quantifier elimination [5] into an equivalent formula ψ that does not contain any universal quantifiers. This quantifier elimination procedure is the same that we used in Section 3 to decide the coverage of constraint sets.

(3) This formula can in turn be written as a set N'_P of constrained clauses. The union $N_P \cup N'_P$ is the *completion* of N_P.

(4) If N is the union of several sets as in (1) defining different predicates, then the completion of N is the union of N and all N'_P, $P \in \mathcal{P}$, where N_P is the set of all clauses in N having a strictly maximal literal $P(\vec{t})$.

Comon and Nieuwenhuis showed in [6, Lemma 47] that the minimal model of a Herbrand-satisfiable unconstrained saturated clause set is also a model of its completion. Moreover, the union of the original set and its conclusion has at most one Herbrand-model over the given signature.[8]

Definition 2. *For each DIG D, we define constrained clause sets $R_0(D)$ and $R(D)$ as follows. If $D = \{G_1, \ldots, G_n\}$ is a DIG over $\{P\}$, let $\dot{P}_1, \check{P}_1, \ldots \dot{P}_n, \check{P}_n$ be fresh predicates. For each G_i, the predicate $\dot{P}_i$ will describe the left hand side of G_i and serve as an over-approximation of P, and $\check{P}_i$ describes the right hand*

[7] For extended examples confer Example 2 later on.

[8] In fact, the result by Comon and Nieuwenhuis requires that the input to the completion procedure is a Horn clause set. However, the proof is identical for clause sets like $R_0(D)$ in which each clause contains a unique strictly maximal positive literal.

side. For each implicit generalization $G_i = P(\vec{s})/\{P(\vec{s}_1), \ldots, P(\vec{s}_n)\}$*, define an auxiliary clause set* $R_0(G_i) = \{\rightarrow \dot{P}_i(\vec{s}),\ \rightarrow \check{P}_i(\vec{s}_1),\ \ldots,\ \rightarrow \check{P}_i(\vec{s}_n)\}$*. Then*

$$R_0(D) = \bigcup_{1 \leq i \leq n} R_0(G_i) \cup \{\dot{P}_i(\vec{x}) \rightarrow \check{P}_i(\vec{x}), P(\vec{x})\} \ .$$

If $D = D_1 \cup \ldots \cup D_m$ *such that each* D_i *is a DIG over a single predicate and* D_i *and* D_j *are DIGs over different predicates whenever* $i \neq j$*, let*

$$R_0(D) = R_0(D_1) \cup \ldots \cup R_0(D_m) \ .$$

We assume that fresh predicates are smaller wrt. $\prec$ *than signature predicates and that* $\dot{P}_i \prec \check{Q}_j$ *for all fresh predicates* $\dot{P}_i$ *and* $\check{Q}_j$*. Finally,* $R(D)$ *is defined as the Comon-Nieuwenhuis completion of* $R_0(D)$*.*

Note that each clause in $R_0(D)$ has a unique strictly maximal literal that is positive. Hence $R_0(D)$ is saturated with respect to COR (with an empty selection function) and $(R_0(D))_{\mathcal{I}}$ is a minimal Herbrand model of $R_0(D)$ over the extended signature. Moreover, the completion procedure is really applicable to $R_0(D)$ and the results of [6, Lemma 47] mentioned above apply here:

Lemma 1. *Let* D *be a DIG over* $\Sigma = (\mathcal{P}, \mathcal{F})$ *and let* $\mathcal{P}'$ *be the set of fresh predicates in* $R_0(D)$*. Then* $R(D)$ *has exactly one Herbrand model over* $\Sigma' = (\mathcal{P} \cup \mathcal{P}', \mathcal{F})$*, namely* $(R_0(D))_{\mathcal{I}}$*.*

Hence for every formula ϕ *over* Σ'*, it holds that* $R(D) \models_{Ind} \phi'$ *iff* $R(D) \models_{\Sigma} \phi'$ *iff* $R(D) \cup \{\neg\phi\}$ *is Herbrand-unsatisfiable over* Σ'*.*

Proposition 5 (Equivalence of D **and** $R(D)$**).** *Let* D *be a DIG and let* $P(\vec{t})$ *be a ground atom over* Σ*. Then* $\mathcal{M}(D) \models P(\vec{t})$ *iff* $R(D) \models_{\Sigma} P(\vec{t})$*.*

Proof. Let $D = \{G_1, \ldots, G_m\}$. $\mathcal{M}(D) \models P(\vec{t})$ holds iff there is a G_i such that $\mathcal{M}(\{G_i\}) \models P(\vec{t})$. If we write $G_i = P(\vec{s})/\{P(\vec{s}_1), \ldots, P(\vec{s}_n)\}$, then this is equivalent to $P(\vec{t})$ being an instance of $P(\vec{s})$ but not of any $P(\vec{s}_j)$. This is turn is equivalent to $R_0(\{G_i\}) \models_{Ind} \dot{P}_i(\vec{t})$ and $R_0(\{G_i\}) \not\models_{Ind} \check{P}_i(\vec{t})$. That this holds for some i is equivalent to $R_0(D) \models_{Ind} P(\vec{t})$, or by Lemma 1 to $R(D) \models_{\Sigma} P(\vec{t})$. ■

Let us investigate the shape of the constrained clauses of $R(D)$ more closely. Consider first an implicit generalization $G = P(\vec{s})/\{P(\vec{s}_1), \ldots, P(\vec{s}_n)\}$. All constrained clauses in $R_0(G)$ are unconstrained units. The completion procedure adds constrained unit clauses of the following forms:

$$\alpha^{\not\approx} \parallel \dot{P}(\vec{t}) \rightarrow$$
$$\alpha^{\not\approx} \parallel \check{P}(\vec{t}) \rightarrow$$

For a DIG D, $R_0(D)$ contains, in addition to clauses as just presented, only clauses of the form $\dot{P}(\vec{x}) \rightarrow \check{P}(\vec{x}), P(\vec{x})$, where $P(\vec{x})$ is the maximal literal occurrence. The only additional constrained clauses in $R(D)$ come from the completion of signature predicates and are of the form

$$P(\vec{x}), \check{P}_{j_1}(\vec{x}), \ldots, \check{P}_{j_m}(\vec{x}) \rightarrow \dot{P}_{j_{m+1}}(\vec{x}), \ldots, \dot{P}_{j_n}(\vec{x}) \ ,$$

where $\dot{P}_{j_1}, \check{P}_{j_1}, \ldots, \dot{P}_{j_n}, \check{P}_{j_n}$ are the fresh predicates introduced for P. Note that all constrained non-unit clauses contain a unique literal that is maximal for all instances of the constrained clause, namely $P(\vec{x})$.

Example 2. Consider the DIG D from Example 1. The sets $R_0(G_1)$ and $R_0(G_2)$ consist of the following unconstrained clauses:

$$\begin{aligned} R_0(G_1) &= \{\rightarrow \dot{P}_1(s(x), s(y)),\ \rightarrow \check{P}_1(x, x)\} \\ R_0(G_2) &= \{\rightarrow \dot{P}_2(0, y),\ \rightarrow \check{P}_2(x, 0)\} \end{aligned}$$

$R_0(D)$ additionally contains the unconstrained clauses

$$\begin{aligned} &\dot{P}_1(x, y) \rightarrow \check{P}_1(x, y), P(x, y) \text{ and} \\ &\dot{P}_2(x, y) \rightarrow \check{P}_2(x, y), P(x, y)\ . \end{aligned}$$

To compute $R(D)$, we have to look at the sets of clauses defining the predicates $\dot{P}_1, \dot{P}_2, \check{P}_1, \check{P}_2$, and P:

$$\begin{aligned} N_{\dot{P}_1} &= \{\rightarrow \dot{P}_1(s(x), s(y))\} & N_{\check{P}_1} &= \{\rightarrow \check{P}_1(x, x)\} \\ N_{\dot{P}_2} &= \{\rightarrow \dot{P}_2(0, y)\} & N_{\check{P}_2} &= \{\rightarrow \check{P}_2(x, 0)\} \\ N_P &= \{\dot{P}_1(x, y) \rightarrow \check{P}_1(x, y), P(x, y)\ , \quad \dot{P}_2(x, y) \rightarrow \check{P}_2(x, y), P(x, y)\}\ . \end{aligned}$$

The negation of $\dot{P}_1$ in the minimal model of $R_0(D)$ is obviously defined by $\neg\dot{P}_1(x, y) \iff \neg\exists x', y'.x \simeq s(x') \wedge y \simeq s(y')$. The quantifier elimination procedure simplifies the right hand side to $x \simeq 0 \vee y \simeq 0$. This results in the unconstrained completion

$$N'_{\dot{P}_1} = \{\dot{P}_1(0, y) \rightarrow,\ \dot{P}_1(x, 0) \rightarrow\}\ .$$

Analogously, the negation of $\dot{P}_1$ in the minimal model of $R_0(D)$ is defined by $\neg\dot{P}_1(x, y) \iff x \not\simeq y$. The corresponding completion is not unconstrained:

$$N'_{\check{P}_1} = \{x \not\simeq y \,\|\, \check{P}_1(x, y) \rightarrow\}$$

Whenever the maximal literal of some input clause is non-linear, i.e. whenever a variable appears twice in this literal, then such constraints consisting of disequations always appear. E.g. the completion of $\{\rightarrow Q(x, x, x)\}$ adds the three clauses $x \not\simeq y \,\|\, Q(x, y, z) \rightarrow$, $x \not\simeq z \,\|\, Q(x, y, z) \rightarrow$, and $y \not\simeq z \,\|\, Q(x, y, z) \rightarrow$. Such non-linearities are also the only reason for the appearance of constraint disequations.

The completions of $\dot{P}_2$ and $\check{P}_2$ are computed analogously as

$$\begin{aligned} N'_{\dot{P}_2} &= \{\dot{P}_2(s(x), y) \rightarrow\} \text{ and} \\ N'_{\check{P}_2} &= \{\check{P}_2(x, s(y)) \rightarrow\}\ . \end{aligned}$$

For P, we have that $\neg P(x, y) \iff (\neg\dot{P}_1(x, y) \vee \check{P}_1(x, y)) \wedge (\neg\dot{P}_2(x, y) \vee \check{P}_2(x, y))$. Rewriting the right hand side to its disjunctive normal form allows to translate this definition into the following clause set:

$$\begin{aligned} N'_P = \{&P(x,y), \check{P}_1(x,y), \check{P}_2(x,y) \rightarrow , \\ &P(x,y), \check{P}_1(x,y) \rightarrow \dot{P}_2(x,y) \ , \\ &P(x,y), \check{P}_2(x,y) \rightarrow \dot{P}_1(x,y) \ , \\ &P(x,y) \rightarrow \dot{P}_1(x,y), \dot{P}_2(x,y)\} \end{aligned}$$

The set $R(D)$ is then the union of $R_0(D)$ and all N'_Q, $Q \in \{\dot{P}_1, \dot{P}_2, \check{P}_1, \check{P}_2, P\}$. ■

Lemma 2. *Let D be a DIG over $\Sigma = (\mathcal{P}, \mathcal{F})$ and let $\mathcal{P}'$ be the set of fresh predicates in $R(D)$. Then the class of constrained clauses over $(\mathcal{P}', \mathcal{F})$ of the following forms is closed under the inference rules of COR:*
(1) $\alpha \parallel C$ where C contains at most one literal
(2) $\alpha \parallel \dot{P}_i(\vec{t}) \rightarrow \check{P}_i(\vec{t})$
(3) $\alpha \parallel \check{P}_{i_1}(\vec{t}), \ldots, \check{P}_{i_k}(\vec{t}), \dot{P}_{i_{k+1}}(\vec{t}), \ldots, \dot{P}_{i_l}(\vec{t}) \rightarrow \dot{P}_{i_{l+1}}(\vec{t}), \ldots, \dot{P}_{i_m}(\vec{t})$ where $0 \leq k \leq l \leq m$.
Moreover, the saturation of a finite set of such constrained clauses with COR terminates.

Lemma 3. *Let D be a DIG over $\Sigma = (\mathcal{P}, \mathcal{F})$ and let $\mathcal{P}'$ be the set of fresh predicates in $R(D)$. If N is a set of constrained clauses over Σ containing at most one literal each, then it is decidable whether $R(D) \cup N$ is Herbrand-satisfiable over $\Sigma' = (\mathcal{P} \cup \mathcal{P}', \mathcal{F})$.*

Proof. Let M^* be a saturation of $M = R(D) \cup N$ by the calculus COR. By Theorem 1, Herbrand-unsatisfiability of M over Σ' is equivalent to the coverage of A_{M^*}, which is decidable if M^* is finite.

To prove that M^* is finite, we show that any derivation starting from M is finite. The only constrained clauses containing at least two literals and a predicate symbol of $\mathcal{P}$ are of the form $\dot{P}_i(\vec{x}) \rightarrow P(\vec{x}), \check{P}_i(\vec{x})$ or $P(\vec{x}), \check{P}_{j_1}(\vec{x}), \ldots, \check{P}_{j_m}(\vec{x}) \rightarrow \dot{P}_{j_{m+1}}(\vec{x}), \ldots, \dot{P}_{j_n}(\vec{x})$, where $P \in \mathcal{P}$, $\dot{P}_{j_1}, \check{P}_{j_n}, \ldots, \dot{P}_{j_n}, \check{P}_{j_n} \in \mathcal{P}'$ are the fresh predicates introduced for P, and $P(\vec{x})$ is the maximal literal occurrence in both types of constrained clauses (cf. the remarks following Lemma 1). Since each inference between constrained clauses containing predicate symbols of $\mathcal{P}$ reduces the number of atoms featuring such a predicate, there are only finitely many such inferences. The conclusion of an inference

$$\frac{\parallel \dot{P}_i(\vec{x}) \rightarrow P(\vec{x}), \check{P}_i(\vec{x}) \qquad \parallel P(\vec{x}), \check{P}_{j_1}(\vec{x}), \ldots, \check{P}_{j_m}(\vec{x}) \rightarrow \dot{P}_{j_{m+1}}(\vec{x}), \ldots, \dot{P}_{j_n}(\vec{x})}{\parallel \Gamma \rightarrow \Delta}$$

between two constrained clauses in $R(D)$ using $P \in \mathcal{P}$ is a tautology (and thus redundant), because either $\dot{P}_i(\vec{x})$ or $\check{P}_i(\vec{x})$ appears in both Γ and Δ. The remaining derivable constrained clauses over $(\mathcal{P}', \Sigma)$ obey the restrictions of Lemma 2, hence the saturation terminates.

With this preliminary work done, we can decide whether two DIGs represent the same model:

Theorem 2 (DIG Equivalence). *Equivalence of DIGs is decidable by saturation with respect to COR.*

Proof. Let D, D' be two DIGs. Because $\mathcal{M}(D) = \bigcup_{G \in D} \mathcal{M}(\{G\})$, and because $\mathcal{M}(D) = \mathcal{M}(D')$ iff $\mathcal{M}(D) \subseteq \mathcal{M}(D')$ and $\mathcal{M}(D') \subseteq \mathcal{M}(D)$, it suffices to show the decidability of $\mathcal{M}(D) \subseteq \mathcal{M}(D')$ in the case where D consists of a single implicit generalization $G = P(\vec{s})/\{P(\vec{s}\sigma_1), \ldots, P(\vec{s}\sigma_n)\}$. Without loss of generality, we assume that $P(\vec{s})$ and $P(\vec{s}\sigma_1), \ldots, P(\vec{s}\sigma_n)$ do not share any variables.

Let $x_1, \ldots, x_m$ be the variables in $P(\vec{s})$ and let $y_1, \ldots, y_k$ be the variables in $P(\vec{s}\sigma_1), \ldots, P(\vec{s}\sigma_n)$. By Proposition 5, $\mathcal{M}(D) \subseteq \mathcal{M}(D')$ holds iff $R(D') \models_\Sigma P(\vec{t})$ for every atom $P(\vec{t}) \in \mathcal{M}(D)$. Equivalently, $R(D') \cup \{\exists \vec{x}.\forall y.\vec{x} \not\simeq \vec{x}\sigma_1 \wedge \ldots \wedge \vec{x} \not\simeq \vec{x}\sigma_n \wedge \neg P(\vec{s})\}$ does not have a Herbrand model over Σ, i.e. the same holds for the constrained clause set $R(D') \cup \{\vec{v} \simeq \vec{x} \parallel P(s) \rightarrow,\ \vec{v} \simeq \vec{x}\sigma_1 \parallel \square,\ \ldots,\ \vec{v} \simeq \vec{x}\sigma_n \parallel \square\}$.

By Lemma 3, this is decidable by means of the calculus COR. ■

Example 3. The DIG $D' = \{P(x, s(y))/P(s(x'), s(x'))\}$ and the DIG D from Examples 1 and 2 describe the same model. We only show that $\mathcal{M}(D) \supseteq \mathcal{M}(D')$.

Expressed as a satisfiability problem of constrained clauses, we have to check whether $R(D) \cup \{v_1 \simeq x, v_2 \simeq y \parallel P(x, s(y)) \rightarrow,\ v_1 \simeq s(x'), v_2 \simeq x' \parallel \square\}$ is Herbrand-satisfiable over Σ. To do so, we saturate this set with respect to COR.

Since $R(D) \cup \{v_1 \simeq s(x'), v_2 \simeq x' \parallel \square\}$ is saturated, all non-redundant inferences use at least one descendant of $v_1 \simeq x, v_2 \simeq y \parallel P(x, s(y)) \rightarrow$. The following constrained clauses can be derived. We index the new constrained clauses by $(0) \ldots (9)$. Each of these constrained clauses is derived from one clause in $R(D)$ (which is not given here) and another clause that is indicated by its index:

(0)	$v_1 \simeq s(x'), v_2 \simeq x'$	$\parallel$	$\square$	
(1)	$v_1 \simeq x, v_2 \simeq y$	$\parallel$ $P(x, s(y)) \rightarrow$		
(2)	$v_1 \simeq x, v_2 \simeq y$	$\parallel \dot{P}_1(x, s(y)) \rightarrow \check{P}_1(x, s(y))$		derived from (1)
(3)	$v_1 \simeq x, v_2 \simeq y$	$\parallel \dot{P}_2(x, s(y)) \rightarrow \check{P}_2(x, s(y))$		derived from (1)
(4)	$v_1 \simeq s(x), v_2 \simeq y$	$\parallel$	$\rightarrow \check{P}_1(s(x), s(y))$	derived from (2)
(5)	$v_1 \simeq s(x), v_2 \simeq y, x \not\simeq s(y)$	$\parallel \dot{P}_1(x, s(y)) \rightarrow$		derived from (2)
(6)	$v_1 \simeq x, v_2 \simeq y, s(x) \not\simeq s(y)$	$\parallel$	$\square$	derived from (4) or (5)
(7)	$v_1 \simeq 0, v_2 \simeq y$	$\parallel$	$\rightarrow \check{P}_2(0, s(y))$	derived from (3)
(8)	$v_1 \simeq x, v_2 \simeq y$	$\parallel \dot{P}_2(x, s(y)) \rightarrow$		derived from (3)
(9)	$v_1 \simeq 0, v_2 \simeq y$	$\parallel$	$\square$	derived from (7) or (8)

No further non-redundant constrained clauses can be derived. The constraint set $\{(v_1 \simeq s(x'), v_2 \simeq x'), (v_1 \simeq s(x), v_2 \simeq y, s(x) \not\simeq s(y)), (v_1 \simeq 0, v_2 \simeq y)\}$ consisting of the constraints of the constrained clauses (0), (6), and (9) is covering, which means that the whole constrained clause set is Herbrand-unsatisfiable over Σ, i.e. that $\mathcal{M}(D) \supseteq \mathcal{M}(D')$. ■

Apart from deciding equivalence of DIGs, we can decide for formulas from a number of classes whether they are true in models represented by DIGs.

Theorem 3 (Decidability of DIG Formula Entailment). *Let D be a DIG and let ϕ be a quantifier-free formula over Σ with variables $\vec{x}, \vec{y}$. The following problems are decidable:*

(1) $\mathcal{M}(D) \models \forall \vec{x}.\exists \vec{y}.\phi$ is decidable if one of the following holds:

(a) ϕ *is a single clause*
(b) ϕ *is a conjunction of clauses of the form* $\rightarrow \Delta$
(c) ϕ *is a conjunction of clauses of the form* $\Gamma \rightarrow$
(d) ϕ *is a conjunction of unit clauses where no predicate appears in both a positive and a negative literal*

(2) $\mathcal{M}(D) \models \exists\vec{x}.\forall\vec{y}.\phi$ *is decidable if one of the following holds:*
(a) ϕ *is a conjunction of unit clauses*
(b) ϕ *is a conjunction of clauses of the form* $\Gamma \rightarrow$
(c) ϕ *is a conjunction of clauses of the form* $\rightarrow \Delta$
(d) ϕ *is a clause where no predicate appears in both a positive and a negative literal*

Proof. We first consider the case (1a). Let $\Sigma = (\mathcal{P}, \mathcal{F})$ and let $\mathcal{P}'$ be the set of fresh predicates in $R(D)$. Let $C = A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$ and let $N = \{\vec{v}\simeq\vec{x} \,\|\, \rightarrow A_1,\ \ldots,\ \vec{v}\simeq\vec{x} \,\|\, \rightarrow A_n,\ \vec{v}\simeq\vec{x} \,\|\, B_1 \rightarrow,\ \ldots,\ \vec{v}\simeq\vec{x} \,\|\, B_n \rightarrow\}$. By Proposition 5, $\mathcal{M}(D) \models \forall\vec{x}.\exists\vec{y}.C$ is equivalent to $R(D) \models_{Ind} \forall\vec{x}.\exists\vec{y}.C$. This in turn is equivalent to the Herbrand-unsatisfiability of $R(D) \cup \{\exists\vec{x}.\forall\vec{y}.\neg C\}$, or equivalently $R(D) \cup N$, over $(\mathcal{P} \cup \mathcal{P}', \mathcal{F})$. By Lemma 3, the latter is decidable.

The proofs for (1b)–(1d) are exactly analogous, using slight variations of Lemma 3. The decidability of the problems (2a)–(2d) reduces to (1a)–(1d), respectively, because $\mathcal{M}(D) \models \exists\vec{x}.\forall\vec{y}.\phi$ if and only if $\mathcal{M}(D) \not\models \forall\vec{x}.\exists\vec{y}.\neg\phi$. ■

The simple nature of atomic representations allows us to go one step further:

Theorem 4 (Decidability of ARM Formula Entailment). *Let D be an ARM over Σ and let ϕ be a quantifier-free formula over Σ with variables $\vec{x}, \vec{y}$. It is decidable whether $\mathcal{M}(D) \models \forall\vec{x}.\exists\vec{y}.\phi$ and whether $\mathcal{M}(D) \models \exists\vec{x}.\forall\vec{y}.\phi$.*

Proof. We first show the decidability of $\mathcal{M}(D) \models \forall\vec{x}.\exists\vec{y}.\phi$. Let $\Sigma = (\mathcal{P}, \mathcal{F})$ and let $\mathcal{P}'$ be the set of fresh predicates in $R(D)$. We write $\neg\phi$ as an equivalent finite set $N_{\neg\phi}$ of unconstrained clauses and set $N = \{(\vec{v}\simeq\vec{x} \,\|\, C) \mid C \in N_{\neg\phi}\}$.

Each clause $\alpha_i \,\|\, C_i \in N$ is now separately narrowed using $R(D)$, producing a set N_i of constrained clauses. The simple structure of $R(D)$ for ARMs guarantees that this process terminates, and in particular that the set A_{N_i} of constraints of empty clauses in N_i is finite. Because $R(D)$ only has a single Herbrand model over $(\mathcal{P} \cup \mathcal{P}', \mathcal{F})$, Proposition 4 ensures that the union of the sets A_{N_i} is covering iff $\mathcal{M}(D) \not\models N_{\neg\phi}$, i.e. iff $\mathcal{M}(D) \models \forall\vec{x}.\exists\vec{y}.\phi$.

$\mathcal{M}(D) \models \exists\vec{x}.\forall\vec{y}.\phi$ is decided analogously, without negating ϕ. ■

5 Conclusion

We have extended the decidability results of [9] for ARMs to arbitrary formulas with one quantifier alternation and for DIGs to several more restrictive formula structures with one quantifier alternation.

Our approach has potential for further research. We restricted our attention to a non-equational setting, whereas our initial fixed domain calculus [12] considers equations as well. It is an open problem to what extend our results also hold in

an equational setting. In [9], the finite and infinite (open) signature semantics for DIGs was considered. Our results refer to the finite signature semantics where actually only the signature symbols of a finite saturated set are considered in the minimal model. It is not known what an infinite (further symbols) signature semantics means to our approach. Finally, in [9] the question was raised what happens if one considers more restrictive, e.g., linear DIGs. We know that linear DIGs require less effort in predicate completion but it is an open question whether this has further effects on decidability (complexity) results.

Acknowledgements. We thank Peter Baumgartner for valuable background information. The authors are supported by the German Transregional Collaborative Research Center SFB/TR 14 AVACS.

References

1. Bachmair, L., Ganzinger, H.: Rewrite-based equational theorem proving with selection and simplification. Journal of Logic and Computation 4(3), 217–247 (1994)
2. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 2, vol. I, pp. 19–99. Elsevier, Amsterdam (2001)
3. Baumgartner, P., Tinelli, C.: The model evolution calculus. In: Baader, F. (ed.) CADE 2003. LNCS, vol. 2741, pp. 350–364. Springer, Heidelberg (2003)
4. Caferra, R., Leitsch, A., Peltier, N.: Automated Model Building. Applied Logic Series, vol. 31. Kluwer, Dordrecht (2004)
5. Comon, H., Lescanne, P.: Equational problems and disunification. Journal of Symbolic Computation 7(3-4), 371–425 (1989)
6. Comon, H., Nieuwenhuis, R.: Induction = I-Axiomatization + First-Order Consistency. Information and Computation 159(1/2), 151–186 (2000)
7. Fermüller, C.G., Leitsch, A.: Hyperresolution and automated model building. Journal of Logic and Computation 6(2), 173–203 (1996)
8. Fermüller, C.G., Leitsch, A., Hustadt, U., Tamet, T.: Resolution decision procedures. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 25, vol. II, pp. 1791–1849. Elsevier, Amsterdam (2001)
9. Fermüller, C.G., Pichler, R.: Model representation over finite and infinite signatures. Journal of Logic and Computation 17(3), 453–477 (2007)
10. Ganzinger, H., Stuber, J.: Inductive theorem proving by consistency for first-order clauses. In: Rusinowitch, M., Rémy, J. (eds.) CTRS 1992. LNCS, vol. 656, pp. 226–241. Springer, Heidelberg (1992)
11. Hillenbrand, T., Weidenbach, C.: Superposition for finite domains. Research Report MPI-I-2007-RG1-002, Max Planck Institute for Computer Science, Saarbrücken, Germany (April 2007)
12. Horbach, M., Weidenbach, C.: Superposition for fixed domains. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 293–307. Springer, Heidelberg (2008)
13. Horbach, M., Weidenbach, C.: Decidability results for saturation-based model building. Technical Report MPI-I-2009-RG1-004, Max Planck Institute for Computer Science, Saarbrücken, Germany (2009)

14. Hustadt, U., Schmidt, R., Georgieva, L.: A survey of decidable first-order fragments and description logics. J. of Relational Methods in CS 1, 251–276 (2004)
15. Lassez, J.-L., Marriott, K.: Explicit representation of terms defined by counter examples. Journal of Automated Reasoning 3(3), 301–317 (1987)
16. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 7, vol. I, pp. 371–443. Elsevier, Amsterdam (2001)
17. Weidenbach, C.: Combining superposition, sorts and splitting. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 27, vol. 2, pp. 1965–2012. Elsevier, Amsterdam (2001)

A Tableau Calculus for Regular Grammar Logics with Converse*

Linh Anh Nguyen[1] and Andrzej Szałas[1,2]

[1] Institute of Informatics, University of Warsaw
Banacha 2, 02-097 Warsaw, Poland
{nguyen,andsz}@mimuw.edu.pl
[2] Department of Computer and Information Science, Linköping University
SE-581 83 Linköping, Sweden

Abstract. We give a sound and complete tableau calculus for deciding the general satisfiability problem of regular grammar logics with converse (REG^c logics). Tableaux of our calculus are defined as "and-or" graphs with global caching. Our calculus extends the tableau calculus for regular grammar logics given by Goré and Nguyen [11] by using a cut rule and existential automaton-modal operators to deal with converse. We use it to develop an EXPTIME (optimal) tableau decision procedure for the general satisfiability problem of REG^c logics. We also briefly discuss optimizations for the procedure.

1 Introduction

Grammar logics were introduced by Fariñas del Cerro and Penttonen in [8] and have been studied widely, e.g., in [2,4,6,11,23]. In [2], Baldoni *et al.* gave a prefixed tableau calculus for grammar logics and used it to show that the general (uniform) satisfiability problem[1] of right linear grammar logics is decidable and the general satisfiability problem of context-free grammar logics is undecidable. In [4], by using a transformation into the satisfiability problem of propositional dynamic logic (PDL), Demri proved that the general satisfiability problem of regular grammar logics is EXPTIME-complete. In [6], Demri and de Nivelle gave a translation of the satisfiability problem of grammar logics with converse into the two-variable guarded fragment GF^2 of first-order logic and have shown that the general satisfiability problem of regular grammar logics with converse is also EXPTIME-complete. In [11], Goré and Nguyen gave an EXPTIME tableau decision procedure for the general satisfiability problem of regular grammar logics.

In this work we consider theorem proving in regular grammar logics with converse (REG^c logics). The class of these logics is large and contains many common and useful modal logics. Here are some examples (see also [6]):

- All 15 basic monomodal logics obtained from K by adding an arbitrary combination of axioms D, T, B, 4, 5 are REG^c logics.[2] The multimodal versions of these logics are also REG^c logics.

* Supported by the MNiSW grants N N206 3982 33 and N N206 399334.

[1] I.e., the specification of the logic is also given as an input of the problem.

[2] See [6] for 10 of them. For the 5 remaining logics, use $\langle\sigma\rangle\top$ as global assumptions.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 421–436, 2009.

- The description logic $\mathcal{SHI}$ and its extension with complex role inclusion axioms studied by Horrocks and Sattler in [18] are REG^c logics. The whole class of regular grammar logics has also been studied, for example by Nguyen [23], as a class of description logics. Note that the notion of complex role inclusion axiom given in [18] is strictly less general than the notion of inclusion axiom (of the form $[\sigma]\varphi \rightarrow [\varrho_1]\ldots[\varrho_n]\varphi$) of REG^c logics. REG^c can be treated as an extension of the description logic $\mathcal{SHI}$, which, when extended with number restrictions and other concept constructors, would yield very expressive description logics.
- Regular modal logics of agent beliefs studied by Goré and Nguyen in [13] are REG^c logics. Those logics use only a simple form of axiom 5 for expressing negative introspection of single agents. Axioms of REG^c can be used to express negative introspection of groups of agents.[3] However, in contrast with the logics studied in [13], cuts seem not eliminable for traditional (unlabeled) tableau calculi for the whole class REG^c.

There are two main approaches for theorem proving in modal logics: the direct approach, where one develops a theorem prover directly for the logic under consideration, and the translation-based approach, where one translates the logic into some other logic with developed proof techniques. The translation method proposed by Demri and de Nivelle [6] for REG^c logics is interesting from the theoretical point of view. It allows one to establish the complexity and sheds new light on translation approach for modal logics. On the other hand, as stated in [6], the direct approach has the advantage that a specialized algorithm can make use of specific properties of the logic under consideration, enabling optimizations that would not work in general.[4] The direct approach based on tableaux has been widely applied for modal logics [25,9,26,2,10,3], because it allows many useful optimization techniques (see, e.g., [17,7,20]), some of which are specific for tableaux.[5]

As far as we know, no tableau calculi have been developed for REG^c logics. In the preprint [5], Demri and de Nivelle gave (also) a translation of REG^c logics into CPDL (converse PDL). One can use that translation together with the tableau decision procedure for CPDL given by De Giacomo and Massacci [3] for deciding REG^c logics. This method uses the translation approach and, additionally, has the disadvantage that the formal decision procedure given in [3] for CPDL has non-optimal NEXPTIME

[3] The general form $\neg[\sigma_1]\varphi \rightarrow [\sigma_2]\neg[\sigma_3]\varphi$ of axiom 5 can be expressed as $[\overline{\sigma}_3]\psi \rightarrow [\overline{\sigma}_1][\sigma_2]\psi$. Here, σ_1, σ_2, and σ_3 may represent groups of agents.

[4] The first author has implemented a complexity-optimal tableau prover called TGC for the description logic $\mathcal{ALC}$ with various optimizations (see Section 5.2). The prover can be extended for deciding REG^c logics in a natural way, using our calculus for REG^c logics. Note that complexity-optimal tableau decision procedures for EXPTIME modal/description logics are very rare, and as far as we know and according to [1, page 26], no such procedures have been implemented for REG^c logics, CPDL, μ-calculus, or GF^2. From our experience on optimizing TGC [20], sometimes a small modification may significantly increase or decrease the performance of a prover. As a translation cannot be treated as a small modification, the direct approach should not be neglected.

[5] Not all optimization techniques proposed in [17,7,20] are particularly useful. Also, they cannot be combined all together. But each of [17,7,20] proposes a number of specific good ideas for optimizing tableau decision procedures.

complexity. Although De Giacomo and Massacci [3] described also a transformation of their NEXPTIME algorithm into an EXPTIME decision procedure for CPDL, the description is informal and unclear: the transformation is based on Pratt's global caching method formulated for PDL [25], but no global caching method has been formalized and proved sound for labeled tableaux that allow modifying labels of ancestor nodes in order to deal with converse.

In this work we develop a sound and complete tableau calculus for deciding the general satisfiability problem of REG^c logics. Our calculus is an extension of the tableau calculus for regular grammar logics given by Goré and Nguyen in [11]. To deal with converse, we use an analytic cut rule. Similarly as in [21,12], our cut rule is a kind of "guessing the future" for nodes in traditional (unlabeled) tableaux. Besides, there is a substantial difference comparing to [11]. Namely, while Goré and Nguyen introduced only universal automaton-modal operators for regular grammar logics, using cuts to deal with converse we have to use also existential automaton-modal operators. Consequently, our calculus for REG^c deals also with fulfilling "eventualities" (like operators $\langle\alpha^*\rangle$ of PDL). For that we adopt the tableau method given by Pratt for PDL [25], but with a more direct formulation. Our tableaux in REG^c logics are "and-or" graphs constructed using traditional tableau rules and global caching. The idea of global caching appeared in Pratt's work [25] on PDL and has been formalized and proved sound by Goré and Nguyen for traditional tableaux in a number of other modal and description logics [11,12,13,14]. Similarly as for PDL [25] but in contrast with [11,12,13,14], checking satisfiability in REG^c logics deals not only with the local consistency but also with a global consistency property of the constructed "and-or" graph.

Using our calculus, we develop an EXPTIME (optimal) tableau decision procedure for the general satisfiability problem of REG^c logics, to which a number of useful optimization techniques can be applied.

The rest of this paper is structured as follows. We give definitions for REG^c logics in Section 2, present our calculus in Section 3, prove its completeness in Section 4, present our decision procedure and discuss optimizations for it in Section 5. Concluding remarks are given in Section 6. Due to the lack of space, some proofs are presented only in the full version [24] of this paper.

2 Preliminaries

2.1 Regular Semi-thue Systems

Let Σ^+ be a finite set of symbols. For $\sigma \in \Sigma^+$, we use $\overline{\sigma}$ to denote a fresh symbol, called the *converse* of σ. We use notation $\Sigma^- = \{\overline{\sigma} \mid \sigma \in \Sigma^+\}$ and assume that $\Sigma^- \cap \Sigma^+ = \emptyset$. For $\varrho = \overline{\sigma} \in \Sigma^-$, let $\overline{\varrho} \stackrel{\text{def}}{=} \sigma$. By an *alphabet with converse* we understand $\Sigma = \Sigma^+ \cup \Sigma^-$.

A *context-free semi-Thue system* S over Σ consists of a set of context-free production rules over alphabet Σ. So it is like a context free grammar, but it has no designated start symbol and there is no distinction between terminal and non-terminal symbols. We assume that for $\sigma \in \Sigma$, the word σ is derivable from σ by such a grammar. We identify S with its set of production rules. We say that S is *symmetric* if, for every rule $\sigma \to \varrho_1 \dots \varrho_k$ of S, the rule $\overline{\sigma} \to \overline{\varrho}_k \dots \overline{\varrho}_1$ also belongs to S.

A context-free semi-Thue system S over Σ is called a *regular semi-Thue system* S over Σ if, for every $\sigma \in \Sigma$, the set of words derivable from σ using the system is a regular language over Σ. Similarly as in [6], we assume that any considered regular semi-Thue system S is always given together with a mapping A that associates each $\sigma \in \Sigma$ with a finite automaton A_σ recognizing words derivable from σ using S. We call A the mapping specifying the finite automata of S. Note that it is undecidable to check whether a context-free semi-Thue system is regular since it is undecidable whether the language generated by a linear grammar is regular [19].

Recall that a *finite automaton* A over alphabet Σ is a tuple $\langle \Sigma, Q, I, \delta, F \rangle$, where Q is a finite set of states, $I \subseteq Q$ is the set of initial states, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and $F \subseteq Q$ is the set of accepting states. A *run* of A on a word $\varrho_1 \dots \varrho_k$ is a finite sequence of states $q_0, q_1, \dots, q_k$ such that $q_0 \in I$ and $\delta(q_{i-1}, \varrho_i, q_i)$ holds for every $1 \leq i \leq k$. It is an *accepting run* if $q_k \in F$. We say that A *accepts* a word w if there exists an accepting run of A on w.

2.2 Regular Grammar Logics with Converse

Our language is based on a set Σ of *modal indices*, which is an alphabet with converse, and a set Φ_0 of *propositions*. *Formulas* of our *primitive language* are defined using the following BNF grammar, with $p \in \Phi_0$ and $\sigma \in \Sigma$:

$$\varphi, \psi ::= \top \mid \bot \mid p \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \rightarrow \psi \mid [\sigma]\varphi \mid \langle\sigma\rangle\varphi$$

A formula is in negation normal form (NNF) if it does not contain $\rightarrow$ and uses $\neg$ only before propositions. Every formula (of the primitive language) can be transformed to an equivalent in NNF. By $\overline{\varphi}$ we denote the NNF of $\neg\varphi$.

A *Kripke model* is a tuple $M = \langle W, R, h \rangle$, where W is a non-empty set of possible worlds, R is a function that maps each $\sigma \in \Sigma$ to a binary relation R_σ on W (which is called the accessibility relation for σ), and h is a function that maps each $w \in W$ to a set $h(w) \subseteq \Phi_0$ of propositions that are true at w. The structure is required to satisfy the property that $R_{\overline{\sigma}} = R_\sigma^- = \{(y, x) \mid (x, y) \in R_\sigma\}$ for every $\sigma \in \Sigma$.

Given a Kripke model $M = \langle W, R, h \rangle$ and a world $w \in W$, the *satisfaction relation* $\models$ is defined as usual for the classical connectives with two extra clauses for the modalities as below:

$$M, w \models [\sigma]\varphi \;\text{ iff }\; \forall v \in W[R_\sigma(w, v)] \text{ implies } M, v \models \varphi$$
$$M, w \models \langle\sigma\rangle\varphi \;\text{ iff }\; \exists v \in W[R_\sigma(w, v)] \text{ and } M, v \models \varphi$$

We say that φ is *satisfied* at w in M (or M *satisfies* φ at w) if $M, w \models \varphi$. We say that M *satisfies* a set X of formulas at w, denoted by $M, w \models X$, if $M, w \models \varphi$ for all $\varphi \in X$. We say that M *validates* X if $M, w \models X$ for every world w of M.

Given two binary relations R_1 and R_2 over W, their relational composition $R_1 \circ R_2 \stackrel{\text{def}}{=} \{(x, y) \mid \exists z \in W[R_1(x, z) \wedge R_2(z, y)]\}$ is also a binary relation over W.

Let S be a symmetric regular semi-Thue system over Σ. The regular grammar logic with converse corresponding to S, denoted by $L(S)$, is characterized by the class of admissible Kripke models $M = \langle W, R, h \rangle$ such that, for every rule $\sigma \rightarrow \varrho_1 \dots \varrho_k$ of S, $R_{\varrho_1} \circ \dots \circ R_{\varrho_k} \subseteq R_\sigma$. Such a structure is called an L-*model*, for $L = L(S)$. We use REG^c to denote the class of regular grammar logics with converse.

Let L be a REG^c logic and X, Γ be finite sets of formulas. We say that X is L-*satisfiable* w.r.t. the set Γ of global assumptions if there exists an L-model that validates Γ and satisfies X at some possible world.

3 A Tableau Calculus for REG^c

From now on, let S be a symmetric regular semi-Thue system over Σ, A be the mapping specifying the finite automata of S, and L be the REG^c logic corresponding to S. For $\sigma \in \Sigma$, we write A_σ in the form $\langle \Sigma, Q_\sigma, I_\sigma, \delta_\sigma, F_\sigma \rangle$.

For the tableau calculus defined here we extend the primitive language with the auxiliary modal operators $\Box_\sigma$, $[A_\sigma, q]$ and $\langle A_\sigma, q \rangle$, where $\sigma \in \Sigma$ and q is a state of A_σ. In the extended language, if φ is a formula, then $\Box_\sigma \varphi$, $[A_\sigma, q]\varphi$ and $\langle A_\sigma, q \rangle \varphi$ are also formulas. The semantics of such formulas is defined as follows. Given a Kripke model $M = \langle W, R, h \rangle$ and a world $w \in W$, define that

- $M, w \models \Box_\sigma \varphi$ if $M, w \models [\sigma]\varphi$;
- $M, w \models [A_\sigma, q]\varphi$ (resp. $M, w \models \langle A_\sigma, q \rangle \varphi$) if $M, w_k \models \varphi$ for all (resp. some) $w_k \in W$ such that there exist worlds $w_0 = w, w_1, \ldots, w_k$ of M, with $k \geq 0$, states $q_0 = q, q_1, \ldots, q_k$ of A_σ with $q_k \in F_\sigma$, and a word $\varrho_1 \ldots \varrho_k$ over Σ such that $R_{\varrho_i}(w_{i-1}, w_i)$ and $\delta_\sigma(q_{i-1}, \varrho_i, q_i)$ hold for all $1 \leq i \leq k$.

The operators $\Box_\sigma$ and $[A_\sigma, q]$ are universal modal operators, while $\langle A_\sigma, q \rangle$ is the existential modal operator dual to $[A_\sigma, q]$. Although $\Box_\sigma \varphi$ has the same semantics as $[\sigma]\varphi$, the operator $\Box_\sigma$ behaves differently than $[\sigma]$ in our calculus. The intuition of these auxiliary operators is as follows. Suppose that a word $\varrho_1 \ldots \varrho_n$ is derivable from σ using a sequence of the rules of S, which may be arbitrarily long. Then $R_{\varrho_1} \circ \cdots \circ R_{\varrho_n} \subseteq R_\sigma$ holds for every *L*-model $\langle W, R, h \rangle$, and hence $[\sigma]\varphi \rightarrow [\varrho_1] \ldots [\varrho_n]\varphi$ is *L*-valid for any φ. So, having $[\sigma]\varphi$ we may need to derive $[\varrho_1] \ldots [\varrho_n]\varphi$. But n may be arbitrarily large (as the sequence of applied production rules may be arbitrarily long) and the formula may be too big. A solution for this problem is to use the finite automaton A_σ to control the behavior of $[\sigma]$. We treat $[\sigma]\varphi$ as the conjunction of $\{[A_\sigma, q]\varphi \mid q \in I_\sigma\}$. Having $[A_\sigma, q]\varphi$ at a possible world u, if $R_\varrho(u, v)$ and $\delta_\sigma(q, \varrho, q')$ hold then we can add $[A_\sigma, q']\varphi$ to v. We deal with this by deriving $\Box_\varrho [A_\sigma, q']\varphi$ from $[A_\sigma, q]\varphi$ when $\delta_\sigma(q, \varrho, q')$ holds. We use $\Box_\varrho$ here instead of $[\varrho]$ because the modal operator is needed only for atomic ϱ-transitions and we do not need to automatize $\Box_\varrho$ as in the case of $[\varrho]$.[6] Automaton-modal operators, especially universal ones, have previously been used, for example, in [15,18,11,16,22].[7]

[6] The operators $\Box_\sigma$ are introduced to simplify the rule (cut) given in Table 1 and make it more intuitive. The use of $\Box_\sigma$ in the rules ([A]) and ($[A]_f$) is just for convenience. The rules ([A]) and ($[A]_f$) are eliminable (by modifying the rule $(trans)$ appropriately).

[7] We have tried to use universal modal operators indexed by a reversed finite automaton instead of existential automaton-modal operators, but did not succeed with that. In the presence of converse, the difficulty lies in that one can travel forward and backward along the skeleton tree that unfolds the model under construction in an arbitrary way, making returns at different possible worlds and continuing the travel from the current world many times before a final return to the current world.

For a set X of formulas, by $psf(X)$ we denote the set of all formulas φ, $\overline{\varphi}$ of the primitive language such that either $\varphi \in X$ or φ is a subformula of some formula of X. The closure $cl_L(X)$ is defined as

$$cl_L(X) = psf(X) \cup \{[A_\sigma, q]\varphi, \Box_\varrho[A_\sigma, q]\varphi, \langle A_\sigma, q\rangle\varphi, \Box_\varrho\langle A_\sigma, q\rangle\varphi, \langle\varrho\rangle\langle A_\sigma, q\rangle\varphi \mid \sigma, \varrho \in \Sigma, q \in Q_\sigma, \varphi \in psf(X), \text{ and } ([\sigma]\varphi \in psf(X) \text{ or } [A_\sigma, q']\varphi \in X \text{ for some } q')\}.$$

For $\sigma \in \Sigma$ and $q \in Q_\sigma$, define $\delta_\sigma(q) = \{(\varrho, q') \mid (q, \varrho, q') \in \delta_\sigma\}$.

Let X and Γ be finite sets of formulas in NNF of the primitive language. We define now a tableau calculus $\mathcal{CL}$ for the problem of checking whether X is *L*-satisfiable w.r.t. the set Γ of global assumptions.[8]

Tableau rules are written downwards, with a set of formulas above the line as the *premise* and a number of sets of formulas below the line as the *(possible) conclusions*. A k-ary tableau rule has k possible conclusions. A tableau rule is either an *"or"-rule* or an *"and"-rule*. Possible conclusions of an "or"-rule are separated by '|', while conclusions of an "and"-rule are separated/specified using '&'. If a rule is a unary rule or an "and"-rule then its conclusions are "firm" and we ignore the word "possible". An "or"-rule has the meaning that, if the premise is *L*-satisfiable w.r.t. Γ then some of the possible conclusions is also *L*-satisfiable w.r.t. Γ. On the other hand, an "and"-rule has the meaning that, if the premise is *L*-satisfiable w.r.t. Γ then all of the conclusions are also *L*-satisfiable w.r.t. Γ (possibly at different worlds of the model under construction).

We use Y to denote a set of formulas, and denote the set $Y \cup \{\varphi\}$ by Y, φ.

We define the tableau calculus $\mathcal{CL}$ w.r.t. a set Γ of global assumptions for the REG^c logic *L* to be the set of the tableau rules given in Table 1.

Notice that the premise of any rule among $(\wedge)$, $(\vee)$, (aut), $([\mathrm{A}])$, $([\mathrm{A}]_f)$, (cut) is a subset of every possible conclusion of the rule. We assume that these rules are applicable only when the premise is a *proper* subset of each of the possible conclusions. Such rules are said to be *monotonic*.

The rule $(trans)$ is the only "and"-rule and the only *transitional rule*. Instantiating this rule, for example, to $Y = \{\langle\sigma\rangle p, \langle\sigma\rangle q, \Box_\sigma r\}$ and $\Gamma = \{s\}$ we get two conclusions: $\{p, r, s\}$ and $\{q, r, s\}$. The other rules are "or"-rules, which are also called *static rules*.[9] The intuition of the sorting of static/transitional is that the static rules keep us at the same possible world of the model under construction, while each conclusion of the transitional rule takes us to a new possible world.

For any rule of $\mathcal{CL}$ except (cut) and $(trans)$, the distinguished formulas of the premise are called the *principal formulas* of the rule. The principal formulas of the rule $(trans)$ are the formulas of the form $\langle\sigma\rangle\varphi$ of the premise. The rule (cut) does not have principal formulas.

The purpose of the restriction on the applicability of the rules $(\wedge)$, $(\vee)$, (aut), $([\mathrm{A}])$, $([\mathrm{A}]_f)$, (cut) is to guarantee that sequences of applications of static rules are always finite. Note that none of the static rules creates a formula of the form $\langle A_\sigma, q\rangle\varphi$ for the

[8] We incorporate global assumptions in order to make a direct connection with description logic (DL). The set of global assumptions plays the role of a TBox of DL. It is known that in some DLs the TBox can be "internalized", but the transformation approach is not practical.

[9] Unary static rules can be treated either as "and"-rules or as "or"-rules.

possible conclusions (provided that the premise is a subset of $cl_L(X \cup \Gamma)$). That is why we do not make the rules $(\langle \mathrm{A} \rangle)$ and $(\langle \mathrm{A} \rangle_f)$ monotonic. The second reason of this is that a formula of the form $\langle A_\sigma, q \rangle \varphi$ must be "reduced" as a principal formula of $(\langle \mathrm{A} \rangle)$ or $(\langle \mathrm{A} \rangle_f)$ because any one of the possible conclusions may play a key role in fulfilling the eventuality expressed by the formula.

Table 1. Rules of the tableau calculus for REG^c

$$(\bot_0)\ \frac{Y, \bot}{\bot} \qquad (\bot)\ \frac{Y, p, \neg p}{\bot}$$

$$(\wedge)\ \frac{Y, \varphi \wedge \psi}{Y, \varphi \wedge \psi, \varphi, \psi} \qquad (\vee)\ \frac{Y, \varphi \vee \psi}{Y, \varphi \vee \psi, \varphi \mid Y, \varphi \vee \psi, \psi}$$

$$(aut)\ \frac{Y, [\sigma]\varphi}{Y, [\sigma]\varphi, [A_\sigma, q_1]\varphi, \ldots, [A_\sigma, q_k]\varphi} \text{ if } I_\sigma = \{q_1, \ldots, q_k\}$$

if $\delta_\sigma(q) = \{(\varrho_1, q_1), \ldots, (\varrho_k, q_k)\}$ and $q \notin F_\sigma$:

$$([\mathrm{A}])\ \frac{Y, [A_\sigma, q]\varphi}{Y, [A_\sigma, q]\varphi, \Box_{\varrho_1}[A_\sigma, q_1]\varphi, \ldots, \Box_{\varrho_k}[A_\sigma, q_k]\varphi}$$

$$(\langle \mathrm{A} \rangle)\ \frac{Y, \langle A_\sigma, q \rangle \varphi}{Y, \langle \varrho_1 \rangle \langle A_\sigma, q_1 \rangle \varphi \mid \ldots \mid Y, \langle \varrho_k \rangle \langle A_\sigma, q_k \rangle \varphi}$$

if $\delta_\sigma(q) = \{(\varrho_1, q_1), \ldots, (\varrho_k, q_k)\}$ and $q \in F_\sigma$:

$$([\mathrm{A}]_f)\ \frac{Y, [A_\sigma, q]\varphi}{Y, [A_\sigma, q]\varphi, \Box_{\varrho_1}[A_\sigma, q_1]\varphi, \ldots, \Box_{\varrho_k}[A_\sigma, q_k]\varphi, \varphi}$$

$$(\langle \mathrm{A} \rangle_f)\ \frac{Y, \langle A_\sigma, q \rangle \varphi}{Y, \langle \varrho_1 \rangle \langle A_\sigma, q_1 \rangle \varphi \mid \ldots \mid Y, \langle \varrho_k \rangle \langle A_\sigma, q_k \rangle \varphi \mid Y, \varphi}$$

$$(cut)\ \frac{Y}{Y, [A_\sigma, q]\varphi \mid Y, \Box_\varrho \langle A_\sigma, q' \rangle \overline{\varphi}}$$

if Y contains a formula $\langle \varrho \rangle \psi, [A_\sigma, q']\varphi$ belongs to $cl_L(Y \cup \Gamma)$, and $(q', \overline{\varrho}, q) \in \delta_\sigma$

$$(trans)\ \frac{Y}{\&\{\ (\{\varphi\} \cup \{\psi \text{ s.t. } \Box_\sigma \psi \in Y\} \cup \Gamma) \text{ s.t. } \langle \sigma \rangle \varphi \in Y\ \}}$$

We assume the following preferences for the rules of $\mathcal{CL}$: the rules $(\bot_0)$ and $(\bot)$ have the highest priority; unary static rules have a higher priority than non-unary static rules; the rule (cut) has the lowest priority among static rules; all the static rules have a higher priority than the transitional rule $(trans)$.

An *"and-or" graph* for (X, Γ) w.r.t. $\mathcal{CL}$, also called a *$\mathcal{CL}$-tableau* for (X, Γ), is an "and-or" graph defined as follows. The root of the graph has contents (i.e., is labeled by) $X \cup \Gamma$. For every node v of the graph, if a k-ary static rule δ is applicable to (the contents of) v in the sense that an instance of δ has the contents of v as the premise and $Z_1, \ldots, Z_k$ as the possible conclusions, then choose such a rule accordingly to the preferences and apply it to v to create k successors of v with contents $Z_1, \ldots, Z_k$, respectively. If the graph already contains a node with contents Z_i then instead of creating a new node with contents Z_i as a successor of v we just connect v to that node. If no static rules but the transitional rule $(trans)$ is applicable to v then apply $(trans)$ to v in a similar way, where each edge (v, w) of the graph is caused by a principal formula of the application, which is used as the *label of the edge*. If the rule applied to v is a static rule then v is an *"or"-node* (as the rule is an "or"-rule). If the rule applied to v is the transitional rule $(trans)$ then v is an *"and"-node* (as $(trans)$ is an "and"-rule). If no rule is applicable to v then v is an *end node*. Note that each node is "expanded" only once (using one rule). Also note that the graph is constructed using *global caching* [25,12,14] and each of its nodes has unique contents. See [24] for an example of "and-or" graph.

Apart from monotonicity, notice also the other restrictions on the applicability of (cut). Observe that, if L is essentially a regular grammar logic without converse (in the sense that for every rule $\sigma \rightarrow \varrho_1 \ldots \varrho_k$ of S either $\{\sigma, \varrho_1, \ldots, \varrho_k\} \subseteq \Sigma^+$ or $\{\sigma, \varrho_1, \ldots, \varrho_k\} \subseteq \Sigma^-$) and the formulas of $X \cup \Gamma$ do not use modal indices from Σ^-, then the rule (cut) will never be used.

A *marking* of an "and-or" graph G is a subgraph G' of G such that:

- The root of G is the root of G'.
- If v is a node of G' and is an "or"-node of G then there exists at least one edge (v, w) of G that is an edge of G'.
- If v is a node of G' and is an "and"-node of G then every edge (v, w) of G is an edge of G'.
- If (v, w) is an edge of G' then v and w are nodes of G'.

Let G be an "and-or" graph for (X, Γ) w.r.t. $\mathcal{CL}$, G' be a marking of G, v be a node of G', and $\langle A_\sigma, q\rangle\varphi$ be a formula of the contents of v. A *trace* of $\langle A_\sigma, q\rangle\varphi$ in G' starting from v is a sequence $(v_0, \varphi_0), \ldots, (v_k, \varphi_k)$ such that:

- $v_0 = v$ and $\varphi_0 = \langle A_\sigma, q\rangle\varphi$;
- for every $1 \leq i \leq k$, (v_{i-1}, v_i) is an edge of G';
- for every $1 \leq i \leq k$, φ_i is a formula of the contents of v_i such that:
  - if φ_{i-1} is not a principal formula of the tableau rule expanding v_{i-1} then the rule must be a static rule and $\varphi_i = \varphi_{i-1}$,
  - else if the rule is $(\langle\mathrm{A}\rangle)$ or $(\langle\mathrm{A}\rangle_f)$ then φ_{i-1} is of the form $\langle A_\sigma, q'\rangle\varphi$ and φ_i is the formula obtained from φ_{i-1},
  - else the rule is $(trans)$, φ_{i-1} is of the form $\langle\sigma\rangle\langle A_\sigma, q'\rangle\varphi$ and is the label of the edge (v_{i-1}, v_i) and $\varphi_i = \langle A_\sigma, q'\rangle\varphi$.

A trace $(v_0, \varphi_0), \ldots, (v_k, \varphi_k)$ of $\langle A_\sigma, q\rangle\varphi$ in G' is called a $\Diamond$-*realization* in G' for $\langle A_\sigma, q\rangle\varphi$ at v_0 if $\varphi_k = \varphi$.

A marking G' of an "and-or" graph G is *consistent* if:

Local Consistency: G' does not contain any node with contents $\{\bot\}$;
Global Consistency: for every node v of G', every formula of the form $\langle A_\sigma, q\rangle\varphi$ of the contents of v has a $\Diamond$-realization (starting from v) in G'.

Theorem 3.1 (Soundness and Completeness of $\mathcal{CL}$). *Let S be a symmetric regular semi-Thue system over Σ, A be the mapping specifying the finite automata of S, and L be the REG^c logic corresponding to S. Let X and Γ be finite sets of formulas in NNF of the primitive language, and G be an "and-or" graph for (X, Γ) w.r.t. $\mathcal{CL}$. Then X is L-satisfiable w.r.t. the set Γ of global assumptions iff G has a consistent marking.* ◁

The "only if" direction means soundness of $\mathcal{CL}$, while the "if" direction means completeness of $\mathcal{CL}$. See the full version [24] for the proof of soundness. The completeness is proved by Lemma 4.2 given in the next section.

4 Proof of Completeness

We prove completeness of $\mathcal{CL}$ via model graphs. The technique has been used in [26,10,21] for logics without induction rules (like the one of PDL). A *model graph* is a tuple $\langle W, R, H\rangle$, where W is a set of nodes, R is a mapping that maps each $\sigma \in \Sigma$ to a binary relation R_σ on W, and H is a function that maps each node of W to a set of formulas. We use model graphs merely as data structures, but we are interested in "consistent" and "saturated" model graphs defined below.

Model graphs differ from "and-or" graphs in that a model graph contains only "and"-nodes and its edges are labeled by accessibility relations. Roughly speaking, given an "and-or" graph G with a consistent marking G', to construct a model graph one can stick together the nodes in a "saturation path" of a node of G' to create a node for the model graph. Details will be given later.

A trace of a formula $\langle A_\sigma, q\rangle\varphi$ at a node in a model graph is defined analogously as for the case of "and-or" graphs. Namely, given a model graph $M = \langle W, R, H\rangle$ and a node $v \in W$, a *trace* of a formula $\langle A_\sigma, q\rangle\varphi \in H(v)$ (starting from v) is a chain $(v_0, \varphi_0), \ldots, (v_k, \varphi_k)$ such that:

- $v_0 = v$ and $\varphi_0 = \langle A_\sigma, q\rangle\varphi$;
- for every $1 \le i \le k$, $\varphi_i \in H(v_i)$;
- for every $1 \le i \le k$, if $v_i = v_{i-1}$ then:
  - φ_{i-1} is of the form $\langle A_\sigma, q'\rangle\varphi$, and
  - either $\varphi_i = \langle \varrho\rangle\langle A_\sigma, q''\rangle\varphi$ for some ϱ and q'' such that $\delta_\sigma(q', \varrho, q'')$
  - or $\varphi_i = \varphi$ and $q' \in F_\sigma$ and $i = k$;
- for every $1 \le i \le k$, if $v_i \ne v_{i-1}$ then:
  - φ_{i-1} is of the form $\langle \varrho\rangle\langle A_\sigma, q'\rangle\varphi$ and $\varphi_i = \langle A_\sigma, q'\rangle\varphi$ and $(v_{i-1}, v_i) \in R_\varrho$.

A trace $(v_0, \varphi_0), \ldots, (v_k, \varphi_k)$ of $\langle A_\sigma, q\rangle\varphi$ in a model graph M is called a $\Diamond$-*realization* for $\langle A_\sigma, q\rangle\varphi$ at v_0 if $\varphi_k = \varphi$.

Similarly as for markings of "and-or" graphs, we define that a model graph $M = \langle W, R, H\rangle$ is *consistent* if:

Local Consistency: for every $v \in W$, $H(v)$ contains neither $\bot$ nor a clashing pair of the form $p, \neg p$;

Global Consistency: for every $v \in W$, every formula $\langle A_\sigma, q\rangle\varphi$ of $H(v)$ has a $\Diamond$-realization (at v).

A model graph $M = \langle W, R, H\rangle$ is said to be *$\mathcal{CL}$-saturated* if the following conditions hold for every $v \in W$:

- for every $\varphi \in H(v)$:
  - if $\varphi = \psi \wedge \xi$ then $\{\psi, \xi\} \subset H(v)$,
  - if $\varphi = \psi \vee \xi$ then $\psi \in H(v)$ or $\xi \in H(v)$,
  - if $\varphi = \langle\sigma\rangle\psi$ then there exists w such that $R_\sigma(v, w)$ and $\psi \in H(w)$,
  - if $\varphi = [\sigma]\psi$ and $I_\sigma = \{q_1, \ldots, q_k\}$ then $\{[A_\sigma, q_1]\psi, \ldots, [A_\sigma, q_k]\psi\} \subset H(v)$,
  - if $\varphi = [A_\sigma, q]\psi$ and $\delta_\sigma(q) = \{(\varrho_1, q_1), \ldots, (\varrho_k, q_k)\}$ then $\{\Box_{\varrho_1}[A_\sigma, q_1]\psi, \ldots, \Box_{\varrho_k}[A_\sigma, q_k]\psi\} \subset H(v)$,
  - if $\varphi = [A_\sigma, q]\psi$ and $q \in F_\sigma$ then $\psi \in H(v)$,
  - if $\varphi = \Box_\sigma\psi$ and $R_\sigma(v, w)$ holds then $\psi \in H(w)$;
- if $R_\varrho(v, w)$ holds and $[A_\sigma, q']\varphi \in H(w)$ and $(q', \overline{\varrho}, q) \in \delta_\sigma$ then $[A_\sigma, q]\varphi \in H(v)$ or $\Box_\varrho\langle A_\sigma, q'\rangle\overline{\varphi} \in H(v)$.

The last condition corresponds to the rule (cut). As shown in [24], it can be strengthened to "if $R_\varrho(v, w)$ and $[A_\sigma, q']\varphi \in H(w)$ and $(q', \overline{\varrho}, q) \in \delta_\sigma$ then $[A_\sigma, q]\varphi \in H(v)$".

Given a model graph $M = \langle W, R, H\rangle$, the *L-model corresponding to M* is the Kripke model $M' = \langle W, R', h\rangle$ such that:

- $h(w) = \{p \in \Phi_0 \mid p \in H(w)\}$ for $w \in W$, and
- R'_σ for $\sigma \in \Sigma$ are the smallest binary relations on W such that:
  - $R_\sigma \subseteq R'_\sigma$ and $R'_{\overline{\sigma}} = (R'_\sigma)^-$ for every $\sigma \in \Sigma$, and
  - if $\sigma \to \varrho_1 \ldots \varrho_k \in S$, where S is the symmetric regular semi-Thue system of L, then $R'_{\varrho_1} \circ \cdots \circ R'_{\varrho_k} \subseteq R'_\sigma$.

Lemma 4.1. *Let X and Γ be finite sets of formulas in NNF of the primitive language, and let $M = \langle W, R, H\rangle$ be a consistent and $\mathcal{CL}$-saturated model graph such that $\Gamma \subseteq H(w)$ for all $w \in W$ and $X \subseteq H(\tau)$ for some $\tau \in W$. Then the L-model M' corresponding to M validates Γ and satisfies X at τ.*

See the full version [24] for the proof of this lemma.

Let G' be a consistent marking of an "and-or" graph and let v be a node of G'. A *saturation path* of v w.r.t. G' is a finite sequence $v_0 = v, v_1, \ldots, v_k$ of nodes of G', with $k \geq 0$, such that, for every $0 \leq i < k$, v_i is an "or"-node and (v_i, v_{i+1}) is an edge of G', and v_k is an "and"-node. Observe that such a saturation path exists.

Lemma 4.2 (Completeness). *Let X and Γ be finite sets of formulas in NNF of the primitive language, G be an "and-or" graph for (X, Γ) w.r.t. $\mathcal{CL}$. Suppose that G has a consistent marking G'. Then X is L-satisfiable w.r.t. the set Γ of global assumptions.*

Proof. We construct a model graph $M = \langle W, R, H\rangle$ as follows:

1. Let v_0 be the root of G' and $v_0, \ldots, v_k$ be a saturation path of v_0 w.r.t. G'. Set $R_\sigma = \emptyset$ for all $\sigma \in \Sigma$ and set $W = \{\tau\}$, where τ is a new node. Set $H(\tau)$ to the sum of the contents of all $v_0, \ldots, v_k$. Mark τ as *unresolved* and set $f(\tau) = v_k$. (Each node of M will be marked either as unresolved or as resolved, and f will map each node of M to an "and"-node of G'.)
2. While W contains unresolved nodes, take one unresolved node w_0 and do:

   (a) For every formula $\langle \varrho \rangle \varphi \in H(w_0)$ do:

      i. Let $\varphi_0 = \langle \varrho \rangle \varphi$ and $\varphi_1 = \varphi$.
      ii. Let $u_0 = f(w_0)$ and let u_1 be the node of G' such that the edge (u_0, u_1) is labeled by φ_0. (As a maintained property of f, φ_0 belongs to the contents of u_0, and hence φ_1 belongs to the contents of u_1.)
      iii. If φ is of the form $\langle A_\sigma, q \rangle \psi$ then:

         A. Let $(u_1, \varphi_1), \ldots, (u_l, \varphi_l)$ be a $\Diamond$-realization in G' for φ_1 at u_1.
         B. Let $u_l, \ldots, u_m$ be a saturation path of u_l w.r.t. G'.

      iv. Else let $u_1, \ldots, u_m$ be a saturation path of u_1 w.r.t. G'.
      v. Let $j_0 = 0 < j_1 < \ldots < j_{n-1} < j_n = m$ be all the indices such that, for $0 \leq j \leq m$, u_j is an "and"-node of G iff $j \in \{j_0, \ldots, j_n\}$. For $0 \leq s \leq n-1$, let $\langle \varrho_s \rangle \varphi_{j_s+1}$ be the label of the edge (u_{j_s}, u_{j_s+1}) of G'. (We have that $\varrho_0 = \varrho$.)
      vi. For $1 \leq s \leq n$ do:

         A. Let Z_s be the sum of the contents of the nodes $u_{j_{s-1}+1}, \ldots, u_{j_s}$.
         B. If there does not exist $w_s \in W$ such that $H(w_s) = Z_s$ then: add a new node w_s to W, set $H(w_s) = Z_s$, mark w_s as unresolved, and set $f(w_s) = u_{j_s}$.
         C. Add the pair (w_{s-1}, w_s) to $R_{\varrho_{s-1}}$.

   (b) Mark w_0 as resolved.

As H is a one-to-one function and $H(w)$ of each $w \in W$ is a subset of the closure $cl_L(X \cup \Gamma)$, the above construction terminates and results in a finite model graph.

Observe that, in the above construction we transform the chain $u_0, \ldots, u_m$ of nodes of G' to a chain $w_0, \ldots, w_n$ of nodes of M by sticking together nodes in every maximal saturation path. Hence, M is $\mathcal{CL}$-saturated and satisfies the local consistency property. For $w'_0 \in W$ and $\langle A_\sigma, q' \rangle \psi \in H(w'_0)$, the formula has a trace of length 2, whose second pair is either (w'_0, ψ) or $(w_0, \langle \varrho \rangle \langle A_\sigma, q \rangle \psi)$ for some w_0, ϱ, q. This together with Step 2(a)iiiA implies that M satisfies the global consistency property. Hence, M is a consistent and $\mathcal{CL}$-saturated model graph.

Consider Step 1 of the construction. As the contents of v_0 are $X \cup \Gamma$, we have that $X \subseteq H(\tau)$ and $\Gamma \subseteq H(\tau)$. Consider Step 2(a)vi of the construction, as $u_{j_{s-1}}$ is an "and"-node and $u_{j_{s-1}+1}$ is a successor of $u_{j_{s-1}}$ that is created by the transitional rule, the contents of $u_{j_{s-1}+1}$ contain Γ. Hence $\Gamma \subseteq H(w_s)$ for every $w_s \in W$. By Lemma 4.1, the Kripke model corresponding to M validates Γ and satisfies X at τ. Hence, X is L-satisfiable w.r.t. Γ. $\lhd$

5 An ExpTime Tableau Decision Procedure for REG^c

In this section, we present a simple EXPTIME tableau algorithm for checking L-satisfiability of a given set X of formulas w.r.t. a given set Γ of global assumptions. We also briefly discuss optimizations for the algorithm.

5.1 The Basic Algorithm

Let X and Γ be finite sets of formulas in NNF of the primitive language, G be an "and-or" graph for (X, Γ) w.r.t. $\mathcal{CL}$, and G' be a marking of G. The *graph G_t of traces of G' in G* is defined as follows:

- nodes of G_t are pairs (v, φ), where v is a node of G and φ is a formula of the contents of v,
- a pair $((v, \varphi), (w, \psi))$ is an edge of G_t if v is a node of G', φ is of the form $\langle A_\sigma, q\rangle\xi$ or $\langle\varrho\rangle\langle A_\sigma, q\rangle\xi$, and the sequence (v, φ), (w, ψ) is a fragment of a trace in G'.

A node (v, φ) of G_t is an *end node* if φ is a formula of the primitive language. A node of G_t is *productive* if there is a path connecting it to an end node.

In Figure 1 we present Algorithm 1 for checking L-satisfiability of X w.r.t. Γ. The algorithm starts by constructing an "and-or" graph G, with root v_0, for (X, Γ) w.r.t. $\mathcal{CL}$. After that it collects the nodes of G whose contents are L-unsatisfiable w.r.t. Γ. Such nodes are said to be *unsat* and kept in the set $UnsatNodes$. Initially, if G contains a node with contents $\{\bot\}$ then the node is *unsat*. When a node or a number of nodes become *unsat*, the algorithm propagates the status *unsat* backwards through the "and-or" graph using the procedure $updateUnsatNodes$ presented in Figure 1. This procedure has property that, after calling it if the root v_0 of G does not belong to $UnsatNodes$ then the maximal subgraph of G without nodes from $UnsatNodes$, denoted by G', is a marking of G. After each calling of $updateUnsatNodes$, the algorithm finds the nodes of G' that make the marking not satisfying the global consistency property. Such a task is done by creating the graph G_t of traces of G' in G and finding nodes v of G' such that the contents of v contain a formula of the form $\langle A_\sigma, q\rangle\varphi$ but $(v, \langle A_\sigma, q\rangle\varphi)$ is not a productive node of G_t. If the set V of such nodes is empty then G' is a consistent marking (provided that $v_0 \notin UnsatNodes$) and the algorithm stops with a positive answer. Otherwise, V is used to update $UnsatNodes$ by calling $updateUnsatNodes(G, UnsatNodes, V)$. After that call, if $v_0 \in UnsatNodes$ then the algorithm stops with a negative answer, else the algorithm repeats the loop of collecting *unsat* nodes. Note that we can construct G_t only the first time and update it appropriately each time when $UnsatNodes$ is changed.

Theorem 5.1. *Let S be a symmetric regular semi-Thue system over Σ, A be the mapping specifying the finite automata of S, and L be the REG^c logic corresponding to S. Let X and Γ be finite sets of formulas in NNF of the primitive language. Then Algorithm 1 is an* EXPTIME *decision procedure for checking whether X is L-satisfiable w.r.t. the set Γ of global assumptions.*

Proof (sketch). It is easy to show that the algorithm has the invariant that a consistent marking of G cannot contain any node of $UnsatNodes$. The algorithm returns *false*

Algorithm 1
Input: finite sets X and Γ of formulas in NNF of the primitive language,
the mapping A specifying the finite automata of the symmetric
regular semi-Thue system of the considered REG^c logic *L*.
Output: *true* if X is *L*-satisfiable w.r.t. Γ, and *false* otherwise.

1. construct an "and-or" graph G, with root v_0, for (X, Γ) w.r.t. $\mathcal{CL}$;
2. $UnsatNodes := \emptyset$;
3. if G contains a node v with contents $\{\bot\}$ then
   $updateUnsatNodes(G, UnsatNodes, \{v\})$;
4. if $v_0 \in UnsatNodes$ then return *false*;
5. let G' be the maximal subgraph of G without nodes from $UnsatNodes$;
   (we have that G' is a marking of G)
6. construct the graph G_t of traces of G' in G;
7. while $v_0 \notin UnsatNodes$ do:
   (a) let V be the set of all nodes v of G' such that the contents of v contain a formula of the form $\langle A_\sigma, q\rangle\varphi$ but $(v, \langle A_\sigma, q\rangle\varphi)$ is not a productive node of G_t;
   (b) if $V = \emptyset$ then return *true*;
   (c) $updateUnsatNodes(G, UnsatNodes, V)$;
   (d) if $v_0 \in UnsatNodes$ then return *false*;
   (e) let G' be the maximal subgraph of G without nodes from $UnsatNodes$;
   (we have that G' is a marking of G)
   (f) update G_t to the graph of traces of G' in G;

Procedure $updateUnsatNodes(G, UnsatNodes, V)$
Input: an "and-or" graph G and sets $UnsatNodes$, V of nodes of G,
where V contains new *unsat* nodes.
Output: a new set $UnsatNodes$.

1. $UnsatNodes := UnsatNodes \cup V$;
2. while V is not empty do:
   (a) remove a node v from V;
   (b) for every father node u of v, if $u \notin UnsatNodes$ and either u is an "and"-node or u is an "or"-node and all the successor nodes of u belong to $UnsatNodes$ then add u to both $UnsatNodes$ and V;

Fig. 1. An algorithm for checking *L*-satisfiability of X w.r.t. Γ

only when the root v_0 belongs to $UnsatNodes$, i.e., only when G does not have any consistent marking. At Step 7b, G' is a marking of G that satisfies the local consistency property. If at that step $V = \emptyset$ then it satisfies also the global consistency property and is thus a consistent marking of G. That is, the algorithm returns *true* only when G has a consistent marking. Therefore, by Theorem 3.1, Algorithm 1 is a decision procedure for the considered problem. See [24] for a complexity analysis of the algorithm (for a sketch of which, just note that the "and-or" graph is constructed using global caching, and the contents of each node are a subset of $cl_L(X \cup \Gamma)$). ◁

5.2 Optimizations

Observe that Algorithm 1 first constructs an "and-or" graph and then checks whether the graph contains a consistent marking. To speed up the performance these two tasks can be done concurrently. For this we update the structures $UnsatNodes$, G', G_t of the algorithm "on-the-fly" during the construction of G. The main changes are as follows:

- During the construction of the "and-or" graph G, each node of G has status *unexpanded*, *expanded*, *unsat* or *sat*. The initial status of a new node is *unexpanded*. When a node is expanded, we change its status to *expanded*. The status of a node changes to *unsat* (resp. *sat*) when there is an evidence that the contents of the node are unsatisfiable (resp. satisfiable) w.r.t. Γ. When a node becomes *unsat*, we insert it into the set $UnsatNodes$.
- When a node of G is expanded or G' is modified, we update G_t appropriately.
- When a new node is created, if its contents contain $\bot$ or a clashing pair $\varphi, \overline{\varphi}$ then we change the status of the node to *unsat*. This is the implicit application of the rule $(\bot_0)$ and a generalized form of the rule $(\bot)$. Thus, we can drop the explicit rules $(\bot_0)$ and $(\bot)$. When a non-empty set V of nodes of G becomes *unsat*, we call $updateUnsatNodes(G, UnsatNodes, V)$ to update the set $UnsatNodes$.
- When $UnsatNodes$ is modified, we update G' appropriately.
- Since G_t is not completed during the construction, when computing the set V of nodes of G' that cause G' not satisfying the global consistency property as in Step 7a of Algorithm 1 we treat a node (v, φ) of G_t also as an *end-node* if v has status *unexpanded* or *sat*.[10] We compute such a set V occasionally, accordingly to some criteria, and when G_t has been completed. The computation is done by propagating "productiveness" backward through the graph G_t. The nodes of the resulting V become $unsat$.

During the construction of the "and-or" graph G, if a subgraph of G has been fully expanded in the sense that none of its nodes has status *unexpanded* or has a descendant node with status *unexpanded* then each node of the subgraph can be determined to be *unsat* or *sat* regardlessly of the rest of G. That is, if a node of the subgraph cannot be determined to be $unsat$ by the operations described in the above list then we can set its status to sat. This technique was proposed in [20].

Recently, the first author has implemented a tableau prover called TGC (Tableau with Global Caching) [20] for checking consistency of a concept w.r.t. a TBox in the description logic $\mathcal{ALC}$. He has developed and implemented for TGC a special set of optimizations that co-operates very well with global caching and various search strategies on search spaces of the form "and-or" graph. Apart from search strategies and global caching for nodes of the constructed "and-or" graph, TGC also uses other optimizations like normalizing formulas, caching formulas using an efficient catalogue, simplification, semantic branching, propagation of *unsat* in a local scale using *unsat*-cores and subset-checking for parent nodes and brother nodes, as well as cutoffs. The test results of TGC on the sets T98-sat and T98-kb of DL'98 Systems Comparison are comparable with the

[10] Note that if v has status *unexpanded* (resp. *sat*) then (v, φ) may (resp. must) be a productive node of G_t.

test results of the best systems DLP-98 and FaCT-98 that took part in that comparison (see [20]). One can say that the mentioned test sets are not representative for practical applications, but the comparison at least shows that various optimization techniques can be applied together with global caching to significantly increase efficiency of tableau decision procedures for modal and description logics.

Most of the optimization techniques of TGC can be applied for our decision procedure for REG^c logics. There remains, however, the problem of cuts (not only of our calculus), as they can make the search space very large. Despite that the applicability of our rule (cut) is quite restricted, the rule is inflexible. It is possible that one can work out a more sophisticated condition for the applicability of the rule (cut). On the implementation level, we hope that depth-first search together with propagation of *unsat* for parent/brother nodes and cutoffs significantly reduces the negative side effects of cuts. If this is not the case, one can try to delay cuts in an appropriate way (preserving completeness).

6 Conclusions

In this paper we have provided a sound and complete tableau calculus for deciding the general satisfiability problem of REG^c logics. The result is novel, since up to now no tableau calculi have been fully developed for REG^c logics. Using the calculus we have provided a complexity-optimal tableau decision procedure for the mentioned problem, to which a number of useful optimization techniques can be applied.

Extending our method, in the full version of this paper [24] we also develop an EXPTIME tableau decision procedure not based on transformation for the problem of checking consistency of an ABox w.r.t. a TBox in a REG^c logic. In [24] we also prove a new result that the data complexity of the instance checking problem in REG^c logics is coNP-complete.

Acknowledgements. We would like to thank the reviewers for their useful comments.

References

1. Baader, F., Sattler, U.: An overview of tableau algorithms for description logics. Studia Logica 69, 5–40 (2001)
2. Baldoni, M., Giordano, L., Martelli, A.: A tableau for multimodal logics and some (un)decidability results. In: de Swart, H. (ed.) TABLEAUX 1998. LNCS, vol. 1397, pp. 44–59. Springer, Heidelberg (1998)
3. De Giacomo, G., Massacci, F.: Combining deduction and model checking into tableaux and algorithms for Converse-PDL. Information and Computation 117-137, 87–138 (2000)
4. Demri, S.: The complexity of regularity in grammar logics and related modal logics. Journal of Logic and Computation 11(6), 933–960 (2001)
5. Demri, S., de Nivelle, H.: Deciding regular grammar logics with converse through first-order logic. arXiv:cs.LO/0306117 (2004)
6. Demri, S., de Nivelle, H.: Deciding regular grammar logics with converse through first-order logic. Journal of Logic, Language and Information 14(3), 289–329 (2005)

7. Donini, F., Massacci, F.: ExpTime tableaux for $\mathcal{ALC}$. Artificial Intelligence 124, 87–138 (2000)
8. Fariñas del Cerro, L., Penttonen, M.: Grammar logics. Logique et Analyse 121-122, 123–134 (1988)
9. Fitting, M.: Proof Methods for Modal and Intuitionistic Logics. Synthese Library, vol. 169. D. Reidel, Dordrecht (1983)
10. Goré, R.: Tableau methods for modal and temporal logics. In: D'Agostino, et al. (eds.) Handbook of Tableau Methods, pp. 297–396. Kluwer, Dordrecht (1999)
11. Goré, R., Nguyen, L.A.: A tableau system with automaton-labelled formulae for regular grammar logics. In: Beckert, B. (ed.) TABLEAUX 2005. LNCS (LNAI), vol. 3702, pp. 138–152. Springer, Heidelberg (2005)
12. Goré, R., Nguyen, L.A.: ExpTime tableaux with global caching for description logics with transitive roles, inverse roles and role hierarchies. In: Olivetti, N. (ed.) TABLEAUX 2007. LNCS (LNAI), vol. 4548, pp. 133–148. Springer, Heidelberg (2007)
13. Goré, R., Nguyen, L.A.: Analytic cut-free tableaux for regular modal logics of agent beliefs. In: Sadri, F., Satoh, K. (eds.) CLIMA VIII 2007. LNCS (LNAI), vol. 5056, pp. 268–287. Springer, Heidelberg (2008)
14. Goré, R., Nguyen, L.A.: Sound global caching for abstract modal tableaux. In: Burkhard, H.-D., et al. (eds.) Proceedings of CS&P 2008, pp. 157–167 (2008)
15. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge (2000)
16. Horrocks, I., Kutz, O., Sattler, U.: The even more irresistible SROIQ. In: Doherty, P., Mylopoulos, J., Welty, C.A. (eds.) Proceedings of KR 2006, pp. 57–67. AAAI Press, Menlo Park (2006)
17. Horrocks, I., Patel-Schneider, P.F.: Optimizing description logic subsumption. Journal of Logic and Computation 9(3), 267–293 (1999)
18. Horrocks, I., Sattler, U.: Decidability of $\mathcal{SHIQ}$ with complex role inclusion axioms. Artificial Intelligence 160(1-2), 79–104 (2004)
19. Mateescu, A., Salomaa, A.: Formal languages: an introduction and a synopsis. In: Handbook of Formal Languages, vol. 1, pp. 1–40. Springer, Heidelberg (1997)
20. Nguyen, L.A.: An efficient tableau prover using global caching for the description logic $\mathcal{ALC}$. In: Proceedings of CS&P (2008) (to appear in Fundamenta Informaticae)
21. Nguyen, L.A.: Analytic tableau systems and interpolation for the modal logics KB, KDB, K5, KD5. Studia Logica 69(1), 41–57 (2001)
22. Nguyen, L.A.: On the deterministic Horn fragment of test-free PDL. In: Hodkinson, I., Venema, Y. (eds.) Advances in Modal Logic, vol. 6, pp. 373–392. King's College Publications (2006)
23. Nguyen, L.A.: Weakening Horn knowledge bases in regular description logics to have PTIME data complexity. In: Ghilardi, et al. (eds.) Proceedings of ADDCT 2007, pp. 32–47 (2007),
`http://www.mimuw.edu.pl/~nguyen/papers.html`
24. Nguyen, L.A., Szałas, A.: ExpTime tableau decision procedures for regular grammar logics with converse (2009) (manuscript),
`http://www.mimuw.edu.pl/~nguyen/creg-long.pdf`
25. Pratt, V.R.: A near-optimal method for reasoning about action. J. Comput. Syst. Sci. 20(2), 231–254 (1980)
26. Rautenberg, W.: Modal tableau calculi and interpolation. JPL 12, 403–423 (1983)

An Optimal On-the-Fly Tableau-Based Decision Procedure for PDL-Satisfiability

Rajeev Goré[1] and Florian Widmann[2]

[1] Logic and Computation Group, The Australian National University
Canberra, ACT 0200, Australia
Rajeev.Gore@anu.edu.au

[2] Logic and Computation Group and NICTA* The Australian National University
Canberra, ACT 0200, Australia
Florian.Widmann@anu.edu.au

Abstract. We give an optimal (EXPTIME), sound and complete tableau-based algorithm for deciding satisfiability for propositional dynamic logic. Our main contribution is a sound method to track unfulfilled eventualities "on the fly" which allows us to detect "bad loops" sooner rather than in multiple subsequent passes. We achieve this by propagating and updating the "status" of nodes throughout the underlying graph as soon as is possible. We give sufficient details to enable an easy implementation by others. Preliminary experimental results from our unoptimised OCaml implementation indicate that our algorithm is feasible.

1 Introduction

Propositional dynamic logic (PDL) is an important logic for reasoning about programs [1]. Its formulae consist of traditional Boolean formulae plus "action modalities" built from a finite set of atomic programs using sequential composition ($;$), non-deterministic choice ($\cup$), repetition ($*$), and test ($?$). The satisfiability problem for PDL is EXPTIME-complete [2]. Unlike EXPTIME-complete description logics with algorithms exhibiting good average-case behaviour, no decision procedures for PDL-satisfiability are satisfactory from both a theoretical (soundness, completeness, optimality) and practical (average case behaviour) viewpoint, as we briefly explain next.

Fischer and Ladner's method [1] for PDL is impractical because it first constructs the set of all consistent subsets of the set of all subformulae of the given formula, which always requires exponential time. Pratt's optimal method [2] for PDL initially builds a "pseudo-model" (graph) and then checks whether the graph is a real model by making multiple passes that prune inconsistent nodes, and prune nodes containing "eventualities" which cannot be fulfilled by the current graph. Since an eventuality is detected as unfulfilled only in the pruning phase, it can do needless work, as we show shortly. LoTREC, which is primarily an educational tool, implements such a multi-pass method for PDL, but it

* NICTA is funded by the Australian Government's Department of Communications, Information Technology and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Centre of Excellence program.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 437–452, 2009.

is suboptimal (2EXPTIME) because it treats disjunctions naively. Baader's [3] tableau-based decision procedure for essentially PDL without "test" is suboptimal (2EXPTIME) and we cannot see how to extend it to "test". DLP implements this method restricted to test-free formulae where $*$ applies only to atomic programs (`http://ect.bell-labs.com/who/pfps/dlp/`). De Giacomo and Massacci [4] give a 2EXPTIME algorithm for deciding converse PDL-satisfiability, discuss ways to obtain optimality, but do not give an actual EXPTIME algorithm. The prover pdl-tableau (`http://www.cs.man.ac.uk/~schmidt/pdl-tableau/`) implements a variation of this method restricted to formulae without nested star.

Theorem provers based on optimal automata-based methods [5] are still in their infancy because good optimisations are not known [6]. Optimal game-theoretic methods for fix-point logics are known [7], but the proof of decidability relies heavily on non-determinism since the main goal is to prove a complexity bound. Brünnler and Lange [8] give "focused" cut-free sequent calculi based on these games, give proofs for PLTL and CTL in detail, and state without giving details that their calculi extend to PDL. The obvious decision procedures obtainable from their completeness proofs are suboptimal (2EXPTIME) since their underlying structure is a tree. We know of no resolution methods for PDL.

Here, we give an optimal, sound and complete tableau-based decision procedure for PDL-satisfiability. Our main contribution is a sound method to track unfulfilled eventualities "on the fly" which allows us to detect "bad loops" sooner rather than in multiple subsequent passes. Essentially, we interleave the graph-building and graph-pruning phases of Pratt's method by propagating and updating the "status" of nodes throughout the underlying graph as soon as is possible, significantly extending a similar method for description logic ALC [9]. The additional technicalities are non-trivial. We present pseudo code rather than traditional tableau rules because the "on the fly" nature of our algorithm gives it a non-local flavour. Thus a set of traditional local tableau "completion rules" would be cluttered by side-conditions to enforce the non-local aspects or would require a complicated strategy of rule applications. Preliminary experimental results from our unoptimised OCaml implementation (`http://rsise.anu.edu.au/~rpg/PDLGraphProver/`) indicate that our algorithm is feasible. Further work is to add the extensive array of optimisations which have proved successful for practical tableau-based methods for description logics.

To see how Pratt's method can do needless work, consider a formula $\langle a\rangle\varphi\wedge\langle b\rangle\psi$ where φ is explored first, φ is unsatisfiable because of some unfulfillable eventuality, and ψ is a huge formula. Pratt's method can only recognise unfulfillable eventualities in the pruning phase. Thus it *must* expand ψ even though it is unnecessary. By detecting unfulfilled eventualities "on the fly", our algorithm can recognise that φ is unsatisfiable before exploring ψ.

2 Syntax and Semantics

Definition 1. *Let* AFml *and* APrg *be two disjoint and countably infinite sets of propositional variables and* atomic programs*, respectively. The set* Fml *of all*

formulae and the set Prg *of all* programs *are defined mutually inductively as follows where* $a \in \mathrm{APrg}$ *and* $p \in \mathrm{AFml}$*:*

$$\begin{array}{ll} \mathrm{Prg} & \gamma ::= a \mid \gamma;\gamma \mid \gamma \cup \gamma \mid \quad \gamma* \quad \mid \quad \varphi? \\ \mathrm{Fml} & \varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle\gamma\rangle\varphi \mid [\gamma]\varphi \ . \end{array}$$

A $\langle \mathrm{ap} \rangle$*-formula is any formula* $\langle\gamma\rangle\varphi$ *where* $\gamma \in \mathrm{APrg}$ *is an atomic program.*

Implication ($\rightarrow$) and equivalence ($\leftrightarrow$) are not part of the core language but can be defined as usual. The size of a formula or a program is defined inductively by adding the sizes of its direct subformulae and subprograms and adding one. In the rest of the paper, let $p \in \mathrm{AFml}$ and $a \in \mathrm{APrg}$.

Definition 2. *A* transition frame *is a pair* (W, R) *where* W *is a non-empty set of worlds and* $R : \mathrm{APrg} \rightarrow W \times W$ *is a function mapping each atomic program* $a \in \mathrm{APrg}$ *to a binary relation* R_a *over* W*. A* model (W, R, V) *is a transition frame* (W, R) *and a valuation function* $V : \mathrm{AFml} \rightarrow 2^W$ *mapping each propositional variable* $p \in \mathrm{AFml}$ *to a set* $V(p)$ *of worlds.*

Definition 3. *Let* $M = (W, R, V)$ *be a model. The functions* $\tau_M : \mathrm{Fml} \rightarrow 2^W$ *and* $\rho_M : \mathrm{Prg} \rightarrow 2^{W \times W}$ *are defined inductively as follows:*

$$\begin{array}{lll} \tau_M(p) & := V(p) & \tau_M(\neg\varphi) := W \setminus \tau_M(\varphi) \\ \tau_M(\varphi \wedge \psi) & := \tau_M(\varphi) \cap \tau_M(\psi) & \tau_M(\varphi \vee \psi) := \tau_M(\varphi) \cup \tau_M(\psi) \\ \tau_M(\langle\gamma\rangle\varphi) & := \{w \mid \exists v \in W.\ (w,v) \in \rho_M(\gamma)\ \&\ v \in \tau_M(\varphi)\} & \\ \tau_M([\gamma]\varphi) & := \{w \mid \forall v \in W.\ (w,v) \in \rho_M(\gamma)\ \Rightarrow\ v \in \tau_M(\varphi)\} & \\ \rho_M(a) & := R_a & \\ \rho_M(\gamma \cup \delta) & := \rho_M(\gamma) \cup \rho_M(\delta) & \rho_M(\varphi?) := \{(w,w) \mid w \in \tau_M(\varphi)\} \\ \rho_M(\gamma;\delta) & := \{(w,v) \mid \exists u \in W.\ (w,u) \in \rho_M(\gamma)\ \&\ (u,v) \in \rho_M(\delta)\} & \\ \rho_M(\gamma*) & := \big\{(w,v) \mid \exists k \in \mathbb{N}_0 . \exists w_0, \ldots, w_k \in W.\ \big(w_0 = w\ \&\ w_k = v\ \& & \\ & \qquad \forall i \in \{0, \ldots, k-1\}.\ (w_i, w_{i+1}) \in \rho_M(\gamma)\big)\big\}\ . & \end{array}$$

For $w \in W$ *and* $\varphi \in \mathrm{Fml}$*, we write* $M, w \Vdash \varphi$ *iff* $w \in \tau_M(\varphi)$*.*

Definition 4. *A formula* $\varphi \in \mathrm{Fml}$ *is* satisfiable *iff there exists a model* $M = (W, R, V)$ *and a world* $w \in W$ *such that* $M, w \Vdash \varphi$*. A formula* $\varphi \in \mathrm{Fml}$ *is* valid *iff* $\neg\varphi$ *is unsatisfiable.*

Definition 5. *A formula* $\varphi \in \mathrm{Fml}$ *is in* negation normal form *if the symbol* $\neg$ *appears only immediately before propositional variables. For every* $\varphi \in \mathrm{Fml}$*, we can obtain a formula* $\mathrm{nnf}(\varphi)$ *in negation normal form by pushing negations inward as far as possible such that* $\varphi \leftrightarrow \mathrm{nnf}\,\varphi$ *is valid. We define* $\sim\varphi := \mathrm{nnf}(\neg\varphi)$*.*

We categorise formulae as α- or β-formulae as shown in Table 1.

Proposition 6. *In the notation of Table 1, the formulae of the form* $\alpha \leftrightarrow \alpha_1 \wedge \alpha_2$ *and* $\beta \leftrightarrow \beta_1 \vee \beta_2$ *are valid.*

Definition 7. *For a given* $\varphi \in \mathrm{Fml}$ *the (infinite) set* $\mathrm{pre}(\varphi)$ *is defined as below. Using it, we define the set* Ev *of all* eventualities *as:*

$$\mathrm{pre}(\varphi) := \{\psi \in \mathrm{Fml} \mid \exists k \in \mathbb{N}_0.\ \exists \gamma_1, \ldots, \gamma_k \in \mathrm{Prg}.\ \psi = \langle\gamma_1\rangle \ldots \langle\gamma_k\rangle\varphi\}$$

$$\mathrm{Ev} := \bigcup_{\varphi \in \Delta} \mathrm{pre}(\varphi) \ \textit{where}\ \Delta := \{\langle\gamma*\rangle\psi \mid \gamma \in \mathrm{Prg}\ \&\ \psi \in \mathrm{Fml}\}\ .$$

Table 1. Smullyan's α- and β-notation to classify formulae

α	$\varphi \wedge \psi$	$[\gamma \cup \delta]\varphi$	$[\gamma*]\varphi$	$\langle\psi?\rangle\varphi$	$\langle\gamma;\delta\rangle\varphi$	$[\gamma;\delta]\varphi$
α_1	φ	$[\gamma]\varphi$	φ	φ	$\langle\gamma\rangle\langle\delta\rangle\varphi$	$[\gamma][\delta]\varphi$
α_2	ψ	$[\delta]\varphi$	$[\gamma][\gamma*]\varphi$	ψ		

β	$\varphi \vee \psi$	$\langle\gamma \cup \delta\rangle\varphi$	$\langle\gamma*\rangle\varphi$	$[\psi?]\varphi$
β_1	φ	$\langle\gamma\rangle\varphi$	φ	φ
β_2	ψ	$\langle\delta\rangle\varphi$	$\langle\gamma\rangle\langle\gamma*\rangle\varphi$	$\sim\psi$

Definition 8. *Let X and Y be sets. We define $X^? := X \uplus \{\bot\}$ where $\bot$ indicates the undefined value. If $f : X \to Y$ is a function and $x \in X$ and $y \in Y$ then the function $f[x \mapsto y] : X \to Y$ is defined as $f[x \mapsto y](x') := y$ if $x' = x$ and $f[x \mapsto y](x') := f(x')$ if $x' \neq x$.*

3 An Overview and Our Algorithm

Our algorithm starts at a root containing a given formula ϕ and builds an and-or tree in a depth-first and left to right manner to try to build a model for ϕ. The rules are based on the semantics of PDL and either add formulae to the current world, or create a new world in the underlying model and add the appropriate formulae to it. For a node x, the attribute Γ_x carries this set of formulae.

The strategy for rule applications is the usual one where we "saturate" a node using the α/β-rules until they are no longer applicable, giving a "state" node s, and then, for each $\langle a\rangle\xi$ in s, we create an a-successor node containing $\{\xi\} \cup \Delta$, where $\Delta = \{\psi \mid [a]\psi \in s\}$. These successors are saturated to produce new states using the α/β-rules, and we create the successors of these new states, and so on.

Our strategy can produce infinite branches as the same node can be created repeatedly on the same branch. We therefore "block" a node from being created if this node exists already on any previous branch. For example, in Fig. 1, if the node y' already exists in the tree, say as node y, then we create a "backward" edge from x to y (as shown) and do not create y'. If y' does not duplicate an existing node then we create y' and add a "forward" edge from x to y'. Thus our tableau is a tree of forward edges, with backward edges that either point upwards from a node to a "forward-ancestor", or point leftwards from one branch to another. Cycles can arise only via backward edges to a forward-ancestor.

Our tableau must "fulfil" every formula of the form $\langle\delta\rangle\varphi$ in a node but only eventualities, as defined in Def. 7, cause problems. If $\langle\delta\rangle\varphi$ is not an eventuality, the α/β-rules reduce the size of the principal formula, ensuring fulfilment. If $\langle\delta\rangle\varphi$ is an eventuality, the main problem is the β-rule for formulae of the form $\langle\gamma*\rangle\varphi$. Its left child reduces $\langle\gamma*\rangle\varphi$ to a strict subformula φ, but the right child "reduces" it to $\langle\gamma\rangle\langle\gamma*\rangle\varphi$. If the left child is always inconsistent, this rule can "procrastinate" an eventuality $\langle\gamma*\rangle\varphi$ indefinitely and never find a world which makes φ true. This non-local property must be checked globally by tracking eventualities.

Consider Fig. 1, and suppose the current node x contains an eventuality e_x. We distinguish three cases. The first is that some path from x fulfils e_x in the existing tree. Else, the second case is that some path from x always procrastinates the fulfilment of e_x and hits a forward-ancestor of x on the current branch: *e.g.* the path x, y, v, u, w, z. The forward-ancestor z contains some "reduction" e_z of e_x.

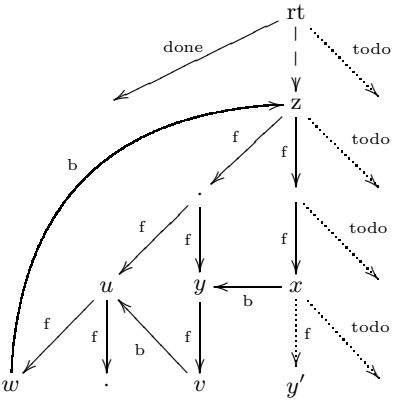


Fig. 1. Tree constructed by our algorithm using forward (f) and backward edges (b)

The path from the root to the current node x contains the only currently existing nodes which may need further expansion, and may allow z to fulfil e_z at a later stage, and hence fulfil e_x. We call the pair (z, e_z) a "potential rescuer" of e_x in Γ_x. The only remaining case is that $e_x \in \Gamma_x$ is unfulfilled, has no potential rescuers, and hence can never become fulfilled later, so x can be "closed". The machinery to distinguish these three cases and compute, if needed, all currently existing potential rescuers of every eventuality in Γ_x is described next.

A tableau node x also contains a status sts_x. The value of sts_x is the constant `closed` if the node x is closed. Otherwise, the node is "open" and sts_x contains a function prs which maps each eventuality $e_x \in \Gamma_x$ to $\perp$ or to a set of pairs (v, e) where v is a forward-ancestor of x and e is an eventuality. The status of a node is determined from those of its children once they have all been processed. A closed child's status is propagated as usual, but the propagation of the function prs from open children is more complicated. We give details later, but the intuition is that we must preserve the following invariant for each eventuality $e_x \in \Gamma_x$:

> if e_x is fulfilled in the tree to the left of the path from the root to the node x then $\mathrm{prs}_x(e_x) := \perp$, else $\mathrm{prs}_x(e_x)$ is exactly the set of all potential rescuers of e_x in the current tableau.

An eventuality $e_x \in \Gamma_x$ whose $\mathrm{prs}_x(e_x)$ becomes the empty set can never become fulfilled later, so $\mathrm{sts}_x := \texttt{closed}$, thus covering the three cases as desired.

Whenever a node n gets a status `closed`, we interrupt the depth-first and left-to-right traversal and invoke a separate procedure which explicitly propagates this status transitively throughout the and-or graph rooted at n. For example, if z gets closed then so will its backward-parent w, which may also close u and so on. This update may break the invariant for some eventuality e in this subgraph by interrupting the path from e to a node that fulfils e or to a potential rescuer of e. We must therefore ensure that the update procedure re-establishes the invariant in these cases by changing the appropriate prs entries. At the end of

the update procedure, we resume the usual depth-first and left-to-right traversal of the tree by returning the status of n to its forward-parent. This "on-the-fly" nature guarantees that unfulfilled eventualities are detected as early as possible.

When our algorithm terminates, formula ϕ is satisfiable iff the root is open.

3.1 The Algorithm

Our algorithm builds a directed graph G consisting of nodes and three types of directed edges: forward-, backward-, and update-edges. We first explain the structure of G in more detail.

Definition 9. *Let $G = (V, E)$ be a graph where V is a set of nodes and E is a set of directed edges. Each node $x \in V$ has three attributes: $\Gamma_x \subseteq \mathrm{Fml}$, $\mathrm{ann}_x : \mathrm{Ev} \to \mathrm{Fml}^?$, and $\mathrm{sts}_x \in \mathfrak{S}^?$ where $\mathfrak{S} := \{\texttt{closed}\} \cup \{\texttt{open}(\mathrm{prs}) \mid \mathrm{prs} : \mathrm{Ev} \to (\mathscr{P}(V \times \mathrm{Ev}))^?\}$. Each directed edge $e \in E$ is either a forward- or a backward- or an update-edge. If e is a forward- or backward-edge then it is labelled with a label $l_e \in \mathrm{Fml}^?$.*

When we say that x is a forward-ancestor of y we mean that x is an ancestor of y when only the forward-edges of G are considered. Similarly for other graph concepts like child and parent, and other edge types.

As indicated in Def. 9, some attributes of x may be undefined initially. Once an attribute becomes defined in x, however, it will never become undefined again.

The attributes Γ_x and ann_x of a node $x \in G$ are initialised at the creation of x and are not changed afterwards. Together, they uniquely identify x, so no two nodes in G have the same values for *both* attributes. The finite set Γ_x contains the formulae which are assigned to x. The attribute ann_x *annotates* each eventuality $\varphi \in \Gamma_x$, as long as it is not a $\langle \mathrm{ap} \rangle$-formula. The value $\mathrm{ann}_x(\varphi) = \bot$ indicates that φ is not expanded in x, and $\mathrm{ann}_x(\varphi) = \varphi'$ indicates that φ has already been "reduced" to $\varphi' \in \Gamma_x$. These annotations identify the fulfilling path for eventualities. Note that ann_x is defined only for some eventualities in (the finite set) Γ_x. Hence we can test whether ann_x and ann_y (for $y \in V$) are equal.

The attribute sts_x describes the *status* of x. It is initially undefined but becomes defined during the algorithm. Unlike all other attributes, its value may be modified several times, but once it becomes `closed`, it will never change. The value `closed` indicates that the node is not "annotated satisfiable", an extension of satisfiability taking annotations into account. If no formula in a set is annotated, "annotated satisfiable" and satisfiable coincide. A value $\texttt{open}(\mathrm{prs}_x)$ indicates that hope still exists that x is "annotated satisfiable". The function prs_x contains information about each eventuality $\varphi \in \Gamma_x$ as explained in the overview. That is, $\mathrm{prs}_x(\varphi)$ is either $\bot$ or is the finite set of all *potential rescuers* for φ. If $(y, \psi) \in \mathrm{prs}_x(\varphi)$ then y is a forward-ancestor of x and $\psi \in \Gamma_y$. Like ann_x, the function prs_x is defined only for some eventualities in (the finite set) Γ_x.

As the algorithm proceeds, we might have to update sts_x. For example, if the status of a node $y \in G$ is changed to `closed`, an eventuality $\varphi \in \Gamma_x$ may no longer be fulfilled in G. Moreover, if $(y, \psi) \in \mathrm{prs}_x(\varphi)$ then (y, ψ) must be removed from $\mathrm{prs}_x(\varphi)$ since it now cannot help to fulfil φ and is therefore no

longer a potential rescuer of φ. If $\mathrm{prs}_x(\varphi)$ becomes empty, we know that φ cannot be fulfilled in G ever. Hence x is unsatisfiable and its status is set to `closed`.

We insert forward- and backward-edges between two nodes x and y as explained in the overview. Note that y might be a forward- and backward-child of x. In this case, the algorithm takes y as a forward-child of x. To track eventualities, we label a forward- or backward-edge between a state and its child by the $\langle \mathrm{ap} \rangle$-formula $\langle a \rangle \chi$ which creates this child. An update-edge from x to y indicates that a status change of x might affect the status of y.

Definition 10. *Let $x, y \in G$ be nodes and $\varphi \in \mathrm{Fml}$. We call x* closed *iff it has $\mathrm{sts}_x = \texttt{closed}$ and* open *iff it has $\mathrm{sts}_x = \texttt{open}(\mathrm{prs})$ for some $\mathrm{prs} : \mathrm{Ev} \to (\mathscr{P}(V \times \mathrm{Ev}))^?$. In the latter case, we define $\mathrm{prs}_x := \mathrm{prs}$.*

Note that a node is either open, closed, or has an undefined status. Thus "not closed" is not really equal to "open" as we pretended in the overview.

Definition 11. *Let $\mathrm{ann}^\perp : \mathrm{Ev} \to \mathrm{Fml}^?$ and $\mathrm{prs}^\perp : \mathrm{Ev} \to (\mathscr{P}(V \times \mathrm{Ev}))^?$ be the functions which are undefined everywhere. For a node $x \in V$ and a label $l \in \mathrm{Fml}^?$, let $\mathrm{getChild}(x, l)$ be the node $y \in V$ (if existent and unique) such that there exists a forward- or backward-edge $e \in E$ from x to y with $l_e = l$. For a function $\mathrm{prs} : \mathrm{Ev} \to (\mathscr{P}(V \times \mathrm{Ev}))^?$, a node $x \in V$, and an eventuality $\varphi \in \mathrm{Ev}$, we define the set $\mathrm{reach}(\mathrm{prs}, x, \varphi)$ of eventualities as follows:*

$$\mathrm{reach}(\mathrm{prs}, x, \varphi) := \Big\{ \psi \in \mathrm{Ev} \mid \exists k \in \mathbb{N}_0.\, \exists \varphi_0, \ldots, \varphi_k \in \mathrm{Ev}.\, \Big(\psi = \varphi_k \;\&\; (x, \varphi_0) \in \mathrm{prs}(\varphi) \;\&\; \forall i \in \{0, \ldots, k-1\}.\, (x, \varphi_{i+1}) \in \mathrm{prs}(\varphi_i) \Big) \Big\} \;.$$

The function $\mathrm{defer} : V \times \mathrm{Ev} \to \mathrm{Fml}^?$ is defined as follows:

$$\mathrm{defer}(x, \varphi) := \begin{cases} \psi & \text{if } \exists k \in \mathbb{N}_0.\, \exists \varphi_0, \ldots, \varphi_k \in \mathrm{Fml}.\, \Big(\varphi_0 = \varphi \;\&\; \varphi_k = \psi \;\&\; \forall i \in \{0, \ldots, k-1\}.\, \big(\varphi_i \in \mathrm{Ev} \;\&\; \mathrm{ann}_x(\varphi_i) = \varphi_{i+1} \big) \;\&\; \big(\varphi_k \notin \mathrm{Ev} \text{ or } \mathrm{ann}_x(\varphi_k) = \perp \big) \Big) \\ \perp & \text{otherwise.} \end{cases}$$

The function $\mathrm{getChild}(x, l)$ retrieves a particular forward- or backward-child of x. It is easy to see that we will only use it in the algorithm if it is well-defined.

Intuitively, the function $\mathrm{reach}(\mathrm{prs}, x, \varphi)$ computes all eventualities which can be "reached" from φ inside x according to prs. If a potential rescuer (x, ψ) is contained in $\mathrm{prs}(\varphi)$, the potential rescuers of ψ are somehow relevant for φ at x. Therefore ψ itself is relevant for φ at x. The function $\mathrm{reach}(\mathrm{prs}, x, \varphi)$ computes exactly the transitive closure of this relevance relation.

Intuitively, the function $\mathrm{defer}(x, \varphi)$ follows the "ann_x-chain". That is, it computes $\varphi_1 := \mathrm{ann}_x(\varphi)$, $\varphi_2 := \mathrm{ann}_x(\varphi_1)$, and so on. There are two possible outcomes. The first outcome is that we eventually encounter a φ_k which is either not an eventuality or has $\mathrm{ann}_x(\varphi_k) = \perp$. Consequently, we cannot follow the "ann_x-chain" any more. In this case we stop and return $\mathrm{defer}(x, \varphi) := \varphi_k$.

Procedure `is-sat`(ϕ) for testing whether a formula is satisfiable

Input: a formula $\phi \in$ Fml in negation normal form
Output: true iff ϕ is satisfiable

$G :=$ a new empty graph
$r :=$ `build-graph`$(\{\phi\}, \text{ann}^{\perp}, \perp, \perp)$
return $\text{sts}_r \neq$ `closed`

The second outcome is that we can follow the "ann_x-chain" indefinitely. Then, as Γ_x is finite, there must exist a cycle $\varphi_0, \ldots, \varphi_n, \varphi_0$ of eventualities such that $\text{ann}_x(\varphi_i) = \varphi_{i+1}$ for all $0 \leq i < n$, and $\text{ann}_x(\varphi_n) = \varphi_0$. In this case we say that x (or Γ_x) contains an *"at a world" cycle* and return $\text{defer}(x, \varphi) := \perp$.

Next we comment on all procedures given in pseudocode.

Procedure `is-sat`(ϕ) is invoked to determine whether a formula $\phi \in$ Fml in negation normal form is satisfiable. It initialises the global variable G as the empty graph and invokes `build-graph` with the singleton set $\{\phi\}$ and no annotations. This is the only invocation of `build-graph` that is not initiated while processing a node in G, so its final two arguments are $\perp$. It returns "satisfiable" iff the resulting node $r \in G$, with $\Gamma_r = \{\phi\}$, is not closed in the final graph.

Procedure `build-graph`$(\Gamma, \text{ann}, p, l)$ builds the graph G in a tree-like fashion as explained in the overview and calls the procedures which compute the status of the nodes. Remember that G is a global variable. The arguments of `build-graph` are a set Γ, an annotation ann, the parent node p which invoked it, and a label l. Note that p is undefined for the very first invocation. If there already exists a node x in G with Γ and ann, we insert a backward-edge labelled with l from p to x (if p is defined) and return x. Otherwise we create the desired node $x \in G$, insert a forward-edge labelled with l from p to x (if p is defined), and process x as described next. Each node has at most one forward-edge pointing to it.

If Γ_x contains an "at a world" cycle or a contradiction, we close x. For the other cases, we assume implicitly that Γ_x does not contain either of these.

If Γ contains an α-formula α whose decompositions are not in Γ, or which is an unannotated eventuality, we call x an *α-node*. We create a new set Γ' by adding all decompositions of α to Γ. If α is an eventuality, we also create a new annotation extending ann by mapping α to α_1. Then we invoke `build-graph` recursively and determine and set the status of x. Note that Γ' is strictly bigger than Γ or α is an eventuality which is annotated in ann$'$ but not in ann.

If x is not an α-node and Γ contains a β-formula β such that neither of its immediate subformulae is in Γ, or such that β is an unannotated eventuality, we call x a *β-node*. For each decomposition β_i we do the following. We create a new set Γ_i by adding β_i to Γ. If β is an eventuality, we also create a new annotation which extends ann by mapping β to β_i. Note that Γ_i is strictly bigger than Γ or β is an eventuality which is annotated in ann$'$ but not in ann. We then invoke `build-graph` recursively. In the end we determine and set the status of x.

If x is neither an α-node nor a β-node, it must be fully saturated and we call it a *state*. For each $\langle \text{ap} \rangle$-formula $\langle a_i \rangle \varphi_i$ we create a new set Γ_i which contains φ_i

Procedure `build-graph`$(\Gamma, \mathrm{ann}, p, l)$ for building the graph

Input: a set $\Gamma \subseteq \mathrm{Fml}$, a function $\mathrm{ann} : \mathrm{Ev} \to \mathrm{Fml}^?$, a node $p \in V^?$, and a label $l \in \mathrm{Fml}^?$
Output: a node $x \in V$

if $\exists x \in V.\, \Gamma_x = \Gamma$ & $\mathrm{ann}_x = \mathrm{ann}$ **then** (* annotated set already exists in G *)
  if $p \neq \bot$ **then** insert a backward-edge from p to x labelled with l in G
  return x
else (* annotated set not in G yet *)
  create new node x with $\Gamma_x := \Gamma$, $\mathrm{ann}_x := \mathrm{ann}$, and $\mathrm{sts}_x := \bot$
  insert x in G
  if $p \neq \bot$ **then** insert a forward-edge from p to x labelled with l in G
  if $\exists \varphi \in \mathrm{Fml}.(\varphi \in \mathrm{Ev}$ & $\mathrm{defer}(x, \varphi) = \bot)$ *or* $\{\varphi, \sim\varphi\} \subseteq \Gamma$ **then**
    $\mathrm{sts}_x :=$ `closed`
  else if $\exists \alpha \in \Gamma.\, \{\alpha_1, \ldots, \alpha_k\} \not\subseteq \Gamma$ *or* $(\alpha \in \mathrm{Ev}$ & $\mathrm{ann}(\alpha) = \bot)$ **then**
    $\Gamma' := \Gamma \cup \{\alpha_1, \ldots, \alpha_k\}$
    $\mathrm{ann}' :=$ **if** $\alpha \in \mathrm{Ev}$ **then** $\mathrm{ann}[\alpha \mapsto \alpha_1]$ **else** ann
    `build-graph`$(\Gamma', \mathrm{ann}', x, \bot)$
    $\mathrm{sts}_x :=$ `det-sts-`$\beta(x)$
  else if $\exists \beta \in \Gamma.\, \{\beta_1, \beta_2\} \cap \Gamma = \emptyset$ *or* $(\beta \in \mathrm{Ev}$ & $\mathrm{ann}(\beta) = \bot)$ **then**
    for $i \longleftarrow 1$ **to** 2 **do**
      $\Gamma_i := \Gamma \cup \{\beta_i\}$
      $\mathrm{ann}_i :=$ **if** $\beta \in \mathrm{Ev}$ **then** $\mathrm{ann}[\beta \mapsto \beta_i]$ **else** ann
      `build-graph`$(\Gamma_i, \mathrm{ann}_i, x, \bot)$
    $\mathrm{sts}_x :=$ `det-sts-`$\beta(x)$
  else (* x is a state *)
    let $\langle a_1\rangle\varphi_1, \ldots, \langle a_k\rangle\varphi_k$ be all of the $\langle\mathrm{ap}\rangle$-formulae in Γ
    for $i \longleftarrow 1$ **to** k **do**
      $\Gamma_i := \{\varphi_i\} \cup \{\psi \mid [a_i]\psi \in \Gamma\}$
      `build-graph`$(\Gamma_i, \mathrm{ann}^\bot, x, \langle a_i\rangle\varphi_i)$
    $\mathrm{sts}_x :=$ `det-sts-state`(x)
  if $\mathrm{sts}_x =$ `closed` **then**
    let $y_1, \ldots, y_k$ be all the nodes that are backward- or update-parents of x
    for $i \longleftarrow 1$ **to** k **do** `update`(y_i)
  return x

and all ψ such that $[a_i]\psi \in \Gamma$. We then invoke `build-graph` recursively. As none of the eventualities in Γ_i is expanded, there are no annotations. In order to relate the resulting node y to $\langle a_i\rangle\varphi_i$, we label the edge from x to y with $\langle a_i\rangle\varphi_i$. We call y the *successor* of $\langle a_i\rangle\varphi_i$. In the end we determine and set the status of x.

If x is closed, we update all nodes that depend on the status of x; except p, whose status is undefined and which will use the result later. Finally we return x. Note that if `build-graph` creates x via the main "else" then sts_x must be either open or closed but not $\bot$. In particular, this applies to node r in `is-sat`.

Procedure `det-sts-`$\beta(x)$ computes the status of an α- or a β-node $x \in G$. For this task, an α-node can be seen as a β-node with exactly one child. If all children of x are closed then x must also be closed. Otherwise we compute the set of potential rescuers for each eventuality φ in Γ_x as follows. For each open child y'_i

Procedure `det-sts-`$\beta(x)$ for determining the status of an α- or a β-node

Input: an α- or a β-node $x \in V$
Output: the new status of x

let $y_1, \ldots, y_k \in G$ be all the nodes that are forward- or backward-children of x
if $\forall i \in \{1, \ldots, k\}.\ \mathrm{sts}_{y_i} =$ `closed` **then return** `closed`
else (∗ at least one child is not closed ∗)
  let $y'_1, \ldots, y'_l$ $(1 \leq l \leq k)$ be all the children of x that are not closed
  $\mathrm{prs} := \mathrm{prs}^{\perp}$
  foreach $\varphi \in \Gamma_x \cap \mathrm{Ev}$ **do**
    for $i \longleftarrow 1$ **to** l **do**
      $\Lambda_{\varphi,i} :=$ **if** y'_i *is a forward-child of* x **then** $\mathrm{prs}_{y'_i}(\varphi)$
        else `det-prs-child`(x, y'_i, φ)
    $\Lambda_\varphi :=$ **if** $\exists i \in \{1, \ldots, l\}.\ \Lambda_{\varphi,i} = \perp$ **then** $\perp$ **else** $\bigcup_{i=1}^{l} \Lambda_{\varphi,i}$
    $\mathrm{prs} := \mathrm{prs}[\varphi \mapsto \Lambda_\varphi]$
  return `filter`(x, prs)

of x we determine the potential rescuers of φ which result from following y'_i. We do this by distinguishing whether y'_i is a forward- or backward-child of x. If y'_i is a forward-child of x then $\mathrm{prs}_{y'_i}(\varphi)$ is just passed on. If y'_i is a backward-child of x then we invoke `det-prs-child`. If the set of potential rescuers corresponding to some y'_i is $\perp$ then φ can currently be fulfilled via y'_i and $\mathrm{prs}_x(\varphi)$ is set to undefined. Otherwise φ cannot be fulfilled in G, but each child returned a set of potential rescuers, and the set of potential rescuers for φ is their union. Finally, we treat potential rescuers of the form (x, χ) for some $\chi \in \mathrm{Ev}$ by calling `filter`.

Procedure `det-sts-state`(x) computes the status of a state $x \in V$. We obtain the successors for all $\langle \mathrm{ap} \rangle$-formulae in Γ_x. If any successor is closed then x is closed. Else we compute the potential rescuers for each eventuality in Γ_x as follows. For each $\langle \mathrm{ap} \rangle$-formula $\langle a_i \rangle \varphi_i$ which is an eventuality, we obtain its set of potential rescuers by distinguishing whether its successor y_i is a forward- or backward-child of x. If y_i is a forward-child of x then $\mathrm{prs}_{y_i}(\varphi_i)$ is passed on to $\langle a_i \rangle \varphi_i$. If y is a backward-child of x, we invoke `det-prs-child`. For every other eventuality φ, we determine $\varphi' := \mathrm{defer}(x, \varphi)$. Note that φ' is defined because the state x cannot contain an "at a world" cycle by definition. If φ' is not an eventuality then φ is fulfilled in x and $\mathrm{prs}(\varphi)$ remains undefined. If φ' is an eventuality, it must be a $\langle \mathrm{ap} \rangle$-formula as x is a state, so we set $\mathrm{prs}(\varphi) := \mathrm{prs}(\varphi')$. Finally, we deal with potential rescuers in prs of the form (x, χ) for some $\chi \in \mathrm{Ev}$.

Procedure `det-prs-child`(x, y, φ) determines whether an eventuality $\psi \in \Gamma_x$, which is not passed as an argument, can be fulfilled via y such that φ is part of the corresponding fulfilling path; or else which potential rescuers ψ can reach via y and φ. We assume that y and φ can "play a part" in fulfilling ψ. First, if there is no edge from x to y in G, we insert an update-edge from x to y in G because a status change of y might affect the status of x. If y is closed, it cannot help to fulfil ψ as indicated by the empty set. If $x = y$ or y is a forward-ancestor of x then (y, φ) itself is a potential rescuer of x. Else, if φ can be

Procedure `det-sts-state`(x) for determining the status of a state

Input: a state $x \in V$
Output: the new status of x

let $\langle a_1\rangle\varphi_1, \ldots, \langle a_k\rangle\varphi_k$ be all of the $\langle\text{ap}\rangle$-formulae in Γ_x
for $i \longleftarrow 1$ **to** k **do** $y_i := \text{getChild}(x, \langle a_i\rangle\varphi_i)$
if $\exists i \in \{1, \ldots, k\}.\ \text{sts}_{y_i} =$ `closed` **then return** `closed`
else (* all children (if any) are not closed *)
  prs := $\text{prs}^{\perp}$
  for $i \longleftarrow 1$ **to** k **do**
    if $\varphi_i \in \text{Ev}$ **then**
      $\Lambda_i :=$ **if** y_i *is a forward-child of* x **then** $\text{prs}_{y_i}(\varphi_i)$
        else `det-prs-child`(x, y_i, φ_i)
      prs := $\text{prs}[\langle a_i\rangle\varphi_i \mapsto \Lambda_i]$
  foreach $\varphi \in \Gamma_x \cap \text{Ev}$ *such that* φ *is not a* $\langle\text{ap}\rangle$*-formula* **do**
    $\varphi' := \text{defer}(x, \varphi)$
    if $\varphi' \in \text{Ev}$ **then** prs := $\text{prs}[\varphi \mapsto \text{prs}(\varphi')]$
  return `filter`(x, prs)

fulfilled, i.e. $\text{prs}_y(\varphi) = \perp$, then ψ can be fulfilled too, so we return $\perp$. Otherwise we invoke the procedure recursively on all potential rescuers in $\text{prs}_y(\varphi)$. If at least one of these invocations returns $\perp$ then ψ can be fulfilled via y and φ and the corresponding rescuer in $\text{prs}_y(\varphi)$. If all invocations return a set of potential rescuers, the set of potential rescuers for ψ is their union.

Each invocation of `det-prs-child` can be uniquely assigned to the invocation of `det-sts-`β or `det-sts-state` which (possibly indirectly) invoked it. To meet our complexity bound, we require that under the same invocation of `det-sts-`β or `det-sts-state`, the procedure `det-prs-child` is only executed at most once for each argument triple. Instead of executing it a second time with the same arguments, it uses the cached result of the first invocation. The second invocation would return the same result and would not modify the graph.

Procedure `filter`(x, prs) deals with the potential rescuers for each eventuality of a node x which are of the form (x, ψ) for some $\psi \in \text{Ev}$. The second argument of `filter` is a provisional prs for x. If an eventuality $\varphi \in \Gamma_x$ is currently fulfillable in G there is nothing to be done, so let $(x, \psi) \in \text{prs}(\varphi)$. If $\psi = \varphi$ then (x, φ) cannot be a potential rescuer for φ in x and should not appear in $\text{prs}(\varphi)$. But what about potential rescuers of the form (x, ψ) with $\psi \neq \varphi$? Since we want the nodes in the potential rescuers to be strict forward-ancestors of x, we cannot keep (x, ψ) in $\text{prs}(\varphi)$; but we cannot just ignore them either.

Intuitively $(x, \psi) \in \text{prs}(\varphi)$ means that $\varphi \in \Gamma_x$ can "reach" $\psi \in \Gamma_x$ by following a loop in G which starts at x and returns to x itself. Thus if ψ can be fulfilled in G, so can φ; and all potential rescuers of ψ are also potential rescuers of φ. The function $\text{reach}(\text{prs}, x, \varphi)$ computes all eventualities in x which are "reachable" from φ in the sense above, where transitivity is taken into account. That is, it detects all self-loops from x to itself which are relevant for fulfilling φ. We add φ as it is not in $\text{reach}(\text{prs}, x, \varphi)$. If any of these eventualities is fulfilled in G then φ

Procedure `det-prs-child`(x, y, φ) for passing a prs-entry of a backward-child to a parent

Input: two nodes $x, y \in V$ such that $x = y$ or y is a forward-ancestor of x or the status of y is defined already; and a formula $\varphi \in \Gamma_y \cap \mathrm{Ev}$
Output: undefined or a set of node-formula pairs
Remark: if `det-prs-child`(x, y, φ) has already been invoked before with exactly the same arguments and *under the same invocation of* `det-sts-`β *or* `det-sts-state`, the procedure is not executed a second time but returns the cached result of the first invocation. We do not model this behaviour explicitly in the pseudocode.

if *there is no edge (of any type) from* x *to* y *in* G **then**
  insert an update-edge from x to y in G
if $\mathrm{sts}_y = $ `closed` **then return** $\emptyset$
else if y *is a forward-ancestor of* x *or* $y = x$ **then return** $\{(y, \varphi)\}$
else (* y is open because $\mathrm{sts}_y \neq$ `closed` and its status is defined already *)
  if $\mathrm{prs}_y(\varphi) = \bot$ **then return** $\bot$
  else (* $\mathrm{prs}_y(\varphi)$ is defined *)
    let $(z_1, \varphi_1), \ldots, (z_k, \varphi_k)$ be all of the pairs in $\mathrm{prs}_y(\varphi)$
    for $i \longleftarrow 1$ **to** k **do** $\Lambda_i :=$ `det-prs-child`(x, z_i, φ_i)
    if $\exists j \in \{1, \ldots, k\}.\ \Lambda_j = \bot$ **then return** $\bot$ **else return** $\bigcup_{i=1}^{k} \Lambda_i$

Procedure `filter`(x, prs) for handling self-loops in G

Input: a node $x \in V$ and a function $\mathrm{prs} : \mathrm{Ev} \to (\mathscr{P}(V \times \mathrm{Ev}))^?$
Output: the new status of x

$\mathrm{prs}' := \mathrm{prs}^{\bot}$
foreach $\varphi \in \Gamma_x \cap \mathrm{Ev}$ *such that* $\mathrm{prs}(\varphi) \neq \bot$ **do**
  $\Delta := \{\varphi\} \cup \mathrm{reach}(\mathrm{prs}, x, \varphi)$
  if *not* $\exists \chi \in \Delta.\ \mathrm{prs}(\chi) = \bot$ **then**
    $\Lambda := \bigcup_{\chi \in \Delta} \{(z, \psi) \in \mathrm{prs}(\chi) \mid z \neq x\}$
    $\mathrm{prs}' := \mathrm{prs}'[\varphi \mapsto \Lambda]$
if $\exists \varphi \in \Gamma_x \cap \mathrm{Ev}.\ \mathrm{prs}'(\varphi) = \emptyset$ **then return** `closed` **else return** `open`(prs')

Procedure `update`(x) for propagating the status of nodes

Input: a node $x \in V$ that has a defined status

if $\mathrm{sts}_x \neq$ `closed` **then**
  $\mathrm{sts} :=$ **if** x *is an* α- *or a* β-*node* **then** `det-sts-`$\beta(x)$ **else** `det-sts-state`(x)
  if $\mathrm{sts}_x \neq \mathrm{sts}$ **then**
    $\mathrm{sts}_x := \mathrm{sts}$
    let $y_1, \ldots, y_k$ be all the nodes that are forward-, backward- or update-parents of x
    for $i \longleftarrow 1$ **to** k **do** `update`(y_i)

can be fulfilled and is consequently undefined in the resulting prs′. Otherwise we take all their potential rescuers which contain proper forward-ancestors of x.

If an eventuality $\varphi \in \Gamma_x$ has the empty set of potential rescuers, x is closed.

Procedure `update`(x) propagates status changes through G. It recomputes the status of a node $x \in V$. If the new status differs from its old one, it updates sts_x and invokes `update` recursively on all nodes whose status may be affected by this change. A node that is not closed is either an α/β-node or a state.

Theorem 12 (Soundness, Completeness and Termination). *Let $\phi \in$ Fml be a formula in negation normal form of size n. The procedure* `is-sat`(ϕ) *terminates, runs in* EXPTIME *in n, and ϕ is satisfiable iff* `is-sat`(ϕ) *returns true.*

4 A Fully Worked Example

Consider the valid formula $\langle(a*;b*)*\rangle p \rightarrow \langle(a \cup b)*\rangle p$. The full tableau for its negation does not fit on one page, but its core subgraph is the tableau for the unsatisfiable formula $\phi := \langle b\rangle\langle b*\rangle\langle(a*;b*)*\rangle p \wedge [(a \cup b)*]\neg p$. We therefore consider the tableau for ϕ. To save space, we use the definitions in Table 2.

Figure 2 (almost) shows the corresponding graph G just before setting the status of node (2). To save space, we (recursively) perform multiple α-expansions inside nodes. Thus, there are no α-nodes in G. For example, the root node contains ϕ, as well as its decompositions $\langle b\rangle\langle b*\rangle\varphi_1$ and $[(a \cup b)*]\neg p$, the decompositions of $[(a \cup b)*]\neg p$, and so on. Incidentally, Δ, and in particular $\{\neg p\}$, is a subset of the Γ-components of all nodes, reflecting the semantics of $[(a \cup b)*]\neg p$.

The function P_{13} maps φ_3 and $\langle a\rangle\langle a*\rangle\langle b*\rangle\varphi_1$ to $\{(8, \varphi_3)\}$ and is undefined elsewhere. All other P_i in Fig. 2 map each eventuality in their corresponding set to $\{(2, \langle b*\rangle\varphi_1)\}$ and are undefined elsewhere. The nodes are labelled in creation order. The annotation ann is given using "$\rightsquigarrow$" in Γ. For example, in node (3), we have $\Gamma_3 = \{\ \varphi_4,\ \langle(a*;b*)*\rangle p\ \} \cup \Delta$, and ann_3 maps the eventuality φ_4 to $\langle(a*;b*)*\rangle p$ and is undefined elsewhere. The bottom line of a node contains its status. Solid arrows represent forward-edges and dashed arrows represent backward-edges. There are no update-edges in this example. The label of a forward- and backward-edge is only given if it is a formula and not $\bot$.

Nodes (1), (2), (3), and (4) are created first, and (4) is closed because it contains p and $\neg p$. Then (5) and (6) are created, but (6) is closed as it contains an "at a world" cycle. Next, nodes (7) to (11) are created. Like (4), node (11) is closed because of a contradiction. When trying to create the second child of (10), we find that the requested node is already contained in G as (6). Hence we insert a backward-edge from (10) to (6). Since both children of (10) are closed, node (10)

Table 2. Some definitions used in the example

$\varphi_1 := \langle(a*;b*)*\rangle p$	$\varphi_2 := \langle a*;b*\rangle\varphi_1$
$\varphi_3 := \langle a*\rangle\langle b*\rangle\varphi_1$	$\varphi_4 := \langle b*\rangle\varphi_1$
$\Delta := \{\ [(a \cup b)*]\neg p,\ p,\ [a \cup b][(a \cup b)*]\neg p,\ [a][(a \cup b)*]\neg p,\ [b][(a \cup b)*]\neg p\ \}$	

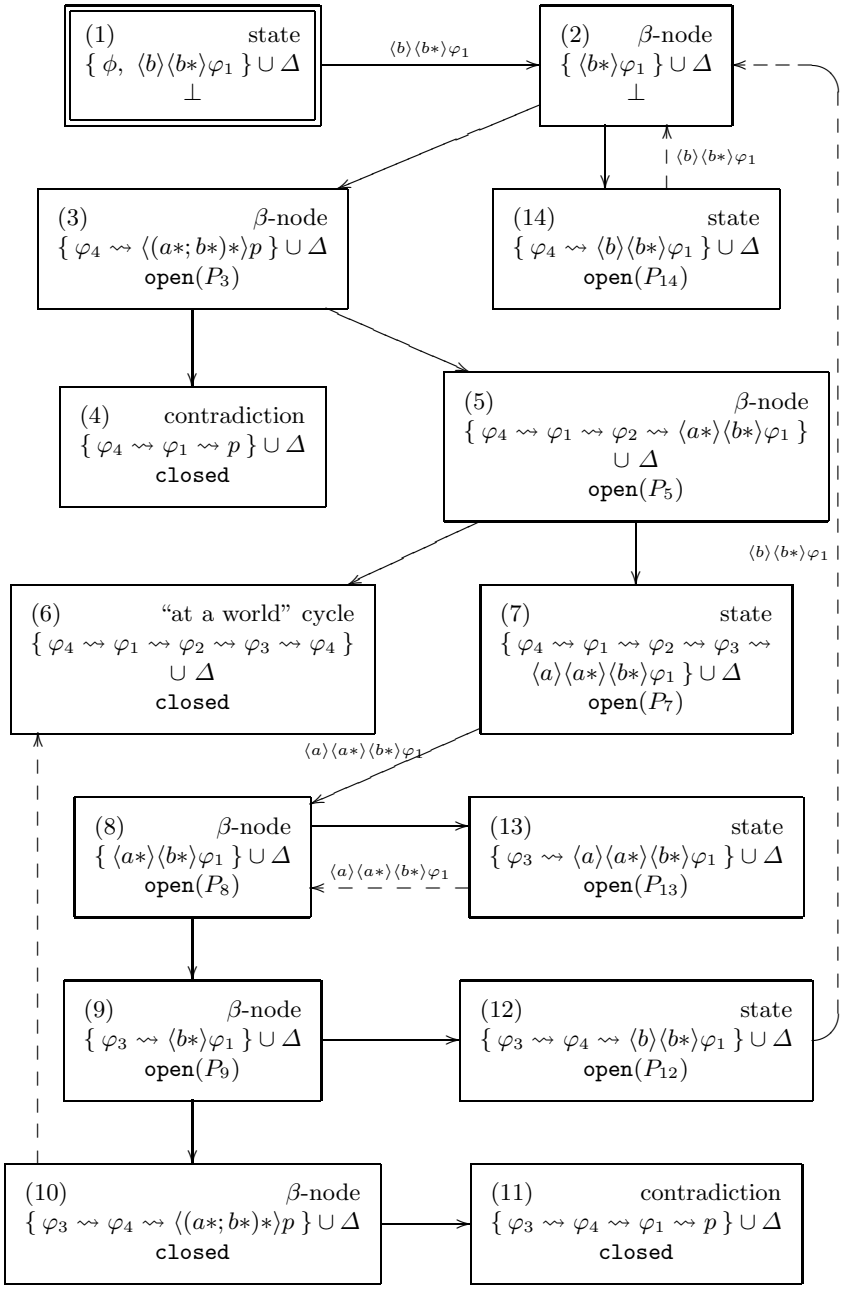


Fig. 2. An example: The graph G just before setting the status of node (2)

itself is closed too via `det-sts-`$\beta(10)$. Next, we create (12), which is a state. Its requested child is already in G as (2), so we insert a backward-edge from (12) to (2). The status of (2) is $\bot$, in particular it is not closed, and (2) is a forward-ancestor of (12). So we set the status of (12) to `open`(P_{12}) via `det-sts-state`(12) and `det-prs-child`(12, 2, $\langle b* \rangle \varphi_1$). Remember that P_{12} maps all eventualities in Γ_{12} to $\{(2, \langle b* \rangle \varphi_1)\}$. The status is then propagated to (9).

Node (13) is conceptually similar to (12). When determining the status of (8) via `det-sts-`$\beta(8)$, node (8) "inherits" two potential rescuers for each eventuality: $(2, \langle b* \rangle \varphi_1)$ from (9) and $(8, \varphi_3)$ from (13). But $(8, \varphi_3)$ is removed by `filter` since a node cannot be part of a potential rescuer of itself. The status is then propagated to (7), (5), and (3). Node (14) is conceptually similar to (12). As stated before, this is the moment at which G is shown in Fig. 2. When determining the status of (2) via `det-sts-`$\beta(2)$, node (2) "inherits" only the potential rescuer $(2, \langle b* \rangle \varphi_1)$ from its children. Since $(2, \langle b* \rangle \varphi_1)$ is removed by `filter`, the eventuality $\langle b* \rangle \varphi_1$ has no potential rescuers. Hence we know that $\langle b* \rangle \varphi_1$ cannot be fulfilled in G now or in the future, so (2) is closed via `filter`.

The subsequent invocation of `update`(2) closes all non-closed nodes in the subgraph rooted at (2). Finally, the root (1) is closed via `det-sts-state`(1).

5 Implementation Issues and Experiments

For clarity, the description of our algorithm omits some immediate optimisations. For example, once a state has a closed forward- or backward-child, creating and exploring its remaining children is moot: see $\langle a \rangle \varphi \wedge \langle b \rangle \psi$ in the introduction. Our implementation in OCaml (`http://rsise.anu.edu.au/~rpg/PDLGraphProver/`) includes this optimisation, but does not include most optimisations which have proved crucial in taming description logics [10].

As explained in the introduction, all existing implementations for PDL by other authors are either educational tools (LoTREC), or do not handle the full language (DLP, pdl-tableaux). Therefore, we compared our graph-based method with our tree-based method [11] on randomly generated formulae. Their implementations share many basic data structures and the same (limited) optimisations. Thus they should differ mostly in their tree versus graph aspects.

Each randomly generated formula of size n contained at most $n/20$ propositional variables and $n/20$ atomic programs. The formulae were created by randomly choosing a connective with equal probability and recursively creating the subformulae or subprograms. If a connective had two subformulae/subprograms, their sizes were chosen randomly so that the final formula had the desired size.

We randomly generated ten million formulae for each size 40, 50, ..., 90 and ran both provers on them. For each formula, we set a timeout of 10 seconds. If a solver timed out, we took its running time for this formula to be the timeout, that is 10 seconds. The results are given in Table 3. The lack of timeouts shows that the graph method is clearly more stable. Ignoring stability, there is not much difference in the running times for satisfiable formulae. But the graph method is clearly superior for unsatisfiable formulae.

Table 3. Average running time per 10,000 formulae (and number of timeouts if greater than 0) for ten million randomly generated formulae of each size, shown separately for satisfiable and unsatisfiable formulae

formulae size	40	50	60	70	80	90
satisfiable (%)	91.3%	91.1%	93.6%	93.5%	94.9%	94.9%
Graph (sat)	0.6s	0.8s	1.0s	1.2s	1.4s	1.6s
Tree (sat)	0.7s (2)	1.2s (23)	1.4s (22)	2.1s (62)	2.2s (46)	3.7s (151)
Graph (unsat)	0.6s	0.8s	1.0s	1.3s	1.6s	1.9s
Tree (unsat)	1.0s	5.7s (27)	7.9s (28)	22.8s (94)	29.6s (102)	52.8s (190)

On individual handcrafted examples, we sometimes observed that the tree-based method was faster, even when it generated more nodes. We believe this is because tracking eventualities in graphs requires more bookkeeping and updating than in trees, and sometimes this bookkeeping is in vain when branches do not share nodes. Another reason might be that the tree-based method requires less space since it can discard previous branches.

References

1. Fisher, M., Ladner, R.: Propositional dynamic logic of regular programs. Journal of Computer and System Sciences 18(2), 194–211 (1979)
2. Pratt, V.R.: A near-optimal method for reasoning about action. Journal of Computer and System Sciences 20(2), 231–254 (1980)
3. Baader, F.: Augmenting concept languages by transitive closure of roles: An alternative to terminological cycles. In: Proc. IJCAI 1991, pp. 446–451 (1991)
4. De Giacomo, G., Massacci, F.: Combining deduction and model checking into tableaux and algorithms for Converse-PDL. Inf. and Comp. 162, 117–137 (2000)
5. Vardi, M., Wolper, P.: Automata theoretic techniques for modal logics of programs. Journal of Computer and System Sciences 32(2), 183–221 (1986)
6. Pan, G., Sattler, U., Vardi, M.Y.: BDD-based decision procedures for the modal logic K. Journal of Applied Non-Classical Logics 16(1-2), 169–208 (2006)
7. Lange, M., Stirling, C.: Focus games for satisfiability and completeness of temporal logic. In: Proc. LICS 2001, pp. 357–365. IEEE Computer Society, Los Alamitos (2001)
8. Brünnler, K., Lange, M.: Cut-free sequent systems for temporal logic. Journal of Logic and Algebraic Programming 76(2), 216–225 (2008)
9. Goré, R., Nguyen, L.A.: EXPTIME tableaux for ALC using sound global caching. In: Proc. of the International Workshop on Description Logics (DL 2007) (2007)
10. Horrocks, I., Patel-Schneider, P.F.: Optimizing description logic subsumption. Journal of Logic and Computation 9(3), 267–293 (1999)
11. Abate, P., Goré, R., Widmann, F.: An on-the-fly tableau-based decision procedure for PDL-satisfiability. Electr. Notes Theor. Comput. Sci. 231, 191–209 (2009)

Volume Computation for Boolean Combination of Linear Arithmetic Constraints*

Feifei Ma[1,2], Sheng Liu[1,2], and Jian Zhang[1]

[1] State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
[2] Graduate University, Chinese Academy of Sciences
{maff,lius,zj}@ios.ac.cn

Abstract. There are many works on the satisfiability problem for various logics and constraint languages, such as SAT and Satisfiability Modulo Theories (SMT). On the other hand, the counting version of decision problems is also quite important in automated reasoning. In this paper, we study a counting version of SMT, i.e., how to compute the volume of the solution space, given a set of Boolean combinations of linear constraints. The problem generalizes the model counting problem and the volume computation problem for convex polytopes. It has potential applications to program analysis and verification, as well as approximate reasoning, yet it has received little attention. We first give a straightforward method, and then propose an improved algorithm. We also describe two ways of incorporating theory-level lemma learning technique into the algorithm. They have been implemented, and some experimental results are given. Through an example program, we show that our tool can be used to compute how often a given program path is executed.

1 Introduction

The past decade has seen much interest in the satisfiability (SAT) problem, i.e., determining whether a Boolean formula (in conjunctive normal form) is satisfiable. It is a classical decision problem in propositional logic reasoning. More recently, Satisfiability Modulo Theories (SMT), as an extension to SAT, has received more and more attention [3,9,18,16,19]. Instead of Boolean formulas, SMT checks the satisfiability of logical formulas with respect to combinations of background theories (often expressed in classical first-order logic with equality).

On the other hand, the counting version of the decision problems is also quite important in automated reasoning. For instance, the model counting problem, i.e., counting the number of models of a propositional formula, is closely related to approximate reasoning [20,2]. It has been studied by various researchers, especially in recent years. See for example, [14,2,23,22,1,8,13,21].

* This work is supported by the National Science Foundation of China (NSFC) under grant No. 60673044. Correspondence to: Jian Zhang, P.O.Box 8718, Beijing 100190, CHINA.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 453–468, 2009.

The counting version of SMT has received less attention so far, although it is also very important. It has potential applications in various areas, such as program analysis and verification, as well as approximate reasoning. Suppose we have a knowledge base or a formal description of some application, specified by an SMT formula Φ, and we are given a formula φ such that neither φ nor $\neg\varphi$ is a logical consequence of Φ. Then it is reasonable to assume that, the more models of Φ support φ, the more likely φ is true for the application.

There are various ways of analyzing programs statically. One class of analysis techniques checks the program's properties by processing individual paths in the program's flow graph. Not all paths in the graph correspond to program executions. A path is called *feasible* if there are some initial data values for the variables that can drive the program to be executed along that path. Otherwise, the path is called *infeasible*. There are quite some works on path feasibility analysis. A basic approach is to compute the path condition (which is a set of constraints in the form of SMT instances), and decide whether it is satisfiable or not. The program path is feasible if and only if the path condition is satisfiable. So the path feasibility analysis problem is reduced to a constraint solving problem [24,26].

Sometimes we can go one step further and ask *how many* data values satisfy the path condition, which means *how often* the program path can be executed. A path is a *hot path* if it is frequently executed. Identification of such paths is necessary in some applications such as embedded systems.

In this paper, we study how to compute the volume of the solution space (or, how to count the number of solutions), given a set of SMT instances (or more precisely, Boolean combination of linear arithmetic constraints).

The paper is organized as follows. We first recall some basic concepts and give some notations in the next section. In Section 3, we outline a straightforward method, and then in Section 4 we propose an improved algorithm. We have implemented the methods, and some experimental results are given in Section 5. In Section 6 we describe the application of the techniques to program analysis. Then we discuss some issues and mention some related works. Finally we conclude the paper.

2 Background

This section describes some basic concepts and notations. We also mention some existing techniques and tools that will be used later.

The main object of study in this paper can be regarded as an SMT instance where the theory is restricted to the linear arithmetic theory. More specifically, we study a set of constraints involving variables of various types (including integers, reals and Booleans). There can be logical operators (like AND, OR) and arithmetic operators (like addition, subtraction).

We use b_i $(i > 0)$ to denote Boolean variables, x_j $(j > 0)$, y, $\ldots$ to denote numeric variables.

A simple example of constraints is $x_1 + x_2 < x_3$. We call it a linear arithmetic constraint (LAC), which is a comparison between two linear arithmetic

expressions. Such a constraint can be denoted by a Boolean variable. A *literal* is a Boolean variable or its negation. A *clause* is a disjunction of literals.

In this paper, a constraint ϕ is represented as a Boolean formula $PS_\phi(b_1, \ldots, b_n)$ together with definitions in the form: $b_i \equiv expr_{i1}\ op\ expr_{i2}$. Here $expr_{i1}$ and $expr_{i2}$ are numeric expressions, *op* is a relational operator like '<', '=', etc. The Boolean formula PS_ϕ is the *propositional skeleton* of the constraint.

For a Boolean formula, a *model* is an assignment[1] of truth values to all the Boolean variables such that the formula is evaluated to TRUE. The *satisfiability* problem is concerned with the existence of models, while the *model counting* problem is to compute the number of models.

The SMT problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. But the subject of this paper is how to compute the *volume* of the solution space (or, how to count the number of solutions) for the formulas. Obviously this problem generalizes both the model counting problem in the propositional logic and the classical volume computation problem for convex polytopes.

Generally speaking, a polytope is defined as the bounded intersection of finitely many halfspaces/inequalities. Formally, it is usually described using the H-representation $\{\mathbf{x}|\mathbf{A}\mathbf{x} \leq \mathbf{b}\}$ (where $\mathbf{A}$ is a matrix of dimension $m \times d$ and $\mathbf{b}$ a vector of dimension m). If $\mathbf{x}$ is a real vector, there are already some tools available to compute the continuous volume of a polytope. For example, `vinci` [6] is such a tool, whose input is a set of linear inequalities over reals.

Sometimes we are interested in the number of *integer* points in the solution space. Tools are also available for counting such points inside a bounded polytope, e.g., `azove` [4] and `LattE` [17]. `azove` is a tool designed for counting and enumeration of 0/1 vertices. That is to say, the domain of the variables is $\{0, 1\}$. Given a polytope $\{\mathbf{x}|\mathbf{A}\mathbf{x} \leq \mathbf{b}\}$, all 0/1 points lying in it can be counted or enumerated. We extended the domain of each variable to $\{0, 1, ..., 2^l - 1\}$ so that it can count or enumerate all integer points with word length l in the polytope. `LattE` is a similar tool dedicated to the counting of lattice points inside convex polytopes and the solution of integer programs. But all the parameters in the matrix $\mathbf{A}$ and vector $\mathbf{b}$ should be integers.

3 A Straightforward Method

We know that an SMT(LAC) instance ϕ is satisfiable if there is an assignment α to the Boolean variables in PS_ϕ such that:

1. α propositionally satisfies ϕ, or formally $\alpha \models PS_\phi$;
2. The conjunction of theory predicates under the assignment α, which is denoted by $\hat{Th}(\alpha)$, is consistent with respect to the addressed theory.

We call an assignment satisfying the above conditions a *feasible* assignment. To check the satisfiability of a given formula, an SMT solver tries to find such an assignment.

[1] In this paper we represent an assignment as a set of literals.

Given a formula ϕ, we denote the set of all its feasible assignments by $Mod(\phi)$. When we talk about the volume of an assignment α, which is denoted by $volume(\alpha)$, we refer to the volume of polytope corresponding to α. The **volume** of a formula ϕ, denoted by V_ϕ, is formally defined as follows:

$$V_\phi = \sum_{\alpha \in Mod(\phi)} volume(\alpha)$$

An SMT solver typically combines SAT and theory-specific solving [16]. Most of the state-of-the-art SMT solvers are built within the DPLL(T) architecture [12], which can be adapted to compute the volume of a given formula. We ask the SMT solver to find out all feasible assignments to formula ϕ, compute the volume of each assignment and then add them up.

Algorithm 1

```
1: V_φ=0;
2: for each model α of PS_φ do
3:    if T̂h(α) is consistent then
4:       compute volume(α);
5:       sum up the volume: V_φ += volume(α);
6:    end if
7: end for
8: return V_φ;
```

4 Volume Computation in Bunches

4.1 The Basic Idea

In the straightforward approach, each time a feasible assignment is obtained, we need to compute the volume of a conjunction of linear constraints corresponding to the literals in the assignment. Since volume computation is a time-consuming task, which is shown to be $\#P$-hard by Dyer and Frieze [10], it is desirable to reduce the number of calls to volume computation routines.

Given an assignment α to the Boolean variables in formula ϕ, concerning the two conditions for feasible assignments, we distinguish four cases:

(i) $\alpha \models PS_\phi$, and $\hat{Th}(\alpha)$ is consistent in the specific theory. (Here α is a feasible assignment.)
(ii) $\alpha \models PS_\phi$, while $\hat{Th}(\alpha)$ is inconsistent in the specific theory.
(iii) α falsifies ϕ propositionally, while $\hat{Th}(\alpha)$ is consistent in the specific theory.
(iv) α falsifies ϕ propositionally, and $\hat{Th}(\alpha)$ is inconsistent in the specific theory.

The volume of formula ϕ is the sum of volumes of all assignments in case (i). When $\hat{Th}(\alpha)$ is inconsistent, as in case (ii) or case (iv), there is no solution to $\hat{Th}(\alpha)$, and consequently $volume(\alpha) = 0$. So when adding up the volume of feasible assignments, it will be safe to count in some theory-inconsistent assignments

since they would not affect the total volume. At first glance it seems that these zero-volume assignments would give rise to additional calls to volume computation routines. However, when properly selected, they can be combined with the feasible assignments to form fewer assignments, reducing the number of volume computations.

Definition 1. *A set of full assignments $\mathcal{S}$ is called a* bunch *if there exists a partial assignment α_c such that for any full assignment α, $\alpha \in \mathcal{S} \longleftrightarrow \alpha_c \subseteq \alpha$. α_c is called the* cube *of $\mathcal{S}$.*

In other words, the assignments in a bunch $\mathcal{S}$ share a partial assignment α_c, and for the Boolean variables which are not assigned by the cube α_c, these assignments cover all possibilities of value combinations. The cardinality of $\mathcal{S}$ is exactly $2^{n-|\alpha_c|}$, where n is the number of Boolean variables, and $|\alpha_c|$ stands for the size of α_c.

For the assignments in a bunch, computing their total volume can be greatly simplified, as the following proposition reveals:

Proposition 1. *For a bunch $\mathcal{S}$ with the cube α_c, $\sum_{\alpha\in\mathcal{S}} volume(\alpha)$=$volume(\alpha_c)$.*

Consider the formula

$$\phi = (((y + 3x < 1) \rightarrow (30 < y)) \vee (x \leq 60)) \wedge ((30 < y) \rightarrow \neg(x > 3) \wedge (x \leq 60))$$

We first introduce a Boolean variable for each linear inequality and obtain its propositional skeleton as follows:

$$PS_\phi = ((b_1 \rightarrow b_2) \vee b_4) \wedge (b_2 \rightarrow \neg b_3 \wedge b_4)$$

where:

$$\begin{cases} b_1 \equiv (y + 3x < 1); \\ b_2 \equiv (30 < y); \\ b_3 \equiv (x > 3); \\ b_4 \equiv (x \leq 60); \end{cases}$$

Using an SMT solver, we find out that there are seven feasible assignments. Hence in the straightforward method, `vinci` is called seven times to compute the volumes of the feasible assignments, and they are added up to get the total volume of the formula. Here we list three of these assignments:

$$\alpha_1 = \{\neg b_1, \neg b_2, b_3, \neg b_4\}$$
$$\alpha_2 = \{\neg b_1, \neg b_2, \neg b_3, b_4\}$$
$$\alpha_3 = \{\neg b_1, \neg b_2, b_3, b_4\}$$

Now let's consider another assignment: $\alpha_4 = \{\neg b_1, \neg b_2, \neg b_3, \neg b_4\}$. It is easy to check that α_4 satisfies PS_ϕ, but $\hat{Th}(\alpha_4)$ is inconsistent in linear arithmetic theory. Thus $volume(\alpha_4) = 0$. Also, these four assignments form a bunch whose cube is $\{\neg b_1, \neg b_2\}$. Noticing this, we have

$$\begin{aligned} & volume(\alpha_1) + volume(\alpha_2) + volume(\alpha_3) \\ = \; & volume(\alpha_1) + volume(\alpha_2) + volume(\alpha_3) + volume(\alpha_4) \\ = \; & volume(\{\neg b_1, \neg b_2\}) \end{aligned}$$

As a result, we only need to call `vinci` once to compute the volume of cube $\{\neg(y + 3x < 1), \neg(30 < y)\}$ so as to obtain the total volume of α_1, α_2 and α_3. In contrast, without incorporating α_4 and combining assignments, three calls to `vinci` are needed.

Obviously, theory-inconsistent assignments, whether propositionally satisfying the formula or not, do not affect the total volume. This observation, together with Proposition 1, suggest a way to reduce number of calls to volume computing procedure. That is, we first list all feasible assignments with the help of an SMT solver and then make possible combinations to form bunches, counting in some theory-inconsistent assignments whenever necessary. However, an SMT solver doesn't provide explicitly theory-inconsistent assignments. In order to incorporate such assignments, extra calls to theory solvers are inevitable.

4.2 The Algorithm

Our method is based on the above idea. But rather than selecting theory-inconsistent assignments and then making possible combinations as a postprocessing step, we implemented it within the decision procedure of the SMT(LAC) solver. A typical SMT solver does not provide deduction procedure with respect to the specific theory for assignments that falsify the propositional skeleton of the formula. Therefore, we have to ignore the assignments in case (iv). In other words, the additional assignments possibly incorporated are those in case (ii).

A key point is that when the SMT solver finds a feasible assignment, we try to obtain a smaller one which still propositionally satisfies the formula. It is formally defined as follows:

Definition 2. *Suppose α is a feasible assignment for formula ϕ. An assignment α_{mc} is called a* minimum cube *of α if*

1. $\alpha_{mc} \subseteq \alpha$ *and* $\alpha_{mc} \models PS_\phi$.
2. $\forall \alpha'(\alpha' \models PS_\phi \rightarrow \alpha' \not\subset \alpha_{mc})$.

In fact, the minimum cube α_{mc} of an assignment α is the cube of a bunch $\mathcal{S}$ such that for any bunch $\mathcal{S}'$, $\alpha \in \mathcal{S}' \rightarrow \mathcal{S} \not\subset \mathcal{S}'$. Any assignment in $\mathcal{S}$ also satisfies PS_ϕ because only part of it, say α_{mc}, has evaluated PS_ϕ to be true. As we have explained before, it is pretty safe to count in such an assignment while computing the total volume, regardless of its consistency in the specific theory.

For a feasible assignment α and its minimum cube α_{mc}, from Proposition 1 we know that $volume(\alpha_{mc})$ includes $volume(\alpha)$, and possibly the volumes of other feasible assignments. Thus it seems rewarding to compute $volume(\alpha_{mc})$ instead of $volume(\alpha)$. Note that an assignment might have several minimum cubes. Currently we use a simple method to find only one minimum cube. It checks the redundancy of each literal contained in α sequentially. If α with a literal l_i removed still evaluates PS_ϕ to be true, then l_i is immediately deleted from α, and the next literal is checked with respect to the modified α. It can be easily proved that the final result is a minimum cube of the original assignment.

The SMT solving framework is adapted to compute the volume of a formula. Each time a feasible assignment is found, its minimum cube is computed and the

volume of the minimum cube is added to the total volume. Then the negation of the minimum cube is added to the original formula so that a feasible assignment would not be counted more than once. It is a blocking clause, ruling out all the assignments in the bunch related to the minimum cube.

In a feasible assignment, some variables are decision variables, while others get assigned by Boolean constraint propagation (BCP). These implied literals need not be checked when finding the minimum cube of an assignment. (It will be proved in Proposition 2.) The detailed algorithm is presented as Algorithm 2.

Algorithm 2. Volume Computation in Bunches

```
Boolean Formula PS = PS_φ;
volume = 0;
while TRUE do
  if BCP() == CONFLICT then
    backtrack-level = AnalyzeConflict();
    if backtrack-level < 0 then
      return volume;
    end if
    backtrack to backtrack-level;
  else
    α = current assignment;
    if α ⊨ PS then
      if Ĥh(α) is inconsistent then
        backtrack to the latest decision variable;
      else
        for all literal l_i ∈ α do
          if l_i is a decision variable or its negation then
            α' = α − {l_i};
            if α' ⊨ PS then
              α = α';
            end if
          end if
        end for
        volume+=VOLcompute(α);
        Add ¬α to PS;
      end if
    else
      choose a Boolean variable and extend the current assignment;
    end if
  end if
end while
```

Proposition 2. *Given a feasible assignment α and a literal l_i in α, if l_i is not a decision variable or its negation, then it must appear in any minimum cube of α.*

Proof. We prove it by contradiction. Suppose α_{mc} is a minimum cube of α and $l_i \notin \alpha_{mc}$, then α_{mc} can be extended to a full assignment α', which is the same

as α except that l_i is replaced with $\neg l_i$. Since $\alpha' - \{\neg l_i\} = \alpha - \{l_i\}$ together with the current Boolean formula PS implies that l_i must be true, we know that α' falsifies PS, and that α_{mc} cannot be a minimum cube. □

Theorem 1. *Algorithm 2 computes the volume of formula ϕ.*

Proof. Since the volume of a formula ϕ is the total volume of all feasible assignments, we shall show that: any feasible assignment to ϕ is counted into the total volume exactly once, and no non-empty volume of an assignment that propositionally falsifies ϕ is introduced.

Firstly, assume a feasible assignment α is missing while computing the volume of ϕ. Denote the new Boolean formula when the program terminates by PS'. Clearly $PS' = PS_\phi \wedge C$, where C stands for all the blocking clauses. In fact, $C = \bigwedge \neg c$, where c ranges over all minimum cubes discovered by the algorithm. Since α is missing, it cannot be obtained by extending any minimum cube c. Thus we have $\alpha \models \neg c$ and further $\alpha \models C$. As a feasible solution to ϕ, α propositionally satisfies ϕ, i.e., $\alpha \models PS_\phi$. Therefore, we have $\alpha \models PS'$, contradicting the termination condition that there is no assignment that satisfies PS'.

Secondly, because of the blocking clauses, a feasible assignment would not be counted more than once.

Finally, it is quite clear that a theory-consistent assignment which falsifies ϕ propositionally can't be extended from any minimum cube and is impossible to be counted in while computing the total volume. □

We would like to further clarify that

Proposition 3. *The number of calls to volume computing procedure in Algorithm 2 is not more than the straightforward method.*

This is quite clear since each minimum cube represents at least one feasible assignment, so the proof is omitted.

4.3 Incorporating Theory-Level Conflict-Driven Learning

Conflict-driven Learning with respect to the specific theory is an important technique for efficient SMT solving. It has been adopted by most of the current state-of-the-art SMT solvers [3,9,18]. When presenting Algorithm 2, we omit this technique for clarity. In this subsection, we shall give a brief introduction to this technique and explain that it can be easily integrated into the framework of our algorithm.

It is known that efficient lemma learning based on conflict analysis contributes greatly to modern SAT solvers. For an SMT solver which combines a DPLL-style SAT solver and a decision procedure for a conjunctive fragment of a theory T, lemma learning could go beyond the propositional level. When the decision procedure discovers that the current assignment α is inconsistent in T, it tries to explain the inconsistency, finding out the literals in α that lead to the conflict. The disjunction of negations of these literals is then obtained and passed to the SAT solver as a lemma. The lemma is learned by analyzing the inconsistency,

and it prevents the same inconsistency from happening again. As a result, the search space is pruned.

Here we study two methods for incorporating lemma learning technique into Algorithm 2.

Method 1. Each time the current assignment α is found theory-inconsistent, a lemma is derived and added to the current Boolean formula PS. The rest of Algorithm 2 is unchanged. The modified algorithm is still correct. The proof is similar to that of Theorem 1: Suppose a feasible assignment of ϕ, denoted by α, is missing when the program terminates. At the termination the Boolean formula PS' consists of three parts: PS_ϕ, blocking clauses C, and lemmas learned from the theory inconsistencies LM. Since α is theory-consistent, for any lemma $cl \in LM$ we have α falsifies $\neg cl$ or equivalently $\alpha \models cl$. Consequently $\alpha \models LM$. From the proof of Theorem 1 we already know that $\alpha \models PS_\phi \wedge C$, thus we have $\alpha \models PS'$. So α cannot be missing when the program terminates. The rest of the proof is the same as that of Theorem 1. Furthermore, it is obvious that Proposition 2 and Proposition 3 still hold.

Method 2. When some lemmas are learnt from theory inconsistency, instead of adding them to the Boolean formula PS, we put them apart from PS. Hence these lemmas participate in the process of finding a feasible assignment α, while they do not affect finding a minimum cube of α. Proposition 2 no longer holds in this situation. For a counterexample, consider the following formula:

$$\phi = (x > 3 \vee x < 6) \wedge (x < 6 \vee x < 2) \wedge (x < 2)$$

Suppose the search procedure has made two decision assignments: $x < 2, x > 3$. Since they are theory-inconsistent, a lemma $\neg(x < 2) \vee \neg(x > 3)$ is derived and the program backtracks to the only decision assignment $x < 2$. Two literals, $\neg(x > 3)$ and $x < 6$ are then implied by BCP. A feasible assignment is now obtained, with a minimum cube $\{x < 2, x < 6\}$. Now the literal $\neg(x > 3)$ is neither a decision variable nor its negation, but it is not contained in the minimum cube, contradicting Proposition 2. As a result, in this method, all literals in the feasible assignment have to be checked for redundancy.

5 Implementation and Experimental Results

The algorithms were implemented on the basis of the SAT solver `MiniSat` 2.0 [11], which serves as the search engine for the Boolean structure of the SMT(LAC) instance. The linear programming tool `lp_solve` [5] is integrated for deciding the consistency of a conjunction of linear constraints. When a minimum cube is obtained, the program calls `vinci` to compute the volume of the corresponding polytope, or `azove` / `LattE` to count the number of integer points in the polytope.

To study the effectiveness of the aforementioned techniques, we randomly generated a number of SMT(LAC) instances whose propositional skeletons are in CNF. Since `vinci` is very slow for polytopes in more than 8 dimensions, each instance contains no more than 8 numeric variables.

Table 1. Comparison of Algorithms

		Algorithm 2				Algorithm 1	
		Method 1		Method 2			
Instance	P cls V	Time (s)	#calls	Time (s)	#calls	Time (s)	#calls
Ran1	8 50 4	0.01	19	0.02	19	0.03	41
Ran2	10 40 5	0.06	53	0.06	50	0.14	182
Ran3	15 40 5	5.35	57	2.36	47	11.12	188
Ran4	20 40 5	106.63	332	116.72	259	431.15	17158
Ran5	10 20 6	1.06	47	1.04	41	7.81	212
Ran6	10 50 6	1.07	74	2.08	74	5.32	247
Ran7	15 50 6	2.12	52	2.15	57	10.97	257
Ran8	7 40 7	1.01	16	1.01	16	2.77	39
Ran9	12 40 7	51.26	245	50.29	250	502.75	1224
Ran10	15 50 7	314.26	833	303.14	856	3872.70	5224
Ran11	20 50 7	214.13	158	143.11	140	1889.36	807
Ran12	10 20 8	13.04	39	12.04	37	150.92	235
Ran13	10 40 8	51.09	91	51.10	91	398.02	379
Ran14	16 80 8	1104.05	648	1074.48	669	4 hours	4273

`P`: number of linear constraints. `cls`: number of clauses.
`V`: number of numerical variables. `#calls`: number of calls to `vinci`.

We have implemented both methods for the theory-level conflict-driven learning mechanism described in subsection 4.3. We compare them with the straightforward approach on the random instances. The experimental results are given in Table 1. The programs are run on an Intel 1.86GHZ Core Duo 2 PC with Fedora 7 OS. For a problem instance with 7 or 8 numerical variables, if we use `azove`, the running time will be too long. So the table only contains data for the cases when `vinci` is called.

From Table 1 we can see that both the running time and the number of calls are reduced with Algorithm 2, in some cases by an order of magnitude. However, there is no clear winner between the two methods for lemma learning.

We have also tried some benchmark problems in SATLIB[2]. It turns out that some SAT instances (e.g., `Bejing`) have so many solutions that the whole search space cannot be exhausted in reasonable time. On the other hand, some other instances have just a few solutions (even one solution per instance), or the solutions are scattered in the multi-dimensional space. Table 1 does not include data for the above instances.

6 Application to Program Analysis

In [25], we proposed a measure $\delta(P)$ for a program path P. Intuitively, it denotes the volume of the subspace in the input space of the program that corresponds to the execution of path P; or, in other words, the number of input data vectors

[2] `http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`

which drive the program to be executed along this path. Obviously, a special case is that P is infeasible, and $\delta(P) = 0$.

For a simple example, see the following program:

```
int i, j;
if(i+j > 10)
    j = 2;
else
    j = 1;
```

The program has two paths, denoted by P_1 and P_2, which correspond to the `if-then` branch and the `else` branch, respectively. The path conditions are $(i + j > 10)$ and $(i + j \leq 10)$.

Suppose the input variables i, j take values from the interval [1..10]. We can compute that $\delta(P_1) = 55$, $\delta(P_2) = 45$. Now suppose the variables can take values from a larger interval, e.g., [1..100]. Then we have $\delta(P_1) = 9955$, $\delta(P_2) = 45$. We can see that the first path is executed much more frequently than the second path, under reasonable assumptions of the input data space.

6.1 Execution Probability

For many paths, the path condition involves just a subset of the input variables. In such a case, we can preprocess the constraints, and remove the irrelevant variables first. This will reduce the dimension of the solution space and also the volume computation time. The following is a contrived example of programs:

```
int i, j;
int a[10], b[10];
for(i = 0; i < 10; i++)
   for(j = 0; j < 10; j++)
      if(a[i] < b[j])
         return a[i];
```

There are many paths in the program's flow graph. The following is the path condition for one of them:

$$(a[0] \geq b[0]) \wedge (a[0] \geq b[1]) \wedge (a[0] < b[2])$$

The condition involves just 4 (among 20) array elements explicitly. So when computing the δ value of the path, we omit the other 16 array elements, assuming the input space is of 4 dimensions rather than 22 (20 array elements and `i`, `j`). The computation is greatly simplified.

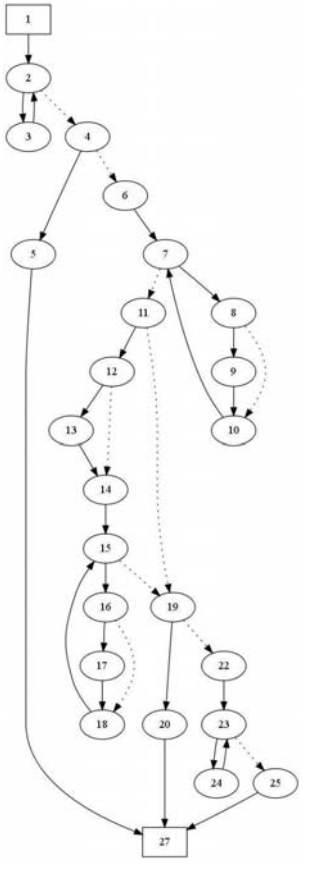
However, a problem arises when two paths involve different variables, since the δ values are no longer comparable. We have to introduce another measurement, namely *execution probability*. The execution probability of a path P, denoted by $\mathcal{XP}(P)$, is defined as $\delta(P)$ divided by the volume of the involved data space.

Suppose there are m variables in the path condition of P, and the range length or domain size of the ith variable is l_i. We have

$$\mathcal{XP}(P) = \frac{\delta(P)}{\prod_{1 \leq i \leq m} l_i}.$$

6.2 A Practical Example

Now we describe our experiments with a real program, i.e., a function called *getop()* which is taken from [15]. It has been used as an example in several research papers on software testing. The function fetches the next operator or operand for a calculator program. Its input variables are characters (except for one integer variable). It has `for`-loops, `while`-loops, and `if` statements. The conditional expressions in these statements contain logical operators. This makes the path conditions a bit complex. The control flow graph is demonstrated in Fig. 1. Each node in the graph represents a conditional expression or a block




```
1:  c=getchar();
2:  !(((c==32 || c==9) || c==10))
3:  c=getchar();
4:  !((c!=46 && (c<48 || c>57)))
5:  RETURN: c;
6:  s[0]=c; c=getchar(); i=1;
7:  !((c>=48 && c<=57))
8:  !(i<lim)
9:  s[i]=c;
10: c=getchar(); _temp_var0=i; i=i+1;
11: !(c==46)
12: !(i<lim)
13: s[i]=c;
14: c=getchar(); _temp_var1=i; i=i+1;
15: !((c>=48 && c<=57))
16: !(i<lim)
17: s[i]=c;
18: c=getchar(); _temp_var2=i; i=i+1;
19: !(i<lim)
20: s[i]=0; RETURN: 1000;
22: NOP
23: !((c!=10 && c!=-1))
24: c=getchar();
25: _temp_var3=lim-1; s[_temp_var3]=0; RETURN: 9999;
27: END
```

Fig. 1. getop()

of statements, as shown on the right hand side of Fig. 1. At a branching node, the dotted arrow represents the `true` branch, and the solid arrow represents the `false` branch. The flow graph and the expressions/blocks are produced by our test generation tools. In this process, some auxiliary variables like `_temp_var2` are introduced.

We obtained several paths from the program, and computed the path condition for each path. The execution probability of each path was computed by our tool quickly. The running times are within 0.03 second.

In the experiment, we assume that each character variable is an integer, taking values from the range $[0, 255]$. We can see that some paths have a larger probability of being executed, as compared with other paths. For example, see the two paths: $Path1$ is $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 27$ and $Path2$ is $1 \rightarrow 2 \rightarrow 4 \rightarrow 7 \rightarrow 11 \rightarrow 19 \rightarrow 20 \rightarrow 27$. With our tool, we found that $\mathcal{XP}(Path1) \approx 0.945$ and $\mathcal{XP}(Path2) \approx 0.021$.

7 Related Works and Discussion

So far as we know, there is little work in the literature concerning volume computation for SMT. In contrast, counting the number of models for SAT has become a hot topic in the AI community recently. However, some advanced model counting techniques (e.g., component analysis [14] and caching [22]) cannot be used directly here. The main reason is that the Boolean variables in the SMT formulas are not independent of each other.

In the software engineering community, Buse and Weimer [7] proposed a method for *estimating* path execution frequency. They give a statistical model, which is based on *syntactic* features of the program's source code. In contrast, our approach uses *semantic* information in the program paths, and it can calculate the *exact* probability of executing a path.

In our approach, the domain of linear arithmetic is separated from the logic part. An alternative is to translate the fixed-length integer variables into Boolean variables, and then using a pure Boolean logic approach. This should have some advantages when the program to be analyzed has both arithmetic operations and bit operations. However, its scalability deserves further investigation.

The main topic of our paper is the volume (or solution space size) of SMT formulas. But as illustrated in the previous section, sometimes it is desirable to talk about the probability of satisfying the formulas. So we can introduce the notion of *satisfying probability* for a formula. Assume that each variable takes values uniformly from its range. Given a formula ϕ, dividing V_ϕ by the size of the whole space, we shall get the probability that ϕ could be satisfied, which is denoted by $\mathcal{P}(\phi)$.

We have been discussing how to compute V_ϕ assuming that the linear constraints in ϕ are restricted to one data type, either integer or real. Now we extend our problem a bit further. Suppose there are two kinds of linear constraints in the formula ϕ: some are defined over p integer variables with the domain size l_I,

and the others are defined over q real variables with the range length l_R. The satisfying probability of ϕ is defined as follows:

$$\mathcal{P}(\phi) = \sum_{\alpha \in Mod(\phi)} \frac{volume(\alpha_I)}{l_I^p} \times \frac{volume(\alpha_R)}{l_R^q}$$

where α_I is the partial assignment to integer constraints, and α_R to real constraints. It can be easily checked that this probability is well defined. Algorithm 2 can also be easily modified to compute $\mathcal{P}(\phi)$.

If a linear constraint in formula ϕ involves both integer and real variables, the explicit expression of $\mathcal{P}(\phi)$ is not clear. Our algorithm is not applicable to such cases, either. But we find that the volume of a polytope is quite close to the number of integer points it contains except for some special cases. Thus in practice, we can use type casting to unify the data types and get an approximate result. Despite this, we consider this issue as an interesting problem, worthy of further investigation.

8 Concluding Remarks

This paper studies the automatic computation of the volume (or solution space size) of SMT instances. We proposed a non-trivial algorithm, and augmented it with theory-level lemma learning technology. The algorithms have been implemented, and some experimental results are presented.

In this paper, we focus on just one application of volume computation techniques. Our experiences show that they can be used to compare program paths with respect to their execution frequency, for real programs. We believe that the techniques can be applied to other problems as well.

There are several ways in which the current approach might be improved. Firstly, Algorithm 2 is an on-the-fly method. The bunches are derived as the program runs, and the number of bunches is not predictable. It would be better if there were a search strategy for finding a small number of bunches without too much extra cost. Secondly, as we mentioned in Section 4, we are not able to incorporate the assignments in case (iv) at present. We plan to devise a post-processing technique to incorporate some of them when necessary. As a result, some of the bunches might be combined. Finally, we will investigate the best way of incorporating lemma learning into our framework.

Acknowledgement

The authors would like to thank the anonymous referees for their helpful comments and suggestions. Runming Lu helped us implement the prototype tool, and Hui Ruan helped in the preparation of the example in Sec. 6.2.

References

1. Andrei, S., Chin, W., Cheng, A.M.K., Lupu, M.: Automatic debugging of real-time systems based on incremental satisfiability counting. IEEE Trans. Computers 55(7), 830–842 (2006)
2. Bacchus, F., Dalmao, S., Pitassi, T.: Algorithms and complexity results for #SAT and Bayesian inference. In: Proceedings of the 44th Symposium on Foundations of Computer Science (FOCS 2003), pp. 340–351 (2003)
3. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007), http://www.cs.nyu.edu/acsys/cvc3
4. Behle, M., Eisenbrand, F.: 0/1 vertex and facet enumeration with BDDs. In: Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX 2007) (2007), http://www.mpi-inf.mpg.de/~behle/azove.html
5. Berkelaar, M., Eikland, K., Notebaert, P.: The lp_solve Page, http://lpsolve.sourceforge.net/
6. Büeler, B., Enge, A., Fukuda, K.: Exact volume computation for polytopes: a practical study. Polytopes–combinatorics and computation (1998), http://www.lix.polytechnique.fr/Labo/Andreas.Enge/Vinci.html
7. Buse, R.P.L., Weimer, W.R.: The road not taken: Estimating path execution frequency statically. In: Proceedings of the 31st International Conference on Software Engineering (ICSE 2009) (2009)
8. Davies, J., Bacchus, F.: Using more reasoning to improve #SAT solving. In: Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI 2007), pp. 185–190 (2007)
9. Dutertre, B., Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006), http://yices.csl.sri.com/
10. Dyer, M., Frieze, A.: On the complexity of computing the volume of a polyhedron. SIAM J. Comput. 17(5), 967–974 (1988)
11. Eén, N., Sorensson, N.: The MiniSat Page, http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/
12. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast decision procedures. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 175–188. Springer, Heidelberg (2004)
13. Gomes, C.P., Hoffmann, J., Sabharwal, A., Selman, B.: From sampling to model counting. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 2293–2299 (2007)
14. Bayardo Jr., R.J., Pehoushek, J.D.: Counting models using connected components. In: Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on on Innovative Applications of Artificial Intelligence (AAAI/IAAI 2000), pp. 157–162 (2000)
15. Kernighan, B.W., Ritchie, D.M.: The C Programming Language. Prentice-Hall, Englewood Cliffs (1978)
16. Kroening, D., Strichman, O.: Decision Procedures — An Algorithmic Point of View. Springer, Heidelberg (2008)
17. Loera, J.A.D., Hemmeckeb, R., Tauzera, J., Yoshidab, R.: Effective lattice point counting in rational convex polytopes. Journal of Symbolic Computation 38(4), 1273–1302 (2004), http://www.math.ucdavis.edu/~latte/

18. Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008), http://research.microsoft.com/projects/z3/index.html
19. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM 53(6), 937–977 (2006)
20. Roth, D.: On the hardness of approximate reasoning. Artif. Intell. 82(1-2), 273–302 (1996)
21. Samer, M., Szeider, S.: Algorithms for propositional model counting. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS, vol. 4790, pp. 484–498. Springer, Heidelberg (2007)
22. Sang, T., Beame, P., Kautz, H.A.: Heuristics for fast exact model counting. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 226–240. Springer, Heidelberg (2005)
23. Wei, W., Selman, B.: A new approach to model counting. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 324–339. Springer, Heidelberg (2005)
24. Zhang, J.: Specification analysis and test data generation by solving boolean combinations of numeric constraints. In: Proceedings of the 1st Asia-Pacific Conference on Quality Software (APAQS 2000), pp. 267–274 (2000)
25. Zhang, J.: Quantitative analysis of symbolic execution. Presented at the 28th International Computer Software and Applications Conference (COMPSAC 2004) (2004)
26. Zhang, J., Wang, X.: A constraint solver and its application to path feasibility analysis. International Journal of Software Engineering and Knowledge Engineering 11(2), 139–156 (2001)

A Generalization of Semenov's Theorem to Automata over Real Numbers*

Bernard Boigelot[1], Julien Brusten[1,**], and Jérôme Leroux[2]

[1] Institut Montefiore, B28
Université de Liège
B-4000 Liège, Belgium
{boigelot,brusten}@montefiore.ulg.ac.be
[2] Laboratoire Bordelais de Recherche en Informatique (LaBRI)
351, cours de la Libération
F-33405 Talence Cedex, France
leroux@labri.fr

Abstract. This work studies the properties of finite automata recognizing vectors with real components, encoded positionally in a given integer numeration base. Such automata are used, in particular, as symbolic data structures for representing sets definable in the first-order theory $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$, i.e., the mixed additive arithmetic of integer and real variables. They also lead to a simple decision procedure for this arithmetic.

In previous work, it has been established that the sets definable in $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$ can be handled by a restricted form of infinite-word automata, weak deterministic ones, regardless of the chosen numeration base. In this paper, we address the reciprocal property, proving that the sets of vectors that are simultaneously recognizable in all bases, by either weak deterministic or Muller automata, are those definable in $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$. This result can be seen as a generalization to the mixed integer and real domain of Semenov's theorem, which characterizes the sets of integer vectors recognizable by finite automata in multiple bases. It also extends to multidimensional vectors a similar property recently established for sets of numbers.

As an additional contribution, the techniques used for obtaining our main result lead to valuable insight into the internal structure of automata recognizing sets of vectors definable in $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$. This structure might be exploited in order to improve the efficiency of representation systems and decision procedures for this arithmetic.

* This work is supported by the *Interuniversity Attraction Poles* program *MoVES* of the Belgian Federal Science Policy Office, by the grant 2.4530.02 of the Belgian Fund for Scientific Research (F.R.S.-FNRS), and by the French project ANR-06-SETI-001 AVERISS.

** Research fellow ("Aspirant") of the Belgian Fund for Scientific Research (F.R.S.-FNRS).

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 469–484, 2009.

1 Introduction

In the program analysis and verification field, one often faces the problem of finding a suitable formalism for expressing the constraints to be satisfied by the system configurations. Ideally, this formalism has to be decidable, while still remaining expressive enough for handling the class of constraints needed by the application. An example of such formalism is Presburger arithmetic, i.e., the first-order additive theory of integers $\langle \mathbb{Z}, +, \leq \rangle$, which is widely used for reasoning about programs manipulating integer variables. Presburger arithmetic is indeed decidable [1,2], yet expressive enough for describing arbitrary linear constraints as well as discrete periodicities [3].

A simple approach to deciding Presburger arithmetic consists in using finite automata. It is indeed known that, using the positional notation for encoding numbers and vectors into words, all Presburger-definable sets are mapped onto regular languages and can thus be recognized by automata [2,3]. A Presburger formula can be decided by recursively constructing an automaton recognizing its solutions, and then checking whether this automaton accepts a nonempty language. In some program verification applications, such automata, called *Number Decision Diagrams (NDDs)* are actually used as data structures for representing and manipulating symbolically the sets of program configurations that need to be handled [4].

Although every subset of $\mathbb{Z}^n$ that is Presburger-definable can be recognized by a finite automaton, the reciprocal property does not hold. For instance, denoting by $r \in \mathbb{N}_{>1}$ the base chosen for encoding numbers, the set $\{r^k \mid k \in \mathbb{N}\}$, which is not Presburger-definable, clearly corresponds to a regular language and is thus recognizable. The well-known Cobham's theorem states that, if a set $S \subseteq \mathbb{Z}$ is simultaneously recognizable by finite automata in two bases $r, s \in \mathbb{N}_{>1}$ that are multiplicatively independent, i.e., such that $r^p \neq s^q$ for all $p, q \in \mathbb{N}_{>0}$, then S is Presburger definable [5]. This result has then been extended to subsets of $\mathbb{Z}^n$, with $n > 0$, i.e., sets of integer vectors, by Semenov [6]. As a corollary of Semenov's theorem, the subsets of $\mathbb{Z}^n$ that are recognizable by finite automata in every base $r \in \mathbb{N}_{>1}$ are exactly those that are Presburger-definable.

Quite recently, automata recognizing sets of numbers and vectors have been generalized to the mixed integer and real domain [7]. In this setting, the base-r encoding of numbers and vectors take the form of infinite words over the alphabet $\{0, 1, \ldots, r-1, \star\}$, where "$\star$" is a separator symbol used for distinguishing their integer and fractional parts. A *Real Vector Automaton (RVA)* recognizing a set $S \subseteq \mathbb{R}^n$ is then an infinite-word automaton that accepts the encodings of the elements of S.

It is worth stressing out that RVA are not only theoretical objects; they are used as actual data structures in verification tools such as LASH for representing symbolically the sets of configurations of programs relying on both integer and real variables during their state-space exploration [8]. The decision tool LIRA also uses RVA for representing the set of solutions of mixed real and integer

arithmetic formulas [9]. For such applications, it is not sufficient to establish that all sets of interest are representable by RVA and that all the needed operations are computable on them, but also to obtain a symbolic representation system that is concise enough for handling complex sets using a reasonable amount of memory, and for which the manipulation algorithms are efficient. In particular, this precludes the use of unrestricted infinite-word automata for describing RVA, due to the difficulty of carrying out some operations such as set complementation [13]. It is therefore essential to define restricted forms of RVA that can be efficiently handled, and to precisely characterize their expressiveness in order to match the requirements of the intended applications. Another goal is to investigate whether the transition relation of these restricted RVA has structural properties that can be exploited in order to represent them more efficiently.

In previous work, a result analogous to Cobham's theorem has been obtained for RVA: The sets $S \subseteq \mathbb{R}$ that are recognizable by RVA in two bases that do not share the same set of prime factors[1] are exactly those that are definable in the first-order theory $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$, i.e., the extension of Presburger arithmetic to mixed integer and real variables [10]. This has an important consequence. One indeed knows that the full expressive power of ω-regular languages is not needed for representing the sets definable in $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$, since those sets can be recognized by *weak deterministic automata* [11], a restricted class of infinite-word automata that are much more easily manipulated algorithmically, and admit a canonical form. It follows that the sets of reals that are recognizable by RVA in every base $r \in \mathbb{N}_{>1}$ are exactly those that can be recognized by weak deterministic RVA.

This paper is aimed at extending this result to sets of vectors with a fixed dimension, i.e., to subsets of $\mathbb{R}^n$ with $n > 0$. This can be seen as a generalization of Semenov's theorem to real vectors. Precisely, we prove that the sets $S \subseteq \mathbb{R}^n$ that are simultaneously recognizable by RVA in two bases that do not share the same set of prime factors are those that are definable in $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$. From this result, it follows that the sets of vectors that are recognizable by RVA in every base $r \in \mathbb{N}_{>1}$ are exactly those that can be recognized by weak deterministic RVA. The same proof also establishes that the sets that are recognizable by weak deterministic RVA in two multiplicatively independent bases are definable in $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$ as well. Those results are significant for practical applications, since they imply that weak deterministic automata can be used for implementing RVA in all cases where the sets of vectors that are symbolically represented are expressed as combinations of linear constraints and discrete periodicities.

As an additional contribution, the techniques used for obtaining this result give out valuable insight into the internal structure of RVA recognizing sets of vectors definable in $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$. It might be possible to exploit this structure in order to improve the efficiency of symbolic representation systems and decision procedures for that arithmetic.

[1] As opposed to the integer case, it has been shown that the result does not hold for multiplicatively independent bases [10].

2 Preliminaries

2.1 Positional Encoding of Vectors

Let $r \in \mathbb{N}_{>1}$ be a *base*. The positional notation in base r *encodes* numbers $x \in \mathbb{R}$ as infinite words of the form $w_I \star w_F$ over the finite alphabet $\Sigma_r \cup \{\star\}$, where $\Sigma_r = \{0, 1, \ldots, r-1\}$, and "$\star$" is a separator symbol. The finite prefix $w_I \in \Sigma_r^+$ and the infinite suffix $w_F \in \Sigma_r^\omega$ respectively encode an *integer* and a *fractional part* of x. In other words, w_I encodes a number $x_I \in \mathbb{Z}$, and w_F a number $x_F \in [0, 1]$, such that $x_I + x_F = x$. Note that integer numbers admit two decompositions into integer and fractional parts, e.g., $x = 2$ leads to both $x_I = 2$ and $x_F = 0$, and $x_I = 1$ and $x_F = 1$.

An encoding w_I of an integer part $x_I \in \mathbb{N}$ is a word $a_{p-1}a_{p-2}\ldots a_0 \in \Sigma_r^+$, with $p > 0$, such that $x_I = \sum_{i=0}^{p-1} a_i r^i$. For signed numbers $x_I \in \mathbb{Z}$, the *r's-complement* representation is used, implying that the *sign* digit a_{p-1} is then equal to 0 for positive (or zero) numbers, and to $r-1$ for negative ones. For a negative number, the value of x_I becomes equal to $-r^p + \sum_{i=0}^{p-1} a_i r^i$. The length p of encodings is not fixed, but chosen large enough for satisfying the constraint $-r^{p-1} \leq x_I < r^{p-1}$. Finally, an encoding w_F of a fractional part $x_F \in [0, 1]$ is a word $b_0 b_1 \ldots \in \Sigma_r^\omega$ such that $x_F = \sum_{i \geq 0} b_i r^{-i}$.

This encoding scheme can be extended to vectors $\boldsymbol{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$, with $n > 0$. The idea is to encode each component x_i separately into a word w_i, but in such a way that these words share the same integer-part length. This can always be achieved, for the sign digit of an encoding can be repeated at will without altering the encoded value. One thus obtains a vector $(w_1, w_2, \ldots, w_n)$ of encodings in which the separator symbol "$\star$" occurs at the same position in each component. By reading those components synchronously one symbol at a time, one eventually obtains an encoding of $\boldsymbol{x}$ as a single word $w_I \star w_F$ over the n-dimensional alphabet Σ_r^n, augmented with a unique separator symbol "$\star$". For each word $w \in \{0, r-1\}^n (\Sigma_r^n)^* \star (\Sigma_r^n)^\omega$, the vector $\boldsymbol{x} \in \mathbb{R}^n$ encoded by w in base r is denoted by $[w]_r$.

2.2 Real Vector Automata

Consider a base $r \in \mathbb{N}_{>1}$ and a set $S \subseteq \mathbb{R}^n$, with $n > 0$. Let $L(S) \subseteq (\Sigma_r^n)^+ \star (\Sigma_r^n)^\omega$ denote the language formed by the encodings of the elements of S. If $L(S)$ is ω-regular, then any infinite-word automaton that accepts $L(S)$ is a *Real Vector Automaton (RVA)* recognizing S [7]. In this paper, in order to simplify some developments thanks to their deterministic transition relation, we assume w.l.o.g. that RVA take the form of *Muller automata* [12]. A set $S \subseteq \mathbb{R}^n$ that can be recognized by a RVA in base r is said to be *r-recognizable*.

For practical applications as symbolic representations of sets, infinite-word automata are somehow problematic, since some of their manipulation algorithms are known to be significantly costlier than their finite-word counterparts [13]. In the case of RVA, its has been shown that the full expressive power of infinite-word automata is not needed for recognizing the sets definable in the first-order theory

$\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$. Indeed, for any base $r \in \mathbb{N}_{>1}$, such sets can be recognized by *weak deterministic RVA* [11]. A weak RVA representing a set S is a Büchi automaton accepting $L(S)$, such that each of its strongly connected components is either globally accepting or globally non-accepting. Compared to general infinite-word automata, weak deterministic ones are much more easily manipulated algorithmically. In addition, they admit a canonical form that simplifies comparison operations between symbolically represented sets [14]. We will say that a set $S \subseteq \mathbb{R}^n$ that can be recognized by a weak deterministic RVA in base r is *weakly r-recognizable*.

2.3 Properties of Recognizable Sets

In the next sections, we study the properties of sets that are recognizable, or weakly recognizable, in one or several bases. Such sets are characterized by the following results.

Theorem 1 ([15]). *Let $n \in \mathbb{N}_{>0}$ and $r \in \mathbb{N}_{>1}$. A set $S \subseteq \mathbb{R}^n$ is r-recognizable iff it is definable in the first-order theory $\langle \mathbb{R}, \mathbb{Z}, +, \leq, X_r \rangle$, where $X_r \subset \mathbb{R}^3$ is a base-dependent predicate such that $X_r(x, u, k)$ holds whenever u is an integer power of r, and there exists an encoding of x in which the digit at the position specified by u is equal to k.*

Theorem 2 ([11]). *Let $n \in \mathbb{N}_{>0}$ and $r \in \mathbb{N}_{>1}$. A set $S \subseteq \mathbb{R}^n$ is weakly r-recognizable iff it is r-recognizable, and it belongs to the topological class $F_\sigma \cap G_\delta$ of the metric topology over $\mathbb{R}^n$ induced by the Euclidean distance. This means that the set has to be decomposable both into a countable union of closed sets, and into a countable intersection of open sets.*

In particular, it is known that every subset of $\mathbb{R}^n$ that is definable in the first-order theory $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$, i.e., the extension of Presburger arithmetic to mixed integer and real variables, satisfies the hypotheses of Theorem 2, and it is therefore weakly recognizable in every base $r \in \mathbb{N}_{>1}$ [11].

The following theorems and lemmas introduce some operations and transformations that preserve the recognizable nature of sets.

Theorem 3. *Let $n \in \mathbb{N}_{>0}$ be a dimension, and $r, s \in \mathbb{N}_{>1}$ be* multiplicatively dependent *bases, i.e., such that $r^k = s^l$ for some $k, l \in \mathbb{N}_{>0}$. A set $S \subseteq \mathbb{R}^n$ is (resp. weakly) r-recognizable iff it is (resp. weakly) s-recognizable.*

Proof sketch. From Theorems 1 and 2, it suffices to show that definability in $\langle \mathbb{R}, \mathbb{Z}, +, \leq, X_r \rangle$ is equivalent to definability in $\langle \mathbb{R}, \mathbb{Z}, +, \leq, X_s \rangle$. Furthermore, since $r^k = s^l$, it actually suffices to establish that definability in $\langle \mathbb{R}, \mathbb{Z}, +, \leq, X_t \rangle$ is equivalent to definability in $\langle \mathbb{R}, \mathbb{Z}, +, \leq, X_{t^i} \rangle$ for all $t \in \mathbb{N}_{>1}$ and $i \in \mathbb{N}_{>0}$. An encoding of a number $x \in \mathbb{R}$ in base t^i can directly be turned into one of the same number in base t by replacing each digit (belonging to the alphabet Σ_{t^i}) into a sequence of i digits from Σ_t. The reciprocal operation is similar. It follows that the predicate X_t can be defined in terms of X_{t^i}, and reciprocally. □

Theorem 3 states that (resp. weak) recognizability in bases that are multiplicatively dependent is equivalent to (resp. weak) recognizability in one of them. In the sequel, we will thus only consider bases r and s that are multiplicatively independent.

Lemma 1. *Let $n \in \mathbb{N}_{>0}$, $S_1, S_2 \subseteq \mathbb{R}^n$, and $r \in \mathbb{N}_{>1}$. If S_1 and S_2 are both (resp. weakly) r-recognizable, then the sets $S_1 \cup S_2$, $S_1 \cap S_2$, $S_1 \setminus S_2$ and $S_1 \times S_2$ are (resp. weakly) r-recognizable as well.*

Proof sketch. The class of sets definable in $\langle \mathbb{R}, \mathbb{Z}, +, \leq, X_r \rangle$ is closed under Boolean and Cartesian operators. The same property holds for the topological class $F_\sigma \cap G_\delta$. The result then follows from Theorems 1 and 2. □

Lemma 2. *Let $n \in \mathbb{N}_{>0}$, $r \in \mathbb{N}_{>1}$, $C \in \mathbb{Q}^{n \times n}$ such that $det(C) \neq 0$, and $\boldsymbol{a} \in \mathbb{Q}^n$. If a set $S \subseteq \mathbb{R}^n$ is (resp. weakly) r-recognizable, then the set $CS + \boldsymbol{a}$ is (resp. weakly) r-recognizable as well.*

Proof sketch. The proof is by similar arguments as in that of Lemma 1. Indeed, the transformation $\boldsymbol{x} \mapsto C\boldsymbol{x} + \boldsymbol{a}$ is definable in $\langle \mathbb{R}, \mathbb{Z}, +, \leq, X_r \rangle$, and preserves the topological class $F_\sigma \cap G_\delta$. □

It is worth mentioning that, in the statement of Lemma 2, it is essential to require $\det(C) \neq 0$ as far as weak recognizability is concerned. Indeed, a transformation $\boldsymbol{x} \mapsto C\boldsymbol{x} + \boldsymbol{a}$ with a singular matrix C amounts to a *projection*, which generally alters topological properties of sets. As an example, the set $S = \{(zr^k, r^k) \mid k, z \in \mathbb{Z}\}$ belongs to $F_\sigma \cap G_\delta$, and it is actually weakly r-recognizable, whereas the set CS, with $C = \mathrm{diag}(1, 0)$, does not.

Although projection does not preserve weak recognizability, one can however sometimes extract from a set a weak recognizable set of smaller dimension. This operation is described by the following lemma.

Lemma 3. *Let $n, m \in \mathbb{N}_{>0}$, $r \in \mathbb{N}_{>1}$. If two sets $S_1 \subseteq \mathbb{R}^n$ and $S_2 \subseteq \mathbb{R}^m$ are such that $S_1 \times S_2$ is weakly r-recognizable, then S_1 and S_2 are weakly r-recognizable as well.*

Proof sketch. The proof is by similar arguments as in that of Lemma 1. □

Finally, the following result addresses the comparison of two recognizable sets.

Theorem 4. *Let $n \in \mathbb{N}_{>0}$ and $r \in \mathbb{N}_{>1}$. Two r-recognizable sets $S_1, S_2 \subseteq \mathbb{R}^n$ are equal iff they coincide over the rational vectors, i.e., iff $S_1 \cap \mathbb{Q}^n = S_2 \cap \mathbb{Q}^n$.*

Proof sketch. The vectors in $\mathbb{Q}^n$ are exactly those that are encoded by ultimately periodic words, i.e., words of the form uv^ω with $|v| \geq 1$. Two ω-regular languages are equal iff they coincide over the ultimately periodic words [16]. □

As a corollary, one can always extract a rational vector from a non-empty r-recognizable set.

3 Problem Reductions

Our main goal will be to prove that a set $S \subseteq \mathbb{R}^n$ that is recognizable or weakly recognizable in two bases r and s that are multiplicatively independent, with some possible additional restrictions on r and s, is necessarily definable in $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$. In this section, we reduce this general problem to simpler ones.

3.1 Reduction to $[0,1]^n$

It is known that a set $S \subseteq \mathbb{R}^n$ that is r-recognizable can be decomposed into a finite union $S = \bigcup_i (S_i^I + S_i^F)$, where the sets $S_i^I \subseteq \mathbb{Z}^n$ are non-empty and pairwise disjoint, and the sets $S_i^F \subseteq [0,1]^n$ are non-empty and pairwise different[2] [17,10]. This decomposition of S into sets S_i^I and S_i^F is independent from the base r. In addition, each set S_i^I is recognizable by a finite-word automaton in base r (operating only on the integer part of r-encodings), and each set S_i^F is (resp. weakly) r-recognizable if S is (resp. weakly) r-recognizable as well [17,10].

Consider two multiplicatively independent bases r and s, and a set $S \subseteq \mathbb{R}^n$ that is both r- and s-recognizable. Applying Semenov's theorem, one obtains that the sets S_i^I are definable in $\langle \mathbb{Z}, +, \leq \rangle$. It follows that, in order to prove that S is definable in $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$, it suffices to show that each set S_i^F is definable in $\langle \mathbb{R}, +, \leq, 1 \rangle$. Since this theory is closed under elimination of quantifiers [18], this is equivalent to proving that each S_i^F can be expressed as a Boolean combination of linear constraints with rational coefficients.

We have therefore reduced our main problem from $\mathbb{R}^n$ to the simpler domain $[0,1]^n$. From now on, we will stay within $[0,1]^n$ and consider that RVA only recognize the fractional part of encodings, their integer part being restricted to zero. Formally, we introduce $[w]_r$, with $w \in (\Sigma_r^n)^\omega$, as a shorthand for $[\mathbf{0} \star w]_r$.

3.2 Reduction to Product-Stable Sets

In order to be able to prove that the recognizability of a subset of $[0,1]^n$ in multiple bases leads to its definability in $\langle \mathbb{R}, +, \leq, 1 \rangle$, we need to establish a link between the arithmetical properties of this set, and the structure of automata recognizing it.

Let $n \in \mathbb{N}_{>0}$, $r \in \mathbb{N}_{>1}$, and let $S \subseteq [0,1]^n$ be a set recognized in base r by a RVA $\mathcal{A}$. We associate to each state q of $\mathcal{A}$ the language $L(q)$ accepted from q, as well as the set of vectors $S(q) \subseteq [0,1]^n$ encoded by $L(q)$, i.e., $S(q) = \{[w]_r \mid w \in L(q)\}$.

Recall that $\mathcal{A}$ is a (deterministic) Muller automaton. For each finite path $q \xrightarrow{\sigma} q'$ of $\mathcal{A}$, the language $L(q')$ can be expressed as $L(q') = \sigma^{-1}L(q) = \{w \in (\Sigma_r^n)^\omega \mid \sigma w \in L(q)\}$. Similarly, the set $S(q')$ can be expressed in terms of $S(q)$. Denoting by $[\sigma]_r$ the integer vector encoded by σ, i.e., $[\sigma]_r = [\mathbf{0}\sigma \star \mathbf{0}^\omega]_r$, we get

$$S(q') = \left\{ \boldsymbol{x} \in [0,1]^n \mid \frac{[\sigma]_r + \boldsymbol{x}}{r^{|\sigma|}} \in S(q) \right\}.$$

[2] The property is actually known for the domain $\mathbb{R}$, but straightforwardly generalizes to $\mathbb{R}^n$.

From this relation and Lemmas 1 and 2, one obtains that $S(q')$ is (resp. weakly) recognizable in all bases for which $S(q)$ is (resp. weakly) recognizable.

Consider now the particular case $q = q'$, i.e., assume that the path labeled by σ cycles from q to itself. The previous relation becomes

$$\boldsymbol{x} \in S(q) \Leftrightarrow \boldsymbol{x} \in [0,1]^n \wedge \frac{[\sigma]_r + \boldsymbol{x}}{r^{|\sigma|}} \in S(q).$$

Remark that the transformation $\boldsymbol{x} \mapsto ([\sigma]_r + \boldsymbol{x})/r^{|\sigma|}$ admits the fixed point $\boldsymbol{x} = [\sigma]_r/(r^{|\sigma|} - 1) = [\sigma^\omega]_r \in [0,1]^n$. Translating $S(q)$ so as to move this fixed point onto $\mathbf{0}$, one gets

$$\boldsymbol{x} \in S(q) - [\sigma^\omega]_r \Leftrightarrow \boldsymbol{x} \in [0,1]^n - [\sigma^\omega]_r \wedge \frac{\boldsymbol{x}}{r^{|\sigma|}} \in S(q) - [\sigma^\omega]_r.$$

This prompts the following definition, adapted from [10].

Definition 1. *Let* $n \in \mathbb{N}_{>0}$, $\boldsymbol{v} \in [0,1]^n \cap \mathbb{Q}^n$ *be a* pivot, *and* $f \in \mathbb{R}_{\geq 1}$ *be a* factor. *A set* $S \subseteq [0,1]^n$ *is* f-product-stable *with respect to the pivot* $\boldsymbol{v}$ *iff* $\forall \boldsymbol{x} \in [0,1]^n - \boldsymbol{v} : \boldsymbol{x} \in S - \boldsymbol{v} \Leftrightarrow (1/f)\boldsymbol{x} \in S - \boldsymbol{v}$.

Intuitively, that a set is f-product-stable with respect to the pivot $\boldsymbol{v}$ means that the set does not change when it magnified by the zoom factor f around the fixed point $\boldsymbol{v}$. Remark that this property is preserved by transformations of the form $\boldsymbol{x} \mapsto C\boldsymbol{x} + \boldsymbol{a}$, with $C \in \mathbb{Q}^{n \times n}$ and $\boldsymbol{a} \in \mathbb{Q}^n$, provided that $[0,1]^n \subseteq C[0,1]^n + \boldsymbol{a}$, and the new pivot $\boldsymbol{v}' = C\boldsymbol{v} + \boldsymbol{a}$ belongs to $[0,1]^n$.

In summary, if $\mathcal{A}$ recognizes the set $S \subseteq [0,1]^n$ in base r, then each reachable state q of $\mathcal{A}$ recognizes a set $S(q) \subseteq [0,1]^n$ that is (resp.weakly) recognizable in all bases for which S is (resp. weakly) recognizable. Furthermore, if there exists a cycle $q \xrightarrow{\sigma} q$, then the set $S(q)$ is $r^{|\sigma|}$-product-stable with respect to the pivot $[\sigma^\omega]_r$. We have thus established a link between a structural property of $\mathcal{A}$ (the presence of a cycle rooted at q) and an arithmetical property of $S(q)$ (its product stability).

The next step is to show that any recognizable set can be decomposed into a combination of product-stable sets that can be considered individually.

Consider the set Q_1 of states q of $\mathcal{A}$ from which there exists a cycle $q \xrightarrow{\sigma} q$, with $\sigma \in (\Sigma_r^n)^+$. Note that every infinite path of $\mathcal{A}$ eventually visits a state in Q_1. Let L be the language of words $\sigma_i \in (\Sigma_r^n)^*$ labeling finite paths $\pi = q_0 \xrightarrow{\sigma_i} q_i$ such that q_0 is the initial state of $\mathcal{A}$, $q_i \in Q_1$, $q' \notin Q_1$ for every state q' distinct from q_i visited by π, and there is only one occurrence of q_i in π. The language L is finite, and it maps each $\sigma_i \in L$ to a state q_i of $\mathcal{A}$. For each such q_i, $\mathcal{A}$ admits a cycle rooted at q_i, hence there exists $\boldsymbol{v_i} \in \mathbb{Q}^n \cap [0,1]^n$ and $l_i \in \mathbb{N}_{>0}$ such that $S(q_i)$ is r^{l_i}-product-stable with respect to the pivot $\boldsymbol{v_i}$. Note that each $S(q_i)$ is (resp. weakly) recognizable in all bases for which S is (resp. weakly) recognizable. Moreover, since $S = \bigcup_{\sigma_i \in L} (1/r^{|\sigma_i|})(S(q_i) + [\sigma_i]_r)$, we have that S is definable in $\langle \mathbb{R}, +, \leq, 1 \rangle$ if all the sets $S(q_i)$ are definable in the same theory.

It follows from this result that, in order to prove that the recognizability of a set $S \subseteq [0,1]^n$ in multiple bases implies its definability in $\langle \mathbb{R}, +, \leq, 1 \rangle$, it is sufficient to prove this property for sets S that are r^l-product-stable, with $l \in \mathbb{N}_{>0}$.

4 Recognizability in Multiple Bases

In this section, we prove the following results, which generalize to $\mathbb{R}^n$ those developed in [17,10].

Theorem 5. *Let $n \in \mathbb{N}_{>0}$ be a dimension, and $r, s \in \mathbb{N}_{>1}$ be bases with different sets of prime factors (i.e., such that there exists a prime factor of one that does not divide the other). If a set $S \subseteq [0,1]^n$ is both r- and s-recognizable, then it is definable in $\langle \mathbb{R}, +, \leq, 1 \rangle$.*

Theorem 6. *Let $n \in \mathbb{N}_{>0}$ be a dimension, and $r, s \in \mathbb{N}_{>1}$ be multiplicatively independent bases. If a set $S \subseteq [0,1]^n$ is both weakly r- and weakly s-recognizable, then it is definable in $\langle \mathbb{R}, +, \leq, 1 \rangle$.*

Our approach is by induction on n. The case $n = 1$ is an immediate consequence of [17,10]. It remains to address the inductive case where $n \geq 2$, assuming that Theorems 5 and 6 hold for smaller dimensions. Exploiting the results of Section 3.2, we only consider w.l.o.g. sets $S \subseteq [0,1]^n$ that are r^l-product-stable for some $l \in \mathbb{N}_{>0}$ and pivot $\boldsymbol{v} \in [0,1]^n \cap \mathbb{Q}^n$.

4.1 Using s-Recognizability

Consider a set $S \subseteq [0,1]^n$ that is recognizable in two bases $r, s \in \mathbb{N}_{>1}$. Assume that there exist $l \in \mathbb{N}_{>0}$ and $\boldsymbol{v} \in [0,1]^n \cap \mathbb{Q}^n$ such that S is r^l-product-stable with respect to the pivot $\boldsymbol{v}$. We show in this section that there exists an integer $l' \in \mathbb{N}_{>0}$ such that S is $s^{l'}$-product-stable as well.

We first consider the case $\boldsymbol{v} = \mathbf{0}$. Let $\mathcal{A}_s$ be a RVA recognizing S in base s. We assume w.l.o.g. that $\mathcal{A}_s$ has a complete transition relation, hence it admits an ultimately cyclic path labeled by $\mathbf{0}^\omega$, which we denote

$$q_0 \xrightarrow{\mathbf{0}^m} [q \xrightarrow{\mathbf{0}^{l'}}]^\omega,$$

where q_0 is the initial state of $\mathcal{A}_s$, $m \in \mathbb{N}$ and $l' \in \mathbb{N}_{>0}$. By the same reasoning as in Section 3.2, we obtain that the set $S(q)$ encoded in base s by the language $L(q)$ accepted from q in $\mathcal{A}_s$ is both r- and s-recognizable. Moreover, $S(q)$ is $s^{l'}$-product-stable with respect to the pivot $\mathbf{0}$.

Since $\mathcal{A}_s$ is deterministic, it admits only one path from q_0 to q labeled by $\mathbf{0}^m$, which leads to $S(q) = s^m S \cap [0,1]^n$. From this relation, and the r^l-product-stability hypothesis on S, it follows that $S(q)$ is r^l-product-stable as well, with respect to the pivot $\mathbf{0}$.

The set $S(q)$ is thus both r^l- and $s^{l'}$-product-stable, with respect to the same pivot $\mathbf{0}$. Let us show that these properties imply that S itself is both r^l- and $s^{l'}$-product-stable. By hypothesis, S is r^l-product-stable with respect to $\mathbf{0}$. For any $\boldsymbol{x} \in [0,1]^n$ and $k \in \mathbb{N}$, we thus obtain

$$\begin{aligned} \boldsymbol{x} \in S \;&\Leftrightarrow\; \frac{1}{r^{lk}}\boldsymbol{x} \in S \;\Leftrightarrow\; \frac{s^{|\sigma'|}}{r^{lk}}\boldsymbol{x} \in s^{|\sigma'|}S \\ &\Leftrightarrow\; \frac{s^{|\sigma'|}}{r^{lk}}\boldsymbol{x} \in S(q), \end{aligned}$$

if k is chosen large enough to have $r^{lk} \geq s^{|\sigma'|}$. Since $S(q)$ is $s^{l'}$-product-stable with respect to $\mathbf{0}$, we get

$$\frac{s^{|\sigma'|}}{r^{lk}}\boldsymbol{x} \in S(q) \Leftrightarrow \frac{s^{|\sigma'|-l'}}{r^{lk}}\boldsymbol{x} \in S(q)$$
$$\Leftrightarrow \frac{1}{r^{lk}s^{l'}}\boldsymbol{x} \in S \Leftrightarrow \frac{1}{s^{l'}}\boldsymbol{x} \in S.$$

proving that S is $s^{l'}$-product-stable with respect to $\boldsymbol{v}$ in the special case $\boldsymbol{v} = \mathbf{0}$.

The general case $\boldsymbol{v} \in [0,1]^n \cap \mathbb{Q}^n$ is obtained by decomposing S according to the 2^n possible positions of vectors in $[0,1]^n$ with respect to $\boldsymbol{v}$. For each vector $\boldsymbol{a} \in \{-1,1\}^n$ we introduce the matrix $M_{\boldsymbol{a}} = \mathrm{diag}(\boldsymbol{a})$, the set $D_{\boldsymbol{a}} = (\boldsymbol{v} + M_{\boldsymbol{a}}\mathbb{R}^n_{\geq 0}) \cap [0,1]^n$ and the set $S_{\boldsymbol{a}} = S \cap D_{\boldsymbol{a}}$. Each set $D_{\boldsymbol{a}}$ is a Cartesian product of intervals $D_{\boldsymbol{a}} = I_1 \times \cdots \times I_n$, where for all $i \in \{1,\ldots,n\}$,

$$I_i = \begin{cases} [v_i, 1] \text{ if } a_i = 1, \\ [0, v_i] \text{ if } a_i = -1. \end{cases}$$

The vectors $\boldsymbol{a}$ such that $D_{\boldsymbol{a}}$ has a positive volume are identified by introducing the set A of vectors $\boldsymbol{a}$ such that $f_i(v_i) > 0$ for any $i \in \{1,\ldots,n\}$, where $f_i(v_i)$ denotes the length of the interval I_i, i.e

$$f_i(v_i) = \begin{cases} 1 - v_i \text{ if } a_i = 1, \\ v_i \quad \text{if } a_i = -1. \end{cases}$$

Since each zero volume set is included into at least one positive volume set, we have $\bigcup_{\boldsymbol{a}\in A} D_{\boldsymbol{a}} = [0,1]^n$, which implies $S = \bigcup_{\boldsymbol{a}\in A} S_{\boldsymbol{a}}$.

For each $\boldsymbol{a} \in A$, the set $M_{\boldsymbol{a}}(D_{\boldsymbol{a}} - \boldsymbol{v})$ takes the form of the Cartesian product $[0, f_1(v_1)] \times \cdots \times [0, f_n(v_n)]$. We can thus map the elements of $D_{\boldsymbol{a}}$ onto $[0,1]^n$ by defining the transformation $\boldsymbol{x} \mapsto C_{\boldsymbol{a}}M_{\boldsymbol{a}}(\boldsymbol{x} - \boldsymbol{v})$, where $C_{\boldsymbol{a}} = \mathrm{diag}(1/f_1(v_1), \ldots, 1/f_n(v_n))$.

We now consider, for each $\boldsymbol{a} \in A$, the set $S'_{\boldsymbol{a}} = C_{\boldsymbol{a}}M_{\boldsymbol{a}}(S_{\boldsymbol{a}} - \boldsymbol{v})$. From the r^l-product-stability of S with respect to $\boldsymbol{v}$, it follows that $S'_{\boldsymbol{a}}$ is r^l-product-stable with respect to $\mathbf{0}$. By Lemmas 1 and 2, $S'_{\boldsymbol{a}}$ inherits the recognizability properties of S. Moreover, there exists $l'_{\boldsymbol{a}} \in \mathbb{N}_{>0}$ such that $S'_{\boldsymbol{a}}$ is $s^{l'_{\boldsymbol{a}}}$-product-stable with respect to $\mathbf{0}$. From this property and the equality $S_{\boldsymbol{a}} = \boldsymbol{v} + M_a^{-1}C_{\boldsymbol{a}}^{-1}S_{\boldsymbol{a}}$, we deduce that $S_{\boldsymbol{a}}$ is $s^{l'_{\boldsymbol{a}}}$-product-stable with respect to $\boldsymbol{v}$. From $S = \bigcup_{\boldsymbol{a}\in A} S_{\boldsymbol{a}}$, it then follows that S is $s^{l'}$-product-stable, where $l' = \mathrm{lcm}_{\boldsymbol{a}\in A}(l'_{\boldsymbol{a}})$.

In summary, we have established that S, in addition to being r^l-product-stable by hypothesis, is $s^{l'}$-product-stable as well. It remains to show that these properties, combined with our inductive hypotheses, imply that S is definable in $\langle \mathbb{R}, +, \leq, 1\rangle$.

4.2 Exploiting Multiple Product Stabilities

We thus consider a set $S \subseteq [0,1]^n$ and two bases $r, s \in \mathbb{N}_{>1}$, such that either r and s are multiplicatively independent and S is weakly r- and weakly s-recognizable,

or r and s do not share the same set of prime factors[3] and S is r- and s-recognizable. Using the results of Section 4.1, we assume that there exist $l, l' \in \mathbb{N}_{>0}$ and $\boldsymbol{v} \in [0,1]^n \cap \mathbb{Q}^n$ such that S is both r^l- and $s^{l'}$-product-stable with respect to the pivot $\boldsymbol{v}$. Our goal is to show that S is definable in $\langle \mathbb{R}, +, \leq, 1 \rangle$.

We first prove the following property.

Property 1. For each $\boldsymbol{x} \in [0,1]^n \cap \mathbb{Q}^n$ such that $\boldsymbol{x} \neq \boldsymbol{v}$, let $h_{\boldsymbol{v}}(\boldsymbol{x})$ denote the set $\{\boldsymbol{v} + \lambda(\boldsymbol{x} - \boldsymbol{v}) \in [0,1]^n \mid \lambda \in \mathbb{R}_{>0}\}$. We have either $h_{\boldsymbol{v}}(\boldsymbol{x}) \subseteq S$, or $h_{\boldsymbol{v}}(\boldsymbol{x}) \cap S = \emptyset$.

Proof sketch. Consider $\boldsymbol{x} \in [0,1]^n \cap \mathbb{Q}^n$ such that $\boldsymbol{x} \neq \boldsymbol{v}$. The set $h_{\boldsymbol{v}}(\boldsymbol{x})$ can be expressed as an intersection of linear inequalities with rational coefficients, and is thus weakly recognizable in all bases, as a consequence of Theorems 1 and 2. From Lemma 1, it follows that the set $S' = S \cap h_{\boldsymbol{v}}(\boldsymbol{x})$ is recognizable both in bases r and s, by the same type of automaton as S. Besides, S' is both r^l- and $s^{l'}$-product-stable with respect to the pivot $\boldsymbol{v}$.

Let $C \in \mathbb{Q}^{n \times n}$ be a non-singular matrix such that $C(\boldsymbol{x} - \boldsymbol{v}) = (1, 0, \ldots, 0)$. Note that the transformation $\boldsymbol{y} \mapsto C\boldsymbol{y}$ maps $h_{\boldsymbol{v}}(\boldsymbol{x})$ onto a line segment that is parallel to the first axis. From Lemmas 1 and 2, we have that the set $S'' = C(S' - \boldsymbol{v}) \cap [0,1]^n$ inherits the recognizability properties of S. Moreover, S'' is r^l- and $s^{l'}$-product-stable with respect to the pivot $\mathbf{0}$.

Note that the set S'' can be decomposed into $S'' = S''' \times \{0\}^{n-1}$, with $S''' \subseteq [0,1]$. Applying Lemma 3, the set S''' has the same recognizability properties as S hence, by the inductive hypotheses, it is definable in $\langle \mathbb{R}, +, \leq, 1 \rangle$. In other words, S''' is equal to a finite union of intervals with rational boundaries.

In addition, we know that S''' is r^l- and $s^{l'}$-product-stable with respect to the pivot 0. Since r and s are multiplicatively independent, r^l and $s^{l'}$ are multiplicatively independent as well. By Kronecker's approximation lemma, the set $\{r^{li} s^{l'j} \in]0,1] \mid i, j \in \mathbb{Z}\}$ is dense in $]0,1]$, as shown in [19,10]. It follows that if $1 \in S'''$ then $S''' \setminus \{0\} =]0,1]$ and if $1 \notin S'''$ then $S''' \setminus \{0\} = \emptyset$. As a consequence, $h_{\boldsymbol{v}}(\boldsymbol{x}) \cap S$ is either empty, or equal to $h_{\boldsymbol{v}}(\boldsymbol{x})$. □

Intuitively, Property 1 hints at the fact that the set S has a conical structure. We formalize this property by the following definition.

Definition 2. *A set $T \subseteq [0,1]^n$ is a* conical set *of vertex $\boldsymbol{v} \in [0,1]^n$ iff $\forall \boldsymbol{x} \in [0,1]^n, f \in]0,1] : \boldsymbol{x} \in T \Leftrightarrow f(\boldsymbol{x} - \boldsymbol{v}) + \boldsymbol{v} \in T$.*

In other words, a conical set is entirely determined by its vertex, and its intersection with the faces of the hypercube $[0,1]^n$. It follows that, in order to establish that S is definable in $\langle \mathbb{R}, +, \leq, 1 \rangle$, it suffices to show that this intersection is definable in the same theory, and that S is a conical set. We have the following results.

Property 2. For each $i \in \{1, 2, \ldots, n\}$ and $\lambda \in \{0, 1\}$, let $F_{\lambda,i} = \{\boldsymbol{x} \in [0,1]^n \mid x_i = \lambda\}$. The set $S \cap F_{\lambda,i}$ is definable in $\langle \mathbb{R}, +, \leq, 1 \rangle$.

[3] Note that this constraint implies that r and s are multiplicatively independent.

Proof sketch. Let $i \in \{1, 2, \ldots, n\}$ and $\lambda \in \{0, 1\}$. We first build the permutation matrix $C \in \{0,1\}^{n \times n}$ such that $C\boldsymbol{x} = (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n, x_i)$ for any $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$. The set $F_{\lambda,i}$ is definable in $\langle \mathbb{R}, +, \leq, 1 \rangle$, hence it is weakly r-recognizable. From Lemmas 1 and 2, the set $S' = C(S \cap F_{\lambda,i})$ inherits the recognizability properties of S. Moreover, we have $S' = S'' \times \{\lambda\}$, where $S'' \subseteq [0,1]^{n-1}$ has the same recognizability properties as well, as a consequence of Lemma 3. The result then follows from the inductive hypotheses. □

Property 3. The set S is conical with respect to the vertex $\boldsymbol{v}$.

Proof sketch. Let S' be the intersection of S with the faces of the hypercube $[0,1]^n$. We have established that S' is definable in $\langle \mathbb{R}, +, \leq, 1 \rangle$. The set S' can thus be expressed as a finite Boolean combination of linear constraints with rational coefficients. As a consequence, there exists a set $S'' \subseteq [0,1]^n$ that is definable in $\langle \mathbb{R}, +, \leq, 1 \rangle$, conical with respect to the vertex $\boldsymbol{v}$, and that coincides with S' over the faces of $[0,1]^n$ and over the vertex $\boldsymbol{v}$. Applying Property 1, we obtain $S'' \cap \mathbb{Q}^n = S \cap \mathbb{Q}^n$. From Theorem 4, we then have $S = S''$. □

5 Internal Structure of RVA

In Section 4, we have proved Theorems 5 and 6, which broadly state that if a set $S \subseteq [0,1]^n$ is recognizable or weakly recognizable in two bases r and s that are sufficiently different, then this set is definable in $\langle \mathbb{R}, +, \leq, 1 \rangle$. As a corollary, such sets are then weakly recognizable in every base $r \in \mathbb{N}_{>1}$ [11]. This result is significant, since it establishes that the class of weak deterministic automata is sufficient for representing all the sets that are recognizable by RVA regardless of the numeration base. As mentioned earlier, the advantage of using weak deterministic automata in actual applications comes from the fact that these automata are basically as easy to handle algorithmically as finite-word ones [20].

We now use Theorems 5 and 6, together with other results obtained in Section 4, in order to get some insight into the internal structure of RVA recognizing the subsets of $[0,1]^n$ definable in $\langle \mathbb{R}, +, \leq, 1 \rangle$. As explained in Section 3, this is equivalent to studying the structure of RVA recognizing sets definable in $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$, staying within the part of automata that reads the fractional part of vectors (i.e., in the sub-automata whose initial states are the destinations of transitions labeled by "$\star$").

Let $n \in \mathbb{N}_{>0}$ be a dimension, $r \in \mathbb{N}_{>1}$ be a base, and $S \subseteq [0,1]^n$ be a set definable in $\langle \mathbb{R}, +, \leq, 1 \rangle$. We consider a weak and deterministic RVA $\mathcal{A}$ recognizing S in base r. We assume w.l.o.g. that $\mathcal{A}$ is complete as well as minimal (in the sense of [14]).

As observed in Section 3.2, for each state q of $\mathcal{A}$, the set $S(q) \subseteq [0,1]^n$ encoded by the language $L(q)$ accepted from q can be derived from S by a transformation that is definable in $\langle \mathbb{R}, +, \leq, 1 \rangle$. It follows that each $S(q)$ is itself definable in that theory.

In addition, it has been established in Section 4.2 that, for the states q that belong to non-trivial strongly connected components of $\mathcal{A}$ (i.e., such that there exists at least one cycle from q to itself), the set $S(q)$ is a conical set. It is however worth noticing that the vertex of this conical set may not be uniquely determined. For instance, every element of the conical set $\{(0,\lambda) \mid \lambda \in [0,1]\}$ is one of its vertices. We have the following result.

Theorem 7. *Let $n \in \mathbb{N}_{>0}$, and $T \subseteq [0,1]^n$ be a conical set. The vertices of T form a bounded affine space $\{\boldsymbol{v} + \nu_1\boldsymbol{u_1} + \cdots + \nu_m\boldsymbol{u_m} \in [0,1]^n \mid \nu_1, \ldots, \nu_m \in \mathbb{R}\}$, with $m \in \mathbb{N}$ and $\boldsymbol{v}, \boldsymbol{u_1}, \ldots, \boldsymbol{u_m} \in \mathbb{R}^n$.*

Proof sketch. It is sufficient to prove that, if $\boldsymbol{v_1}, \boldsymbol{v_2} \in [0,1]^n$ are two distinct vertices of T, then each point on the line segment $L = \{\mu\boldsymbol{v_1} + (1-\mu)\boldsymbol{v_2} \in [0,1]^n \mid \mu \in \mathbb{R}\}$ linking $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$ is also a vertex of T. This can be achieved by showing that T is invariant under any translation parallel to L that stays within $[0,1]^n$, i.e., that $\boldsymbol{x} \in T \Leftrightarrow \boldsymbol{x} + \mu(\boldsymbol{v_1} - \boldsymbol{v_2}) \in T$ for all $\boldsymbol{x} \in [0,1]^n$ and $\mu \in \mathbb{R}$ such that $\boldsymbol{x} + \mu(\boldsymbol{v_1} - \boldsymbol{v_2}) \in [0,1]^n$.

Let $\boldsymbol{x}$ be an arbitrary vector in $[0,1]^n$. Consider an arbitrary value $\mu \in \mathbb{R}_{\geq 0}$ such that $\boldsymbol{x} + \mu(\boldsymbol{v_1} - \boldsymbol{v_2}) \in [0,1]^n$. (Note that restricting μ to be non negative does not weaken the property.)

We define $\boldsymbol{x'} = \boldsymbol{x} + \mu(\boldsymbol{v_1} - \boldsymbol{v_2})$ and $f = 1/(1+\mu)$. Since T is a conical set w.r.t. the vertex $\boldsymbol{v_1}$, we have $\boldsymbol{x} \in T$ if and only if $f(\boldsymbol{x} - \boldsymbol{v_1}) + \boldsymbol{v_1} \in T$. Exploiting the conical structure of T w.r.t. the vertex $\boldsymbol{v_2}$, we then get $\boldsymbol{x'} \in T$ if and only if $f(\boldsymbol{x'} - \boldsymbol{v_2}) + \boldsymbol{v_2} \in T$. By replacing $\boldsymbol{x'}$ by $\boldsymbol{x} + \mu(\boldsymbol{v_1} - \boldsymbol{v_2})$ we deduce the equality

$$f(\boldsymbol{x'} - \boldsymbol{v_2}) + \boldsymbol{v_2} \;=\; f(\boldsymbol{x} - \boldsymbol{v_1}) + \boldsymbol{v_1}$$

which yields $\boldsymbol{x} \in T \Leftrightarrow \boldsymbol{x'} \in T$. □

We are now ready to describe the structure of $\mathcal{A}$: Its initial state is the root of a (possibly empty) acyclic structure, composed of states belonging to trivial strongly connected components, and leading to states q belonging to non-trivial components. For each such state q, the set $S(q) \subseteq [0,1]^n$ is conical. Such a set is entirely characterized by the affine space containing its vertices, a Boolean value stating whether these vertices belong or not to $S(q)$, and the intersection of the set with the $2n$ faces of the hypercube $[0,1]^n$. This intersection can be expressed in terms of at most $2n$ subsets of $[0,1]^{n-1}$ (in the bottom case $n = 1$, this degenerates into the two extremities of the interval $[0,1]$), which are known to be recognizable in $\langle \mathbb{R}, +, \leq, 1 \rangle$.

Those observations could lead to a more efficient data structure for representing canonically the subsets of $[0,1]^n$ that are definable in additive arithmetic. For instance, RVA could be represented implicitly, by using BDDs for describing their initial acyclic structure [21] and linking this structure to representations of conical sets. These sets could be described by encoding separately the affine space containing their vertices and the faces of their enclosing hypercube. These faces could themselves be represented by the same type of structure applied to subsets of $[0,1]^{n-1}$. The bottom layer of such a hierarchical representation would correspond to individual rational numbers, which could be encoded explicitly.

The detailed study of such a representation system, and its application to decision procedures for $\langle \mathbb{R}, +, \leq, 1 \rangle$ and $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$, will be the subject of future work.

6 Conclusions

In this paper, we have characterized the subsets of $\mathbb{R}^n$, with $n \in \mathbb{N}_{>0}$, that are recognizable by RVA, or weak deterministic RVA, in multiple bases. Precisely, we have established that the sets that are either weakly recognizable in two multiplicatively independent bases, or recognizable in two bases that do not share the same set of prime factors, are exactly those that are definable in the first-order theory $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$. These results were already known for the particular case $n = 1$ [17,10]. They generalize to automata operating on real vectors Semenov's theorem, which states that the sets of integer vector that are recognizable in multiplicatively independent bases are those that are definable in Presburger arithmetic. The theory $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$ can indeed be seen as an extension of Presburger arithmetic to mixed integer and real variables [22]. It is worth mentioning that, in the case of (non weak) recognizability, the condition on the numeration bases cannot be replaced by multiplicative independence. Indeed, there exist subsets of $\mathbb{R}$ that are simultaneously recognizable in two multiplicatively independent bases, but without being definable in additive arithmetic [10].

An important corollary of our results is that every subset of $\mathbb{R}^n$ that is recognizable in every base $r \in \mathbb{N}_{>1}$ can be recognized by a weak deterministic automaton. This provides a theoretical justification to the use of these automata for representing sets of integer and real vectors, in addition to their practical advantages.

As an additional contribution, we have obtained interesting insight into the structure of weak deterministic automata recognizing sets definable in the theory $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$. In future work, we will address the problem of exploiting this structure in order to develop more efficient symbolic representation systems for subsets of $\mathbb{R}^n$, as well as an improved decision procedure for $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$. Our aim is to be able to benefit from the advantages of automata-based symbolic representations, which mainly reside in their easy algorithmic manipulation and their canonicity, while managing to avoid some of their drawbacks, such as the unnecessarily large size of automata obtained from some classes of constraints [23]. This could be achievable by keeping a part of the transition relation of RVA implicit. Such a representation would also simplify the problem of extracting formulas from automata recognizing arithmetic sets [24,25].

References

1. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In: Comptes Rendus du Premier Congrès des Mathématiciens des Pays Slaves, Warsaw, pp. 92–101 (1929)

2. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: Proc. International Congress on Logic, Methodoloy and Philosophy of Science, pp. 1–12. Stanford University Press, Stanford (1962)
3. Bruyère, V., Hansel, G., Michaux, C., Villemaire, R.: Logic and p-recognizable sets of integers. Bulletin of the Belgian Mathematical Society 1(2), 191–238 (1994)
4. Boigelot, B.: Symbolic methods for exploring infinite state spaces. PhD thesis, Université de Liège (1998)
5. Cobham, A.: On the base-dependence of sets of numbers recognizable by finite automata. Mathematical Systems Theory 3, 186–192 (1969)
6. Semenov, A.: Presburgerness of predicates regular in two number systems. Siberian Mathematical Journal 18, 289–299 (1977)
7. Boigelot, B., Bronne, L., Rassart, S.: An improved reachability analysis method for strongly linear hybrid systems. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 167–177. Springer, Heidelberg (1997)
8. The Liège Automata-based Symbolic Handler (LASH), `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`
9. Becker, B., Dax, C., Eisinger, J., Klaedtke, F.: LIRA: Handling constraints of linear arithmetics over the integers and the reals. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 307–310. Springer, Heidelberg (2007)
10. Boigelot, B., Brusten, J., Bruyère, V.: On the sets of real numbers recognized by finite automata in multiple bases. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 112–123. Springer, Heidelberg (2008)
11. Boigelot, B., Jodogne, S., Wolper, P.: An effective decision procedure for linear arithmetic over the integers and reals. ACM Transactions on Computational Logic 6(3), 614–633 (2005)
12. van Leeuwen, J. (ed.): Handbook of Theoretical Computer Science. Formal Models and Semantics, vol. B. Elsevier and MIT Press (1990)
13. Vardi, M.: The Büchi complementation saga. In: Thomas, W., Weil, P. (eds.) STACS 2007. LNCS, vol. 4393, pp. 12–22. Springer, Heidelberg (2007)
14. Löding, C.: Efficient minimization of deterministic weak ω-automata. Information Processing Letters 79(3), 105–109 (2001)
15. Boigelot, B., Rassart, S., Wolper, P.: On the expressiveness of real and integer arithmetic automata. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 152–163. Springer, Heidelberg (1998)
16. Perrin, D., Pin, J.: Infinite words. Pure and Applied Mathematics, vol. 141. Elsevier, Amsterdam (2004)
17. Boigelot, B., Brusten, J.: A generalization of Cobham's theorem to automata over real numbers. Theoretical Computer Science (2009) (in press)
18. Ferrante, J., Rackoff, C.: The Computational Complexity of Logical Theories. Lecture Notes in Mathematics, vol. 718. Springer, Heidelberg (1979)
19. Hardy, G.H., Wright, E.M.: An introduction to the theory of numbers, 5th edn. Oxford University Press, Oxford (1985)
20. Wilke, T.: Locally threshold testable languages of infinite words. In: Enjalbert, P., Wagner, K.W., Finkel, A. (eds.) STACS 1993. LNCS, vol. 665, pp. 607–616. Springer, Heidelberg (1993)
21. Bryant, R.: Symbolic Boolean manipulation with ordered binary decision diagrams. ACM Computing Surveys 24(3), 293–318 (1992)
22. Weispfenning, V.: Mixed real-integer linear quantifier elimination. In: Proc. ACM SIGSAM ISSAC, Vancouver, pp. 129–136. ACM Press, New York (1999)

23. Eisinger, J., Klaedtke, F.: Don't care words with an application to the automata-based approach for real addition. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 67–80. Springer, Heidelberg (2006)
24. Latour, L.: Presburger arithmetic: from automata to formulas. PhD thesis, Université de Liège (2005)
25. Leroux, J.: A polynomial time Presburger criterion and synthesis for number decision diagrams. In: Proc. 20th LICS, Chicago, pp. 147–156. IEEE Computer Society, Los Alamitos (2005)

Real World Verification

André Platzer[1], Jan-David Quesel[2], and Philipp Rümmer[3]

[1] Carnegie Mellon University, Pittsburgh, PA
aplatzer@cs.cmu.edu
[2] University of Oldenburg, Germany
quesel@informatik.uni-oldenburg.de
[3] Oxford University, Computing Laboratory, UK
Philipp.Ruemmer@comlab.ox.ac.uk

Abstract. Scalable handling of real arithmetic is a crucial part of the verification of hybrid systems, mathematical algorithms, and mixed analog/digital circuits. Despite substantial advances in verification technology, complexity issues with classical decision procedures are still a major obstacle for formal verification of real-world applications, e.g., in automotive and avionic industries. To identify strengths and weaknesses, we examine state of the art symbolic techniques and implementations for the universal fragment of real-closed fields: approaches based on quantifier elimination, Gröbner Bases, and semidefinite programming for the Positivstellensatz. Within a uniform context of the verification tool KeYmaera, we compare these approaches qualitatively and quantitatively on verification benchmarks from hybrid systems, textbook algorithms, and on geometric problems. Finally, we introduce a new decision procedure combining Gröbner Bases and semidefinite programming for the real Nullstellensatz that outperforms the individual approaches on an interesting set of problems.

Keywords: Real-closed fields, decision procedures, hybrid systems, software verification.

1 Introduction

The field of formal verification has the important ambition to check the behavior of systems by either proving the correct functioning of the system or finding bugs in its design. For several classes of systems that come from real-world domains, reasoning about real quantities is an inherent aspect of the problem. This includes (*i*) embedded systems or complex physical systems, (*ii*) formal analysis of mixed discrete/analog effects in chip design, or (*iii*) mathematical textbook algorithms, numerical algorithms or floating point arithmetic in standard programs. For domains (*i*)–(*ii*), *hybrid systems* are a common model, i.e., systems governed by interacting discrete and continuous transitions in the state space. In these domains, the need for real arithmetic reasoning comes from the temporal evolution of the continuous part of the state space, e.g., positions, velocities, analog signals. For case (*iii*), real arithmetic occurs in the computations on program data or are used as a first approximation for floating-point arithmetic.

R.A. Schmidt (Ed.): CADE 2009, LNAI 5663, pp. 485–501, 2009.

By a famous result due to Tarski [1], real arithmetic is decidable in the sense that (the first-order theory of) real arithmetic is equivalent to the first-order theory of real-closed fields, which is decidable by quantifier elimination (i.e., the process of replacing quantified formulas equivalently by quantifier-free formulas). Numerous algorithmic improvements have been made both for the handling of basic real arithmetic and for specific verification procedures for the problem domains (*i*)–(*iii*). However, for a large number of real-world systems, the underlying problems in real arithmetic still have a prohibitive complexity for quantifier elimination. Even numerical procedures for real arithmetic [2] suffer from the curse of dimensionality limiting their scalability.

In this paper we compare three state of the art approaches to reasoning about real-arithmetic in real-closed fields based on: quantifier elimination [3,4] Gröbner Bases [5], and semidefinite programming [6] for the Positivstellensatz [7]. Quantifier elimination is defined for full quantified (polynomial) nonlinear real arithmetic. The other approaches are for the universal fragment, i.e., formulas with a universal quantifier prefix. We discuss strengths and weaknesses of these approaches for formal verification and compare multiple algorithms and implementations on a set of benchmarks originating from real verification problems or interesting instances of real arithmetic. To obtain representative experimental results, we integrate all these approaches within a single uniform framework of the automated theorem prover KeYmaera for hybrid systems [8].

Finally, we introduce a new decision procedure for the universal fragment of real-closed fields that combines Gröbner Basis computations with semidefinite programming for the real Nullstellensatz [7] to avoid the scalability issues with semidefinite programming for the Positivstellensatz. Our algorithm outperforms the other algorithms on an interesting set of benchmarks.

With the goal of finding out which approaches are most suitable for real world verification problems, we provide an experimental evaluation for a wide range of techniques for real arithmetic. We contrast multiple state of the art approaches and different implementations:

1. Quantifier elimination for real-closed fields in Mathematica, QEPCAD B [9], Redlog [10], and HOL Light [11];
2. Real arithmetic handling with Gröbner Bases using external procedures in Mathematica, the Orbital library, and internally with KeYmaera proof rules;
3. Semidefinite programming relaxations [6] for the Positivstellensatz [7] using the CSDP solver [12] in our own implementation and in HOL Light [13];
4. Our new algorithm combining Gröbner Bases and semidefinite programming for the real Nulstellensatz [7] using CSDP [12] and the Orbital library.

In this paper, we consider problems in the continuous world of reals that arise in real world verification problems, including hybrid systems analysis and program verification. Our contributions are a systematic quantitative and qualitative comparison of multiple techniques for handling real arithmetic within a uniform verification framework and the introduction of a novel decision procedure for universal real arithmetic that combines Gröbner Bases with semidefinite programming for the real Nullstellensatz. We further address the question how

expensive various levels of confidence in real verification are in real examples: external (unverified) blackboxes, external blackboxes producing formally checkable certificates, and internal formal reasoning within a proof system.

2 Overall Verification Approach

We briefly discuss our formal verification approach for hybrid systems and mathematical algorithms within the automated theorem prover KeYmaera [8]. It is an implementation of a Gentzen-style sequent proof calculus for hybrid systems [14] that uses deduction modulo decision procedures for handling real arithmetic. The calculus works on sequents of the form $\phi_1, \ldots, \phi_n \vdash \psi_1, \ldots, \psi_m$ with the semantics of the formula $\bigwedge_{i=1}^{n} \phi_i \rightarrow \bigvee_{i=1}^{m} \psi_i$. Among several other rules, the calculus transforms the propositional structure into a sequent representation.

The deduction modulo calculus of KeYmaera gives us a uniform context for comparing the performance of multiple approaches and implementations for real arithmetic. The input for KeYmaera is a formula given in differential dynamic logic [14]. This logic extends first-order logic over real arithmetic by constructs for reasoning about hybrid systems as well as real-valued mathematical algorithms. For the verification task, the proof calculus transforms the input formulas into first-order formulas over real-arithmetic. For details about the proof rules of this transformation we refer to [14].

In this paper we address the question of handling the resulting real arithmetic formulas. Although first-order logic over real arithmetic is decidable by quantifier elimination [1] its complexity is doubly exponential in theory and can be high in practice. The central point of this work is to examine the question which approach to handling real arithmetic is best for which class of real world examples. We further want to determine the computational cost for techniques that provide formal proof certificates.

3 Methods for Handling Real Arithmetic

We survey different approaches to handling real arithmetic in background provers for verification. We phrase these approaches in terms of reals for simplicity. Yet, all subsequent theory in Sections 3–4 generalizes from $\mathbb{R}$ to real-closed fields.

In the sequel we assume the presence of standard rules for propositional connectives. Such rules are not presented here, as propositional reasoning is orthogonal to the handling of arithmetic. The KeYmaera system uses classical propositional sequent calculus rules; see [14,15] for details. To simplify the presentation, we further assume simple rules to normalise sequents that translate, e.g., $g \leq f$ to $f \geq g$, $f \neq g$ to $\neg(f = g)$ and $\vdash f > g$ to $f \leq g \vdash$ respectively. We assume all inequalities to be moved to the antecedent in this way.

3.1 Gröbner Bases for Real Arithmetic

Gröbner bases [5] provide a sound but incomplete procedure for proving validity of formulas in the universal fragment of equational first-order real arithmetic.

Preliminaries. Let $\mathbb{Q}[X_1, \ldots, X_n]$ be the set of *multivariate polynomials* over the indeterminates $X_1, \ldots, X_n$ with coefficients in $\mathbb{Q}$. A subset $I \subseteq \mathbb{Q}[X_1, \ldots, X_n]$ is an *ideal*, iff I is a subgroup with respect to addition and

$$rx \in I, \text{ for all } x \in I, r \in \mathbb{Q}[X_1, \ldots, X_n] \ .$$

The ideal *generated* by a set $G \subseteq \mathbb{Q}[X_1, \ldots, X_n]$ is the smallest ideal I containing G, and is denoted by (G).

The notions of Gröbner bases and polynomial reductions are relative to an admissible monomial order $\prec$, which is a strict well-order on monomials such that $uw \prec vw$ whenever $u \prec v$ for arbitrary monomials u, v, w. Admissible orders extend canonically to $\mathbb{Q}[X_1, \ldots, X_n]$ as a multiset order; see [5] for details. The monomial order determines the *leading term* in multivariate polynomials, i.e., the maximal monomial with respect to $\prec$.

Definition 1 (Reduction). *Let $f, g \in \mathbb{Q}[X_1, \ldots, X_n]$. We say that f* reduces to *g with respect to $G \subset \mathbb{Q}[X_1, \ldots, X_n]$ iff for some $m \in \mathbb{N}$ there are $f_0, f_1, \ldots, f_m$ in $\mathbb{Q}[X_1, \ldots, X_n]$ with $f_0 = f, f_m = g$ such that, for all i, $f_{i+1} = f_i - h_i g_i$ for some $h_i \in \mathbb{Q}[X_1, \ldots, X_n]$, $g_i \in G$, and $f_{i+1} \prec f_i$. We write $g = \mathrm{red}_G f$ if, in addition, g cannot be reduced further, i.e., there is no $h_{m+1} \in \mathbb{Q}[X_1, \ldots, X_n]$ and $g_{m+1} \in G$ with $g - h_{m+1}\, g_{m+1} \prec q$.*

Definition 2 (Gröbner basis). *A finite subset G of an ideal I of $\mathbb{Q}[X_1, .., X_n]$, is called* Gröbner basis *iff $I = (G)$ and $\mathrm{red}_G f$ is unique for all polynomials f. Further G is* reduced *if no $g \in G$ can be reduced further with respect to $G \setminus \{g\}$.*

There are several equivalent alternative formulations of this definition, for instance that $\mathrm{red}_G f = 0$ iff $f \in I$. This means that Gröbner bases solve the *ideal membership problem* and can, thus, directly be used as an (incomplete) proof rule for equational arithmetic.

Gröbner Basis Eliminations. The most naive use of Gröbner bases for real arithmetic is described by the rules A1, A2 in Fig. 1. The rule A1 closes a goal if the ideal G generated by equations in the antecedent contains 1, which (by Hilbert's Nullstellensatz) implies that the equations do not have common solutions (i.e., are contradictory). Similarly, A2 can be applied if the sides f, g of an equation in the succedent have the same remainder modulo G, which means $f - g \in (G)$.

The scope of the rules can be extended by testing for *radical membership* instead of ideal membership, which can prove problems like $x^2 = 0 \vdash x = 0$ that A2 cannot prove. The *radical* of an ideal I is the set

$$\sqrt{I} = \bigcup_{i=1}^{\infty} \{g \in \mathbb{Q}[X_1, \ldots, X_n] \ : \ g^i \in I\} \supseteq I$$

Because the inclusion $I \subseteq \sqrt{I}$ can be strict (e.g., $\sqrt{(x^2)} = (x)$), testing for radical membership is more liberal than ideal membership, while still being sound.

In practice, the rule A3, which is known as *Rabinowitch's trick*, represents a simple way of testing for radical membership. It is based on the observation that

$$\text{(A1)}\ \frac{*}{\Gamma, g_1 = \tilde{g}_1, \ldots, g_n = \tilde{g}_n \vdash \Delta} \qquad \text{(A4)}\ \frac{\Gamma, f - g = z^2 \vdash \Delta}{\Gamma, f \geq g \vdash \Delta}$$

$$\text{(A2)}\ \frac{*}{\Gamma, g_1 = \tilde{g}_1, \ldots, g_n = \tilde{g}_n \vdash f = h, \Delta} \qquad \text{(A5)}\ \frac{\Gamma, (f - g)z^2 = 1 \vdash \Delta}{\Gamma, f > g \vdash \Delta}$$

$$\text{(A3)}\ \frac{\Gamma, (f - g)z = 1 \vdash \Delta}{\Gamma \vdash f = g, \Delta} \qquad \text{(A6)}\ \frac{\Gamma \vdash 1 + s_1^2 + \cdots + s_n^2 = 0, \Delta}{\Gamma \vdash \Delta}$$

In all rules, z is a fresh variable. With the Gröbner basis G of the ideal $(g_1 - \tilde{g}_1, \ldots, g_n - \tilde{g}_n)$, rule A1 is applicable if $\mathrm{red}_G\, 1 = 0$, and A2 if $\mathrm{red}_G\, f = \mathrm{red}_G\, h$. Rules similar to A2, A4 and A5 can be defined for inequalities in the succedent. In A6, the polynomials $s_1, \ldots, s_n$ can be chosen arbitrarily.

Fig. 1. Rule schemata of Gröbner calculus rules

$g \in \sqrt{I}$ if and only if $1 \in (I \cup \{gz - 1\})$ (where z is a fresh indeterminate). The latter property can be tested by first applying A3 and then A1.

Finally, inequalities can be translated to equations using A4, A5, which exploit the fact that a real number is positive iff it is a square (A5 is an optimized version including Rabinowitch's trick). Combined with the rules A1, A2, this encoding of inequalities is rather weak, and not able to derive simple facts like $a \leq b \wedge b \leq c \rightarrow a \leq c$. It is, however, an important preprocessing step for the complete procedure described in the next section (where we explain rule A6).

Proposition 1 (Soundness). *The Gröbner basis rules in Fig. 1 are sound. Rules A3, A4, A5 are even satisfiability-equivalent transformations, i.e., their respective premisses and conclusions are satisfiability-equivalent. (See [16]).*

The Gröbner basis approach gives a sound but incomplete overapproximation. To see why Gröbner bases are incomplete for real arithmetic, consider the following. Gröbner bases are a general approach and do not take into account the special properties of the reals. For instance, the sequent $x^2 = -1 \vdash$ is valid, i.e., the formula $x^2 = -1$ is unsatisfiable over $\mathbb{R}$, but the Gröbner basis of $x^2 + 1$ is $\{x^2 + 1\}$ and, in fact, $x^2 = -1$ is satisfiable over $\mathbb{C}$ but not over $\mathbb{R}$.

Implementations. We compare three implementations of the Gröbner basis rules:

GM. The implementation provided by the Mathematica 7.0 computer algebra system, which can be used as a reasoning back-end by KeYmaera.

GO. The implementation of Buchberger's algorithm [5] in the open-source Java-library Orbital (written by the first author of this paper).

GK. An implementation of Buchberger's algorithm with (verified) proof rules that are directly defined within KeYmaera. This procedure generalizes a calculus for integer arithmetic [17] to the reals, and differs from GM and GO in that it does not use the rules A3, A4, A5, but instead integrates the Fourier-Motzkin variable elimination rule [18] to handle inequalities (which is complete for linear arithmetic). This tight integration of the two procedures can simplify terms in inequalities using Gröbner bases, and can feed

equations derived by the Fourier-Motzkin procedure back to Buchberger's algorithm. We evaluate the benefits of this cooperation in Sect. 5. Since our domain are the reals, we do not use the heuristic approach tailored to nonlinear integer inequalities from [17].

3.2 A Complete Rule Using the Real Nullstellensatz

While the rules A1, A2, A3, A4, A5 only form an incomplete calculus for problems in real arithmetic, the situation is different over the complex numbers: Hilbert's Nullstellensatz tells that A1, A3 together yield a decision procedure for universal equational problems in $\mathbb{C}$. A corresponding result for real-closed fields is Stengle's real Nullstellensatz [7]; also see [13]:

Theorem 1 (Nullstellensatz [7] for real-closed fields). *Let R be a real-closed field (e.g., $R = \mathbb{R}$) and G be a finite subset of $R[X_1, \ldots, X_n]$. Then the set $\{x \in R^n \ : \ g(x) = 0$ for all $g \in G\}$ is empty if and only if there are polynomials $s_1, \ldots, s_m \in R[X_1, \ldots, X_n]$ such that $1 + s_1^2 + \cdots + s_m^2 \in (G)$. If, moreover, $G \subseteq \mathbb{Q}[X_1, \ldots, X_n]$, then also the polynomials $s_1, \ldots, s_m$ can be chosen among the elements of $\mathbb{Q}[X_1, \ldots, X_n]$.*

This theorem leads to an extremely simple, yet complete, proof method for the universal fragment of real arithmetic: in addition to the rules that we have already discussed, we add rule A6 in Fig. 1 for injecting the equation $1 + s_1^2 + \cdots + s_m^2 = 0$ into a proof goal. Any valid proof goal can then be closed in the following way: (*i*) inequalities and equations in the succedent are turned into equations in the antecedent with the help of A3, A4, A5, (*ii*) the *witness* $1 + s_1^2 + \cdots + s_m^2$ due to the real Nullstellensatz is generated using A6, and (*iii*) the goal is closed by the Gröbner Basis computations with A2.

Corollary 1 (Completeness). *Along with propositional rules, the rules in Fig. 1 are complete for the universal fragment of real arithmetic.*

Proof. Completeness follows from Theorem 1 using the satisfiability-equivalence properties for the transformation by A3, A4, A5 according to Proposition 1. □

The main difficulty with this calculus is obvious: it does not provide any guidance for choosing the witness $1 + s_1^2 + \cdots + s_m^2 = 0$. One technique to tackle the required search is semidefinite programming, following the work based on Stengle's Positivstellensatz (Sect. 3.4) in [6,13]. We describe a new approach that combines semidefinite programming with Gröbner bases in Sect. 4.

Example 1. In Fig. 2, we show a proof for the following implication (leaving out propositional reasoning):

$$x \geq y \wedge z \geq 0 \rightarrow xz \geq yz. \tag{1}$$

The inequalities $x \geq y$ and $z \geq 0$ are turned into equations using A4. Proving by contradiction (or using propositional rules), the conclusion $xz \geq yz$ is considered as an assumption $yz > xz$ and subsequently eliminated with the help of

$$\dfrac{\dfrac{\dfrac{}{x-y=a^2, z=b^2, (yz-xz)c^2=1 \vdash 1+(abc)^2=0}\,{\scriptstyle *}}{x-y=a^2, z=b^2, (yz-xz)c^2=1 \vdash}\,{\scriptstyle \text{A6}}}{x \geq y, z \geq 0, yz > xz \vdash}\,{\scriptstyle \text{A2}}\quad{\scriptstyle \text{A4,A5}}$$

Fig. 2. Example proof using the real Nullstellensatz

A5. Once this is done, we rely on an oracle to tell us the witness $1+(abc)^2$, which is introduced using A6. Finally, the proof can be closed by A2: the set $\{a^2-x+y,\, b^2-z,\, xzc^2-yzc^2+1\}$ is a Gröbner basis representing the equations in the antecedent. The basis reduces the term $1+(abc)^2$ to 0 as follows:

$$1+a^2b^2c^2 \overset{b^2-z}{\rightsquigarrow} 1+a^2zc^2 \overset{a^2-x+y}{\rightsquigarrow} 1+xzc^2-yzc^2 \rightsquigarrow 0$$

3.3 Quantifier Elimination in Real-Closed Fields

A general method for handling quantified real arithmetic is based on the seminal work by Tarski [1]. He showed that there is an algorithm computing a quantifier-free formula that is equivalent to a given formula in (first-order) real arithmetic.

Theorem 2 (Quantifier elimination [1]). *The first-order theory of reals (or of real-closed fields) admits* quantifier elimination*, i.e., to each first-order formula ϕ, a quantifier-free formula* $\mathrm{QE}(\phi)$ *can be associated effectively that is equivalent and has no additional free variables. Thus* QE *yields a decision procedure for closed formulas when evaluating the remaining quantifier-free formulas.*

Unlike the other approaches outlined in this paper, QE directly applies to full nonlinear (polynomial) real arithmetic and not just to the universal fragment. QE is also independent of propositional rules, except that computational efficiency considerations advise to combine both [19].

Example 2. For instance, QE yields the following equivalence:

$$\exists x\,(ax^2+bx+c=0) \;\equiv\; a \neq 0 \wedge b^2-4ac \geq 0 \vee a=0 \wedge (b=0 \rightarrow c=0)$$

Tarski's approach has been extended to practical algorithms [3,4], which are quite sophisticated. Unfortunately, the complexity of QE is doubly exponential in the number of quantifier alternations [20].

Implementations. We compare six implementations of QE in experiments:

QQ. Partial cylindrical algebraic decomposition (PCAD) [3] in QEPCAD B [9];
QM. QE based on partial CAD [3] and validated numerics [21] in Mathematica;
QR$_c$. Partial CAD [3] in Redlog [10];
QR$_s$. Virtual substitution [4] in Redlog [10], falling back to **QR$_c$**;
QC. Harrison's implementation of Cohen-Hörmander quantifier elimination;
QH. Proof-producing quantifier elimination [11] in HOL Light.

3.4 Semidefinite Programming for the Positivstellensatz

The Positivstellensatz for real-closed fields [7] is a generalisation of the real Nullstellensatz. It gives rise to a sound and complete proof method for the universal fragment of first-order real arithmetic that does not require the reductions A3, A4, A5. The Positivstellensatz has recently been exploited in combination with relaxations from semidefinite programming [6,13].

The *multiplicative monoid* $\mathrm{mon}(H)$ generated by $H \subseteq R[X_1, \ldots, X_n]$ is the set of finite products of elements of H (including the empty product 1). The *cone* $\mathrm{con}(F)$ generated by a set $F \subseteq R[X_1, \ldots, X_n]$ is the smallest set containing F and squares s^2 of arbitrary polynomials $s \in R[X_1, \ldots, X_n]$ that is closed under addition and multiplication. For more computational representations of cones and ideals, we refer to [6,22].

Theorem 3 (Positivstellensatz [7] for real-closed fields). *Let R be a real-closed field (e.g., $R = \mathbb{R}$) and F, G, H finite subsets of $R[X_1, \ldots, X_n]$. Then*

$$\{x \in R^n \ : \ f(x) \geq 0 \text{ for all } f \in F,\ g(x) = 0 \text{ f.a. } g \in G,\ h(x) \neq 0 \text{ f.a. } h \in H\}$$

is empty iff

$$\text{there are } s \in \mathrm{con}(F), g \in (G), m \in \mathrm{mon}(H) \text{ such that } s + g + m^2 = 0 \ .$$

If, moreover, $F, G, H \subseteq \mathbb{Q}[X_1, \ldots, X_n]$, then also the polynomials s, g, m can be chosen among the elements of $\mathbb{Q}[X_1, \ldots, X_n]$.

The polynomials s, g, m are polynomial infeasibility witnesses. For bounded degree, witnesses s, g, m can be searched for using numerical semidefinite programming [6] by parameterising the resulting polynomials. As (theoretical) degree bounds exist for the certificate polynomials s, g, m, the Positivstellensatz yields a decision procedure. These bounds are, however, at least triply exponential [6]. Thus, the approach advocated by Parrilo [6] is to increase the bound successively and solve the existence of bounded degree witnesses due to the Positivstellensatz by semidefinite programming [23].

As a simple corollary to Theorem 3 we have that A7 is a sound proof rule.

Corollary 2 (Soundness). *The rule in Fig. 3 is sound.*

In contrast to the rules in Fig. 1 the only additional transformation necessary for rule A7 is a reduction from $>$ to $\geq$ via $f > g \leftrightarrow f \geq g \wedge f \neq g$. All other transformations follow from the propositional sequent calculus rules and the rewriting

$$\text{(A7)}\ \frac{*}{f_1 \geq \tilde{f}_1, \ldots, f_m \geq \tilde{f}_m, g_1 = \tilde{g}_1, \ldots, g_n = \tilde{g}_n \vdash h_1 = \tilde{h}_1, \ldots, h_l = \tilde{h}_l}$$

A7 is applicable iff $s + g + m^2 = 0$ for some $s \in \mathrm{con}(\{f_1 - \tilde{f}_1, \ldots, f_m - \tilde{f}_m\})$, some $g \in (g_1 - \tilde{g}_1, \ldots, g_n - \tilde{g}_n)$, and some $m \in \mathrm{mon}(\{h_1 - \tilde{h}_1, \ldots, h_l - \tilde{h}_l\})$.

Fig. 3. Rule schemata of Positivstellensatz calculus rules

$$\dfrac{\dfrac{}{x \geq y, z \geq 0, (yz - xz)c^2 = 1 \vdash}\,{}^{*}\,{}^{\text{A7}}}{x \geq y, z \geq 0, yz > xz \vdash}\,{}^{\text{A5}}$$

Fig. 4. Example proof using the Positivstellensatz

rules described in the beginning of Sect. 3. Therefore, this approach does not introduce new variables, as it does not need the rules A3 – A5. Alternatively, A5 can be used in place of the $f > g$ axiomatisation as we show in the sequel.

Example 3. A proof for the implication (1) that uses the Positivstellensatz is in Fig. 4. In contrast to the proof in Fig. 2, it is now unnecessary to eliminate the inequalities $x \geq y$ and $z \geq 0$, while the rule A5 has to be used for $xz \geq yz$ (corresponding to $yz > xz$ in the antecedent). A witness for the problem is:

$$\underbrace{c^2 \cdot (x - y) \cdot z}_{s} + \underbrace{(yz - xz)c^2 - 1}_{g} + \underbrace{1}_{m^2} \;=\; 0$$

The terms $x - y$ and z in s stem from the inequalities in the sequent, while the term g is derived from the equation.

Implementations. We compare two implementations using the semidefinite programming optimization tool CSDP [12] to find witnesses for the Positivstellensatz:

PH. John Harrison's implementation [13] in HOL Light.

PK. Our implementation within KeYmaera directly follows the approach presented by Parrilo [6] and Harrison [13]. We follow Parrilo's enumeration of polynomials without further optimization.

4 Gröbner Bases for the Real Nullstellensatz (GRN)

We describe a new approach to turn the complete calculus based on the real Nullstellensatz (NSS, Theorem 1) into an effective proof procedure. While our method is strongly inspired by, and in parts based on, semidefinite programming for the Positivstellensatz (PSS, Theorem 3) [6,13], there are two main motivations to deviate from this approach: (*i*) the application of the PSS requires reasoning about ideal membership (the set (G) in Theorem 3) and, thus, to solve systems of polynomial equations. This is an incentive to integrate Gröbner bases as a computational, efficient, and well-studied method to this end; (*ii*) the PSS requires constructing three witnesses s, g, m simultaneously, which makes it intricate to balance degree bounds and the number of parameters to be determined by semidefinite programming. Using a combination of Gröbner basis computations and the single witnesses of the real NSS, we avoid these issues.

In order to prove by NSS that a set G of polynomials does not have common zeroes, we need to find polynomials $s_1, \ldots, s_m$ such that $1 + s_1^2 + \cdots + s_m^2 \in (G)$.

We reduce this problem to a search for positive semidefinite matrices with the help of the following lemma. A matrix $X \in \mathbb{R}^{k\times k}$ is called *positive semidefinite* (PSD) if it is symmetric, and if $x^t X x \geq 0$ for each vector $x \in \mathbb{R}^k$. There is a simple correspondence between PSD matrices and sums of squares:

Lemma 1. *Suppose $p \in \mathbb{Q}[X_1, \ldots, X_n]^k$ is a vector of rational polynomials. The following identities hold (for the proof see [16]):*

$$\begin{aligned} & \left\{ \sum_{i=1}^{l} (c_i p)^2 \ : \ l \in \mathbb{N}, c_i \in \mathbb{Q}^k \right\} \\ = & \left\{ \sum_{i=1}^{l} \alpha_i (c_i p)^2 \ : \ l \in \mathbb{N}, \alpha_i \in \mathbb{Q}, \alpha_i \geq 0, c_i \in \mathbb{Q}^k \right\} \\ = & \left\{ p^t X p \ : \ X \in \mathbb{Q}^{k\times k} \text{ positive semidefinite} \right\} \end{aligned}$$

By combining Lemma 1 with the NSS, we see that a set G of polynomials does not have any common zeroes if and only if there is a vector p of polynomials and a PSD matrix $X \in \mathbb{R}^{k\times k}$ such that $1 + p^t X p \in (G)$. As the vector space of polynomials is generated by monomials, it is sufficient to consider vectors p of monomials.

Semidefinite programming [23] provides a simple method to determine such matrices X. A *semidefinite program* (SDP) is an optimisation problem in terms of traces (tr) of matrices:

maximise	$\mathrm{tr}(CX)$
subject to	$\mathrm{tr}(A_i X) = b_i \quad (\text{for } i \in \{1, \ldots, n\})$,
where	X positive semidefinite

where $A_i, C \in \mathbb{R}^{k\times k}$ are symmetric matrices and $b_i \in \mathbb{R}$. Such optimisation problems can be solved efficiently using numerical convex optimization [23].

The key insight underlying our method is the following: by computing a Gröbner basis B for the ideal (G), the NSS condition $1 + p^t X p \in (G)$ can be encoded as the linear side constraints $\mathrm{tr}(A_i X) = b_i$ ($i \in \{1, \ldots, n\}$) of a semidefinite program searching for X. To see this, note that both the expression $1 + p^t X p$ and the reduction $\mathrm{red}_B(1 + p^t X p)$ are linear in X. Because Gröbner bases determine unique remainders, we therefore have $1 + p^t X p \in (G)$ if and only if $\mathrm{red}_B(1 + p^t X p) = 0$. This equation is a linear constraint on X suitable for SDP.

To capture this observation formally, let Q be a symmetric $k \times k$ matrix of parameters:

$$Q \ = \ \begin{pmatrix} q_{1,1} & q_{1,2} & \cdots & q_{1,k} \\ q_{1,2} & q_{2,2} & \cdots & q_{2,k} \\ \cdots & \cdots & \cdots & \cdots \\ q_{1,k} & q_{2,k} & \cdots & q_{k,k} \end{pmatrix}$$

The polynomial $1 + p^t Q p$ is linear in Q and can be represented in the form $1 + p^t Q p = q^t C m$, where $q = (q_{1,1}, q_{1,2}, \ldots, q_{k,k})^t$ is the vector of all the Q-parameters, $m = (m_1, \ldots, m_s)^t$ is a vector of monomials over $X_1, \ldots, X_n$ (containing, at least, 1 and all products $p_i p_j$ of components of p), and $C \in \mathbb{Q}^{k^2 \times s}$

is a matrix. By computing the remainder $q^tDm = \mathrm{red}_B(q^tCm)$ of this term for a Gröbner basis B over $\mathbb{Q}[X_1, \ldots, X_n]$, we can construct the required side constraints:

Lemma 2. *Suppose that the components of m are pairwise distinct, and that q^tCm and q^tDm are two polynomials over $\mathbb{Q}[q_{1,1}, q_{1,2}, \ldots, q_{k,k}][X_1, \ldots, X_n]$ defined by the matrices $C, D \in \mathbb{Q}^{k^2 \times s}$, such that $q^tDm = \mathrm{red}_B(q^tCm)$. Then the following equation holds (see [16] for a proof):*

$$\{x \in \mathbb{R}^k : red_B(x^tCm) = 0\} \;=\; \{x \in \mathbb{R}^k : x^tD = 0\} \tag{2}$$

Example 4. We return to the implication (1) proven in Fig. 2 by showing that the polynomials $B = \{a^2 - x + y,\, b^2 - z,\, xzc^2 - yzc^2 + 1\}$ have no common zeroes. The witness $1 + (abc)^2$ used in the proof of Fig. 2 can be constructed systematically for a suitable set of basis monomials, say, $p = (1, a^2, abc)^t$. We need to find a PSD matrix $X \in \mathbb{Q}^{3\times 3}$ such that $1 + p^tXp \in (B)$. To do so, we compute the reduction $\mathrm{red}_B(1 + p^tQp)$ for a symbolic 3×3 parameter matrix Q:

$$\begin{aligned}
&\mathrm{red}_B(1 + p^tQp)\\
&= \mathrm{red}_B(1 + q_{1,1}1^2 + 2q_{1,2}a^2 + 2q_{1,3}abc + 2q_{2,3}a^3bc + q_{3,3}a^2b^2c^2)\\
&= 1 + q_{1,1} - q_{3,3} + 2q_{1,2}x - 2q_{1,2}y + 2q_{1,3}abc + 2q_{2,3}abcx - 2q_{2,3}abcy
\end{aligned}$$

By comparing coefficients, the constraints on Q for this polynomial to be 0 are:

$$\begin{aligned}
1 + q_{1,1} - q_{3,3} &= 0 & -2q_{1,2} &= 0 & 2q_{2,3} &= 0\\
2q_{1,2} &= 0 & 2q_{1,3} &= 0 & -2q_{2,3} &= 0
\end{aligned}$$

A positive semidefinite solution of the constraints is $q_{3,3} = 1$ and $q_{i,j} = 0$ for all $(i, j) \neq (3, 3)$, which means $1 + p^tQp = 1 + (abc)^2$.

Theorem 4 (Completeness). *By enumerating all monomials for p successively, Gröbner bases for the real Nullstellensatz give a complete method for universal real arithmetic: If the original formula is valid, then, when p contains all monomials of a sufficiently large degree, the corresponding semidefinite programs will have a solution (the witness).*

Proof. The proof is a combination of Lemma 2 with Corollary 1.

4.1 Discussion and Practical Considerations

Semidefinite programming turns the search for witnesses $1 + s_1^2 + \cdots + s_m^2$ into a (simpler) search for suitable basis monomials p. As the number of basis monomials that need to be considered is finite (due to degree bounds on witnesses [6]), this yields a theoretical decision procedure. Practically, we enumerate all monomials with ascending degree. There might be more sophisticated methods, however: the number of monomials that witnesses are actually built of is usually small, and it might be possible to locate likely candidates by analyzing the

Gröbner basis B. In our experience, the number of basis monomials that are considered before a solution is found (and thus the difficulty of a problem) depends on (*i*) the number of variables in the polynomial ring, and (*ii*) the degree of the leading monomials in the Gröbner basis.

Another issue is that implementations for semidefinite programming (like the CSDP solver [12] used by us) are numerical and produce answers in floating point arithmetic. To recover precise solutions in $\mathbb{Q}$ from such answers, we use a similar approach as in [13]: We approximate floating point numbers to a certain precision by rationals (with the help of Stern-Brocot trees [24]), and check resulting solution candidate for semidefiniteness. We increase the precision successively as long as the solution candidate remains indefinite.

Optimizations. We found it essential to use preprocessing steps to reduce the number of variables in a problem, such that the number of potential basis monomials becomes tractable. Some heuristics are:

- If the Gröbner basis B contains a polynomial $x + t$ such that x does not occur in t, then x and the polynomial can be eliminated by simple rewriting.
- If B contains polynomials $xy - 1$ and $x^n + t$ such that x^n does not divide t, then x and the polynomial $xy - 1$ can be eliminated by multiplying each polynomial in B (except $xy - 1$) with a power of y and reducing w.r.t. $xy - 1$.
- Polynomials $\alpha_1 m_1^2 + \cdots + \alpha_n m_n^2 \in B$ such that $\alpha_i > 0$ for $i \in \{1, \ldots, n\}$ can be replaced by the monomials $m_1, \ldots, m_n$.
- If B contains a polynomial $\alpha_0 x^2 - \alpha_1 m_1^2 - \cdots - \alpha_n m_n^2$ such that $\alpha_i > 0$ for $i \in \{0, \ldots, n\}$ where x only occurs with even degree in B, then x can be eliminated by rewriting and the polynomial can be removed.

The last two cases are surprisingly common, due to the encoding of inequalities by quadratic terms performed by A4 and A5.

5 Experimental Results

We have integrated the techniques presented in Sect. 3–4 into KeYmaera. With the various methods for real arithmetic integrated into a common framework and real arithmetic examples from different domains, we have a solid base for our experiments. The benchmarks[1] are a collection of challenging arithmetic problems from the hybrid system world [25], the verification of invariant properties for mathematical algorithms [26,27] and algebraic geometry [28], as well as a smaller number of synthetic problems. For the examples with mixed quantifiers, our setting applies QM to the existential quantifiers such that we can still gain insight into the scalability of the approaches that are restricted to the universal fragment on these examples. We run our experiments on a dual Intel Xeon E5430 (quad core with 2.66 GHz) and 32 gigabytes RAM.

[1] Available along with KeYmaera from `http://symbolaris.com/info/KeYmaera.html`

The experimental results [16] summarized in Fig. 5 show that, for our particular mix of examples, quantifier elimination procedures are still faster than recent approaches with semidefinite programming relaxations for the Positivstellensatz, while Gröbner bases alone have difficulties with "real" problems. As expected, procedures tailored for real arithmetic can solve substantially more cases than Gröbner bases for general fields. Gröbner bases that integrate Fourier-Motzkin (GK) solve many more problems.

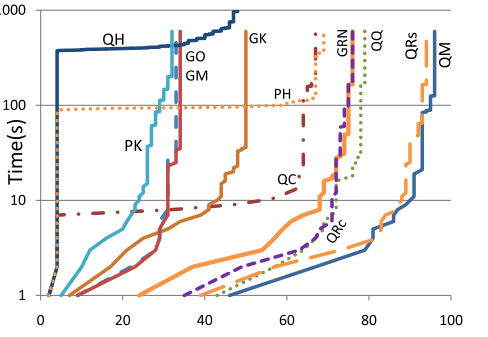


Fig. 5. Examples solved per time

Our combination, GRN, of Gröbner bases with the real Nullstellensatz is competitive with quantifier elimination by partial CAD [3]. The experiments also show that substantial performance improvements (QR_s and QM) are still possible beyond partial CAD. Another interesting observation from the experiments is that the Positivstellensatz (PH and PK) and our GRN approach complement each other quite well. PH and GRN together can solve 84 out of 97 problems [16].

The experiments show that GM and GO are on a par. Further, QM and QR_s are very close, but clearly outperform QR_c, QQ and QC both in runtime and number of provable cases. QH is slower but competitive with the number of examples solved by GK but does not yet perform as well as other QE implementations or GRN. The performance gap between PK and PH is surprising. In part, it shows how important Harrison's optimizations [13] of Parrilo's work [6] are, but may also be caused by different heuristics for recovering rationals from floats and different enumeration orders for polynomials. This might indicate that PK, indeed, gives a more objective comparison for GRN than PH, because PK and GRN share exactly the same KeYmaera framework and rational recovering. Our new GRN procedure is a clear win compared to PK. Inevitably, performance depends on the system options and on the set of benchmarks.

6 Related Work

Nipkow [29] presented a formally verified implementations of quantifier elimination in an executable fragment of Isabelle/HOL, currently for linear real arithmetic only. McLaughlin and Harrison [11] presented a nonverified but proof-producing implementation of general quantifier elimination, so that the result of the procedure can be checked independently.

The sum of squares approach has been pioneered by Parrilo [6] and Harrison [13]. Harrison also gives optimizations for the univariate case.

Tiwari [30] presents an approach using Gröbner bases and sign conditions on variables to produce unsatisfiability witnesses for nonlinear constraints. The

approach depends on appropriate heuristic variable orderings that are formed by successively introducing new variables for polynomial expressions following certain heuristics (which may not terminate). Our work and that of Tiwari share the combination of Gröbner bases with witness generation. Yet we follow semidefinite programming for the real Nullstellensatz, whereas [30] uses heuristic generation of polynomial witness expressions. Tiwari uses the Positivstellensatz to prove refutational completeness but not as part of his technique.

RSolver [2] is a numerical approach for deciding validity of (robust instances of) first-order formulas over real arithmetic extended with transcendental functions. Unlike our work, this relies on numerical stability of the input formula.

MetiTarski [31] is an interesting approach for handling special functions using a combination of resolution proving with simple QE procedures. Their focus is on handling special functions not on handling real arithmetic.

Hunt et al. [32] describe the handling of nonlinear arithmetic in ACL2, which is based on heuristic multiplication of inequalities in the style of (1) and yields an incomplete method. The method is claimed to be empirically successful, though, and can also be applied to nonlinear integer arithmetic.

7 Discussion and Conclusions

The respective approaches from Sect. 3–4 have different advantages and weaknesses for formal verification of real world problems in real arithmetic. We draw a qualitative comparison complementing the quantitative comparison from Sect. 5.

Quantifier Elimination. Quantifier elimination procedures [3] can handle full nonlinear real arithmetic, including existential quantifiers. Their implementations are quite intricate algorithms for which correctness is not easily established formally. Unfortunately, QE does not produce simple checkable certificates.

Proof-producing [11] or verified [29] QE procedures may be interesting improvements on the formal traceability of QE. Unfortunately, their performance is not yet fully competitive with other quantifier elimination implementations or our new proof-producing GRN procedure.

A compromise is reverification: Proof search [33,19] in KeYmaera generates several problems of real arithmetic to find a proof, but only those in the final proof are soundness-critical. For soundness, it is sufficient to use a fast or untrusted implementation of QE during the proof search and to reverify the final proof in a proof checker with a verified or proof-producing QE implementation [11,29]. For this purpose, KeYmaera strategies are especially useful that identify the sweetspot for applying QE iteratively during the proof search [19].

Positivstellensatz. In the context of verification, a useful property of the Positivstellensatz is that it produces a witness ($s + g + m^2 = 0$) for the validity of a formula. Once the witness has been found, it is checkable by simple computations in the polynomial ring to determine whether the polynomial identity holds by comparing the coefficients. Similarly, the well-formedness of the witness can be

determined by checking whether s is build from sums of squares using an extension of "completing the square" [13]. Thus, complicated numerical semidefinite programming tools [23] do not need to be part of the trusted computing base concerning soundness. Due to its enumerative nature with a large number of extra parameters, scalability with the number of variables is still limited.

Gröbner Bases. The Gröbner Basis approach does not have simple witnesses like Positivstellensatz approaches. Their working principle, however, is strictly based on symbolic computations, which can be carried out from a small set of rewrite rules within a logic. This corresponds to our built-in Gröbner basis approach GK, which is almost as efficient as external Gröbner basis implementations. Our experimental results indicate that, due to the partial ignorance of real-closed field properties, the capabilities of Gröbner bases alone are not sufficient, even in combination with Fourier-Motzkin elimination.

Real Nullstellensatz. Our new decision procedure based on Gröbner basis computations and the real Nullstellensatz share the presence of checkable witnesses with approaches based on the Positivstellensatz. Once a witness $1 + \sum_i s_i^2 = 0$ has been found, the polynomial equality check can be performed easily within a proof system using the GK rules, giving a fully formal proof. The performance in our experiments show that this new approach is promising. It outperforms most other approaches, except for highly tuned QE procedures, which lack support for formal traceability. We believe that further research in this area is likely to produce competitive but traceable solutions for real arithmetic.

References

1. Tarski, A.: A Decision Method for Elementary Algebra and Geometry, 2nd edn. University of California Press, Berkeley (1951)
2. Ratschan, S.: Efficient solving of quantified inequality constraints over the real numbers. ACM Trans. Comput. Log. 7, 723–748 (2006)
3. Collins, G.E., Hong, H.: Partial cylindrical algebraic decomposition for quantifier elimination. J. Symb. Comput. 12, 299–328 (1991)
4. Weispfenning, V.: Quantifier elimination for real algebra - the quadratic case and beyond. Appl. Algebra Eng. Commun. Comput. 8, 85–101 (1997)
5. Buchberger, B.: An Algorithm for Finding the Basis Elements of the Residue Class Ring of a Zero Dimensional Polynomial Ideal. PhD thesis, University of Innsbruck (1965)
6. Parrilo, P.A.: Semidefinite programming relaxations for semialgebraic problems. Math. Program. 96, 293–320 (2003)
7. Stengle, G.: A Nullstellensatz and a Positivstellensatz in semialgebraic geometry. Math. Ann. 207, 87–97 (1973)
8. Platzer, A., Quesel, J.D.: KeYmaera: A hybrid theorem prover for hybrid systems. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS, vol. 5195, pp. 171–178. Springer, Heidelberg (2008)
9. Brown, C.W.: QEPCAD B: A program for computing with semi-algebraic sets using CADs. SIGSAM Bull. 37, 97–108 (2003)

10. Dolzmann, A., Sturm, T.: Redlog: Computer algebra meets computer logic. ACM SIGSAM Bull. 31, 2–9 (1997)
11. McLaughlin, S., Harrison, J.: A proof-producing decision procedure for real arithmetic. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS, vol. 3632, pp. 295–314. Springer, Heidelberg (2005)
12. Borchers, B.: CSDP, a C library for semidefinite programming. Optimization Methods and Software 11, 613–623 (1999)
13. Harrison, J.: Verifying nonlinear real formulas via sums of squares. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 102–118. Springer, Heidelberg (2007)
14. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reasoning 41, 143–189 (2008)
15. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. LNCS, vol. 4334. Springer, Heidelberg (2007)
16. Platzer, A., Quesel, J.D., Rümmer, P.: Real world verification. Reports of SFB/TR 14 AVACS 52, SFB/TR 14 AVACS (2009) ISSN: 1860-9821, `http://www.avacs.org`
17. Rümmer, P.: A sequent calculus for integer arithmetic with counterexample generation. In: Beckert, B. (ed.) VERIFY 2007 at CADE, Bremen, Germany. CEUR-WS.org, vol. 259 (2007)
18. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, Chichester (1986)
19. Platzer, A.: Combining deduction and algebraic constraints for hybrid system analysis. In: Beckert, B. (ed.) VERIFY 2007 at CADE, Bremen, Germany. CEUR Workshop Proceedings, vol. 259, pp. 164–178. CEUR-WS.org (2007)
20. Davenport, J.H., Heintz, J.: Real quantifier elimination is doubly exponential. J. Symb. Comput. 5, 29–35 (1988)
21. Strzebonski, A.W.: Cylindrical algebraic decomposition using validated numerics. J. Symb. Comput. 41, 1021–1038 (2006)
22. Bochnak, J., Coste, M., Roy, M.F.: Real Algebraic Geometry. Ergebnisse der Mathematik und ihrer Grenzgebiete, vol. 36. Springer, Heidelberg (1998)
23. Boyd, S., Vandenberghe, L.: Convex Optimization. Cambridge Univ. Press, Cambridge (2004)
24. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete Mathematics: A Foundation for Computer Science. Addison-Wesley Longman, Amsterdam (1994)
25. Platzer, A., Quesel, J.D.: Logical verification and systematic parametric analysis in train control. In: Egerstedt, M., Mishra, B. (eds.) HSCC 2008. LNCS, vol. 4981, pp. 646–649. Springer, Heidelberg (2008)
26. Kovács, L.: Aligator: A mathematica package for invariant generation (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS, vol. 5195, pp. 275–282. Springer, Heidelberg (2008)
27. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
28. Dolzmann, A., Sturm, T., Weispfenning, V.: A new approach for automatic theorem proving in real geometry. J. Autom. Reason. 21, 357–380 (1998)
29. Nipkow, T.: Linear quantifier elimination. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS, vol. 5195, pp. 18–33. Springer, Heidelberg (2008)
30. Tiwari, A.: An algebraic approach for the unsatisfiability of nonlinear constraints. In: Ong, C.H.L. (ed.) CSL 2005. LNCS, vol. 3634, pp. 248–262. Springer, Heidelberg (2005)

31. Akbarpour, B., Paulson, L.C.: Extending a resolution prover for inequalities on elementary functions. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS, vol. 4790, pp. 47–61. Springer, Heidelberg (2007)
32. Warren, A., Hunt, J., Krug, R.B., Moore, J.S.: Linear and nonlinear arithmetic in ACL2. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 319–333. Springer, Heidelberg (2003)
33. Platzer, A., Clarke, E.M.: Computing differential invariants of hybrid systems as fixedpoints. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 176–189. Springer, Heidelberg (2008)

Author Index