Fine-grained Concurrent Completion

Claude Kirchner Christopher Lynch Christelle Scharff

INRIA Lorraine & CRIN,
615, rue du Jardin Botanique, BP 101,
54602 Villers-lès-Nancy Cedex, France.
E-mail: {Claude.Kirchner, Christopher.Lynch, Christelle.Scharff}@loria.fr
http://www.loria.fr/equipe/protheo.html

Abstract. We present a concurrent Completion procedure based on the use of a SOUR graph as data structure. The procedure has the following characteristics. It is asynchronous, there is no need for a global memory or global control, equations are stored in a SOUR graph with maximal structure sharing, and each vertex is a process, representing a term. Therefore, the parallelism is at the term level. Each edge is a communication link, representing a (subterm, ordering, unification or rewrite) relation between terms. Completion is performed on the graph as local graph transformations by cooperation between processes. We show that this concurrent Completion procedure is sound and complete with respect to the sequential one, provided that the information is locally time stamped in order to detect out of date information.

1 Introduction

Parallelization is an attractive way for improving efficiency of automated deduction, and the main approaches are surveyed in [BH94] and [SS93]. We present in this paper a new approach to term rewriting completion which is based on fine grain concurrency, and which relies on a novel approach to completion.

The resolution (and paramodulation) inference systems are theorem proving procedures for first-order logic (with equality) that can run exponentially long for subclasses which have polynomial time decision procedures, as in the case of the Knuth-Bendix ground completion procedure. Wayne Snyder proved that completion of ground equations can be done in $nlog(n)$ [Sny93]. Recently C. Lynch has shown [Lyn95] that a special form of paramodulation which does not need to copy terms or literals runs in polynomial time in ground cases that include ground completion. This can be implemented in an elegant way using the notion of SOUR graph [LS95]. These graphs represent in a very convenient way a state of the completion, where all the basic ingredients (i.e. orientation, unification and rewriting) are made explicit at the object level. This makes explicit the fundamental operations of completion. A SOUR graph has its edges labelled by S when representing a subterm relation, by O when representing an orientation, by U when representing a unification problem and by R when representing a rewrite rule. The nodes of the graph are labelled by function symbols, and edges are labelled by constraints and renamings.

The properties of SOUR graphs are useful to study the parallelization and implementation of automated deduction on parallel distributed memory machines. There is no duplication of work since there is no copying. There is also no need of a consistency check. The explicit representation of basic operations allows a direct implementation of the inference rules as transition rules and thus to get completion as the result of independent (and asynchronous) operations as well as an easier way to describe and prove soundness and completeness of the graph transformations. This property is always desirable, but the increased complexity of concurrent processes makes it even more important.

Thus this paper presents a new fine-grained concurrent completion procedure based on the notion of distributed SOUR graphs and it is proved to be sound and complete, for simplicity in the case of ground completion, although everything can be extended to non-ground completion.

We consider each node of the SOUR graph as a process and the edges as communication links between processes. Each process is in charge of detecting particular configurations corresponding to critical pairs, unification problems or ordering of terms. A node acts only in response to a message, independently of the other nodes. The processes are thus completely asynchronous. When a successful configuration is found, the corresponding operation is performed, typically directing a subterm or a rewrite edge from one node to another. In the first design of the approach we were thought that this was enough to ensure correctness of the process. But after investigations, we discovered that since all these detections and actions are performed asynchronously, we need to keep account of only the newest information arriving from a given node, in order to ensure the global consistency of the SOUR graph. This is performed via the use of local time stamps, so that the system still works asynchronously, but old information is ignored.

The paper describes the principles of this approach and its current implementation on a network of processors. We show that this implements ground completion using fine-grain concurrency. This relies in particular on an original parallelization of the detection of the satisfiability of unification and LPO orientation constraints, detection that runs concurrently with the standard completion process. Because it is fine-grained and completely asynchronous, our approach is quite different from the PaReDuX work [BGK95] and the clause diffusion method [BH95] but it also allows backward contraction. It is also quite different from the discount approach [ADF95] or the work on Partheo [LS90]. Let us finally emphasise that we think our approach quite promising since as opposed to the unsuccessful attempts to parallelize prolog on fine-grained architectures, it is backed by a very simple concept, SOUR graphs, no synchronization is needed, and each process gets enough work because of the internalization of unification and ordering constraint satisfiability.

The paper is structured as follows. Assuming the reader familiar with term rewriting (see [DJ90]), we summarize in section 2 the notion of SOUR graph. In section 3, we present the principle of concurrent completion on SOUR graphs. We show how critical pairs are detected, how unification and LPO ordering constraints are checked satisfiable, why time stamps are needed and how they are used. We also show how to detect the termination of the completion process and we prove that this model of completion is sound and complete.

The full version of this paper that includes full proofs and a complete description of the transitions at the node level is available in [KLS96].

2 Preliminaries

To simplify our presentation of inference rules, we will present them for ground terms. We refer the reader to [LS95] to see how the definitions and the inference rules are lifted to non-ground terms.

The symbol $\approx$ is a binary symbol, written in infix notation, representing semantic equality. Let EQ be a set of equations. We define a function Sub so that $Sub(EQ)$ is the set of subterms of EQ. If $t = f(t_1, \cdots, t_k)$ with $k \geq 0$, then $Sub(t) = \{t\} \cup \bigcup_{1 \leq i \leq k} Sub(t_i)$. We define $Sub(s \approx t) = Sub(s) \cup Sub(t)$ and $Sub(EQ) = \bigcup_{eq \in EQ} Sub(eq)$.

In this paper $\preceq$ will refer to the *lexicographic path ordering* ($\prec$ in its strict version). If $u[s]$ is a ground term, and EQ is a set of ground equations, we write $u[s] \Rightarrow u[t]$ and say that $u[s]$ *rewrites in one step* to $u[t]$ if there is an equation $s \approx t \in EQ$ such that $s \succ t$. We write $u_0 \stackrel{*}{\Rightarrow} u_n$ and say that u_0 *rewrites* to u_n if there is a a set of terms $\{u_1, \cdots, u_{n-1}\}$ such that for all i, $1 \leq i \leq n$, $u_{i-1} \Rightarrow u_i$. A ground set of equations is *convergent* if it is terminating and confluent.

A *SOUR* graph is a compact dag representation of a set of equations. Let EQ be a set of equations. The SOUR graph of EQ is the graph which has one node associated with each element of $t \in Sub(EQ)$, labelled with the root symbol of t. For each element $f(t_1 \cdots, t_k) \in Sub(EQ)$, with $k > 0$, for each i, $1 \leq i \leq k$, there is a directed edge called a *subterm edge* in the SOUR Graph from the node associated with $f(t_1, \cdots, t_k)$ to the node associated with t_i labelled with i, its index. For each equation $s \approx t \in EQ$ with $s \succ t$, there is a directed edge called a *rewrite edge* from the node associated with s to the node associated with t.

We define a semantic function $Term$ from the nodes of the graph to the set of terms. If v is a node in the graph labelled with f such that $arity(f) = k$, then there are k subterms edges from v labelled $1, \cdots, k$ to terms $v_1 \cdots, v_k$ respectively. Then $Term(v) = f(Term(v_1), \cdots, Term(v_k))$. We define a function $Rule$ from the rewrite edges in the graph to the set of equations. If e is a rewrite edge from v_1 to v_2, then $Rule(e)$ is $Term(v_1) \approx Term(v_2)$. Graph G represents $\{Rule(e) \mid e \text{ is a rewrite edge in } G\}$.

We also add other kinds of edges to the SOUR graph. Some terms have an undirected edge called a *unification edge* between them, including between a node and itself. For now, we assume these edges are placed between every pair of nodes v_1 and v_2 such that $Term(v_1) = Term(v_2)$. There is a directed edge called an *orientation edge* between some pairs of nodes v_1 and v_2 such that $Term(v_1) \succ Term(v_2)$. These edges are used to perform inferences. The graph is called a *SOUR graph* because of **S**ubterm, **O**rientation, **U**nification and **R**ewrite edges.

The inference rule that we are coding with SOUR Graphs in the ground case is the *critical pair* inference rule.

$$\text{Critical Pair} \qquad \frac{s \approx t \qquad u[s] \approx v}{u[t] \approx v} \quad \text{if } s \succ t \textit{ and } u[s] \succ v$$

We simulate the completion inference rules by searching for patterns (or configurations) in the graph and performing a transformation of the graph whenever we find one. A transformation consists of removing a subterm or rewrite edge and adding a new subterm or rewrite edge. Afterwards, unification and orientation edges are re-calculated. There are three graph transformations.

The first transformation is called an *SUR transformation*. It consists of finding a set of edges v_1, v_2, v_3 and v_4 such that there is a subterm edge e_S from v_1 to v_2, a unification edge e_U between v_2 and v_3, and a rewrite edge v_R from v_3 to v_4. Then e_S is removed, and a new subterm edge e is added from v_1 to v_4, labelled with the same index as e_S (see Figure 1). This simulates the critical pair rule, because it unifies a subterm of a term with the larger side of an equation and replaces it with the smaller side of the equation. Figure 1 shows a critical pair between $f(a) \approx c$ and $a \approx b$.

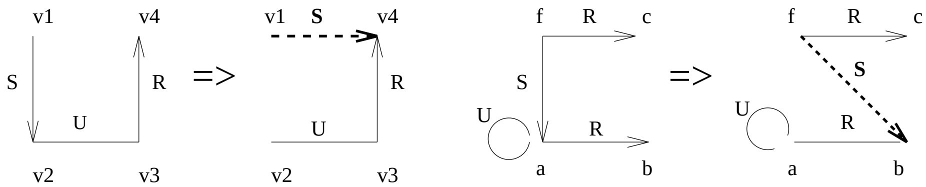

Fig. 1. *SUR* transformation and an example

The second transformation is called an *RUR transformation*. It consists of finding a set of edges v_1, v_2, v_3 and v_4, such that there is a rewrite edge e_{R_1} from v_1 to v_2, a unification edge e_U between v_1 and v_3, and a rewrite edge v_{R_2} from v_3 to v_4. Also $Term(v_2) \succ Term(v_4)$. Then e_{R_1} is removed, and a new rewrite edge e is added from v_2 to v_4 (see Figure 2). This simulates the critical pair rule, because it unifies the larger side of two equations and adds a new equation between the two smaller sides.

The third transformation is called an $RUR - rhs$ *transformation*. It consists of finding a set of edges v_1, v_2, v_3 and v_4, such that there is a rewrite edge e_{R_1} from v_1 to v_2, a unification edge e_U between v_2 and v_3, and a rewrite edge v_{R_2} from v_3 to v_4. Then e_{R_1} is removed, and a new rewrite edge e is added from v_1 to v_4 (see Figure 2). This simulates a simplification of the smaller side of an equation, because is unifies the smaller side with the larger side of another equation and replaces it with the smaller side of the other equation.

The completeness result from [LS95] says that if a SOUR graph is created from a set of equations EQ and the above transformations are performed in

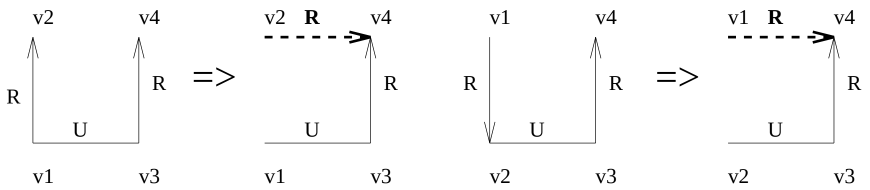

Fig. 2. *RUR* and *RUR-rhs* transformation

any order, until no longer possible, then the resulting SOUR graph represents a convergent system, logically equivalent to EQ.

We describe briefly how this is lifted to the non-ground case, and refer the reader to [LS95] for more details. For the non-ground case, the initial SOUR graph is constructed in exactly the same way. However, the transformations are handled differently. We simulate Basic Completion. Therefore, each inference is performed by renaming the variables in one of the premises, and by applying a constraint to the conclusion of the inference, representing the unification problem. In the SOUR graph, the new edge that is added to the graph in a transformation is labelled with the constraint and renaming associated with the inference, which is a combination of the constraints and renamings labelling the edges which caused the transformation to be performed. Initial edges can be considered to have constraints and renamings which are trivial. As opposed to the ground case, each inference does not represent a simplification. It is only possible to delete an edge when the inference represents a simplification.

Since edges are not deleted when new ones are added, a node represents a set of terms, instead of a single term. The semantics of SOUR graphs must also be modified to accommodate the constraints and renamings on the edges. Each dag of subterm edges still represents a term. But the constraints on the dag must be conjoined with each other, and the renamings on the edges must be applied to everything that appears underneath it in the term. Also, instead of performing transformations among edges with a unification constraint of true, we now perform transformations where the unifications constraints must be satisfiable, and the constraint representing the orientation is satisfiable. These constraints are passed along to to the new edge in a transformation, along with the constraints on the old edges.

In summary, the algorithm for completion of SOUR graphs is the same in the non-ground case as in the ground case, except for the fact that an old edge might not be deleted in a transformation, and the new edge is labelled by a renaming inherited from the old edges, and a constraint inherited from the old edges and the unification constraint determined by the unification problem. For simplicity, we present the completion procedure in the ground case, but the algorithm for the non-ground case is the same, except that messages will contain constraints and nodes will try to satisfy these constraints.

3 Concurrent Completion using SOUR Graphs

We now present our concurrent method for performing completion using SOUR graphs. At the beginning of the concurrent completion process, the *root process* has knowledge of the whole SOUR graph. For each node, it launches a process. To each process is associated its identification number called its *tid*. The preliminary information that is initially loaded by each process, is the symbol of the node and a dictionary containing for each symbol its arity and its place in the precedence. Then, for each node, unification edges, subterm edges, and rewrite edges are sent in this order to the associated process using messages called *INITU*, *INITS* and *INITR* respectively. The root process creates a unification edge between every pair of nodes with the same symbol, with an initial value of false. One node adjacent to each unification edge is designated as being the node in charge of calculating the unification constraint. An *INITU* message is sent to the nodes on both ends of each unification edge. The INITU message contains the *tid* of the process at the other end of the edge, the initial false unification constraint, and a notification whether that node is in charge of calculating the unification problem. *INITS* is sent to each node adjacent to a subterm edge. It contains the *tid* and the symbol of the process at the other end of the edge, a boolean indicating if the edge is incoming or outgoing, and the index of the subterm edge. *INITR* is sent to each node adjacent to a rewrite edge. It contains the *tid*, the symbol of the process at the other end of the edge and a boolean indicating if the rewrite edge is incoming or outgoing. We call this phase the *Initialization phase*.

The set of information stored in a node is called its *state*. It is composed of its symbol *symb*, its dictionary *dico_order_arity* and its unification edges, its outgoing and incoming subterm edges and its outgoing and incoming rewrite edges, which are saved in data structures : *U_list*, *S_out_list*, *S_in_list*, *R_out_list*, *R_in_list*. Initially, the U_list of a process contains the unary cycle between the node and itself, which forms a unification edge with a true unification constraint.

Completion, including the following different phases: configuration creation, configuration processing and Unification and Orientation computation, concerns all but the *root process*. A process works only by *reaction to messages*, which allows the whole process to be fully asynchronous.

Actions of a process implied by a received message are formalized using the *transition rules*. An α-*transition* describes the transformation of a *state* of a process, when it receives a message. It is denoted by: $(\{Mesg\}, State) \xrightarrow{\alpha} (Mesgset, State')$, where α is a phase of completion, $Mesg$ is the received message, $State$ is the state or a part of the state of the process before receiving the message, $Mesgset$ is the set of all messages that must be sent as computed consequences of the message $Mesg$, and $State'$ is the new *state* of the process after processing the message $Mesg$. In the set $Mesgset$, each message $mesg$ is characterized by its destination $dest$, and is denoted $mesg[dest]$. We have *D-transitions* for the creation of configurations, *P-transitions* for the processing of configurations, *U-transitions* and *O-transitions* for the calculation of Unification and Orientation respectively. We now give more details on these transitions.

3.1 Implementation of Inference Rules for Completion

Completion, i.e. local graph transformation, is performed by cooperation between processes by message passing, because each process has a local view of the graph.

Creation of configurations: For the creation of a configuration, we use the fact that a process can detect a sequence of two adjacent edges forming a configuration: SUR , *RUR-rhs*, RUR with a unification edge between a node and itself, or two adjacent edges forming a semi-configuration UR, (a unification edge and an outgoing rewrite edge). These patterns of two adjacent edges are detected when a subterm or rewrite edge is added or when the unification constraint of a unification edge becomes true.

When a semi-configuration UR is detected, a message called *SEMICONF* is sent to the process at the other end of the unification edge. When a configuration of type SUR, *RUR-rhs* or RUR is detected, a message called *CONFIG* containing this configuration is sent to only one process, since we want to avoid redundant work. *SUR* and *RUR-rhs* configurations are sent to the process at the top left with respect to figures 1 and 2. An *RUR* configuration is sent to the process of the outgoing end of the rewrite edge with the maximal *tid*.

Processing of configurations: When a *CONFIG* message is received, it releases a sequence of actions depending on the type of the received configuration. The process receiving the message processes the configuration. i.e. performs the associated transformation, and sends messages which will be used to update information of other processes. The two main steps of the processing of a configuration are the addition and the deletion of an edge. Both these actions can be expressed using *P-transitions*. However, the processing of configurations of type *RUR-rhs* or *RUR* is linked to the calculation of Orientation, which is calculated by *O-transitions* (see section 3.2).

Messages used for the processing of configurations are *INITS*, *INITR*, *UPDATES*, *UPDATER* and *UPDATESEMICONF*. These three last messages are used to delete respectively a subterm edge, a rewrite edge or a semi-configuration.

For example, the following *P-transition* processes an SUR configuration:

$$(\{CONFIG\},\ \{St.S\_out\_list\})$$
$$\xrightarrow{P}$$
$$(\{INITS(e')[tid1],\ UPDATES(e)[tid2]\},\ \{St.S\_out\_list + e' - e\})$$

where St is the *state* of the process receiving *CONFIG*, e is the subterm edge contained in the SUR configuration, $tid2$ is the *tid* at the outgoing end of e, e' is the new outgoing subterm edge and $tid1$ is the *tid* of the process at the outgoing end of the rewrite edge contained in the SUR configuration.

The transformations for the full completion process are described in the full version of the paper.

3.2 Concurrent Unification and Orientation

Although concurrency cannot improve the worst-case behavior of unification [DKM84], we give an algorithm that reduces the amount of processing by a single node. Unification is evaluated by a request-answer method A process sends a request to each child to ask if there is a unification edge with a true constraint between itself and another process. The child responds only if the constraint is true.

Consider the problem represented by figure 3. U_0 is the unification edge between the process representing term $f(s_1, \ldots, s_n)$ and the process representing term $f(t_1, \ldots, t_n)$. Computation of the unification constraint c_0 of unification edge U_0 depends on the calculation of unification constraints c_1,. . .,c_n of unification edges U_1,. . .,U_n. We have $c_0 = c_1 \wedge \ldots \wedge c_n$. Each constraint c_i represents the unification problem $s_i \stackrel{?}{=} t_i$.

Unification must be re-calculated when subterm edges are added or deleted.

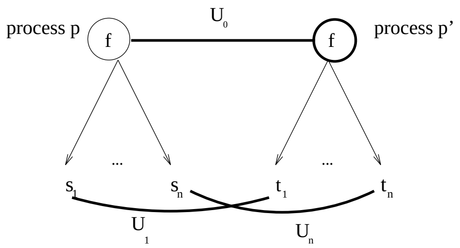

Fig. 3. Calculation of the unification constraint c_0 of the unification edge U_0

Using figure 3, we will explain the principles of the calculation of the unification constraint c_0 concurrently:

— Suppose process p represents $f(s_1, \ldots, s_n)$ and p' represents $f(t_1, \ldots, t_n)$, such that p' in charge of computing c_0. Process p sends to p' the *tids* of processes representing terms $s_1, \ldots, s_n$, so that p' knows on which children's unification constraints c_0 depends.

— When p' receives this information, it asks the processes representing terms t_1, . . ., t_n, if their unification constraints c_1, . . ., c_n respectively are true. This calculation is top-down.

— If a unification constraint c_i $(i > 0)$ becomes true and has a saved request concerning the computation of c_0, then the process representing term t_i informs process p' that c_i became true. This calculation is bottom-up.

— If process p' detects that the unification constraint c_0 becomes true i.e. the number of answer messages received for the calculation of c_0 is equal to the arity of the symbol of p', then it must tell the node at the other end of the unification edge that the unification constraint became true.

Soundness and completeness results for this concurrent Unification algorithm are given in section 3.5.

An Orientation problem is to determine if a term s is bigger than a term t with respect to a given LPO. If it is, we create an outgoing orientation edge from the process representing term s to the process representing term t. We use a request-answer method as we did for Unification.

3.3 Detection of the termination of the program

It is difficult to detect termination of an asynchronous concurrent program, because a process cannot detect that it will not receive any more messages. We give a termination detection algorithm different from the classical one [DFvG83].

Definition 1. The program terminates when all processes are *idle* and all sent messages have been received.

Our termination detection algorithm uses a mailbox, which can be handled by the *root process*. It is based on the following principles.

Sent and received messages are notified to the mailbox using a message called *NOTIFY*. In practice, a message can be specified in the mailbox as received and not yet as sent. The mailbox contains envelopes of sent or received messages. An envelope is composed of a flag *SENT* or *RECEIVED* for a sent or received message $mesg$, the source process of $mesg$, the destination process of $mesg$ and an identification number, permitting us to distinguish two messages having the same origin, the same destination and the same contents. So, an envelope is one of the following forms: $(p_i, p_j, NB, SENT)$ for message NB sent by process p_i to p_j, and $(p_j, p_i, NB, RECEIVED)$ for message NB received by process p_i from p_j. A mailbox consists of two lists: *list_sent*, which contains envelopes of messages specified as sent but not received and *list_received*, which contains envelopes of messages specified as received but not sent.

Processes cycle from idle to busy to idle. The busy state is entered when a message is received. Other messages are sent as a result of this message. In the busy state, other messages may be received and messages are sent as a result of them. When the process re-enters the idle state, it sends the mailbox one NOTIFY message for each message received in that cycle. The NOTIFY message contains one envelope for the received message, and an envelope for each message sent as a result of this received message. This fact is crucial for the correctness of our termination detection algorithm.

When the mailbox receives a message *NOTIFY* containing an envelope $(p_i, p_j, NB, SENT)$, the mate of this envelope $(p_j, p_i, NB, RECEIVED)$ is searched for in the list *list_received*. If the envelope $(p_j, p_i, NB, RECEIVED)$ does not belong to this list, $(p_i, p_j, NB, SENT)$ is added to the list *list_sent*, otherwise $(p_j, p_i, NB, RECEIVED)$ is deleted from *list_received*. RECEIVED messages are processed similarly.

Initially, *list_sent* contains all the envelopes of SENT messages of the *Initialization phase*. This ensures the correctness of the algorithm.

Theorem 2. *The program is terminated iff* list_sent *and* list_received *are empty.*

3.4 Time Stamps

Why are time stamps needed? Let $E = \{g(f(a)) \approx b, f(a) \approx f(b), a \approx c\}$. be the set of equations to complete. Consider the precedence $g \prec f \prec c \prec b \prec a$. The set of equations E can be represented by the *SOUR graph* of figure 4.

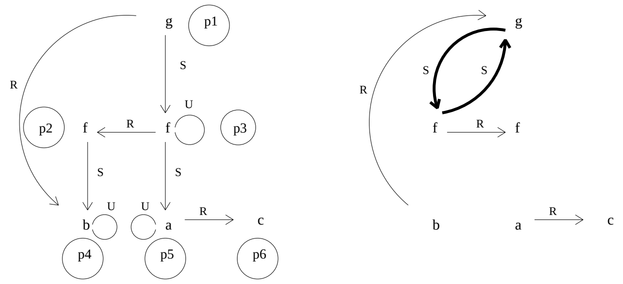

Fig. 4. The SOUR Graph representing E and the result of the execution plan

Consider the following execution plan. Process p_3 detects an SUR configuration $conf1$ and sends it to process p_1. Process p_5 detects an SUR configuration $conf2$ and sends it to process p_3. Process p_3 receives the SUR configuration $conf2$ and processes it. Process p_3 changes the direction of its rewrite edge because $f(b) \succ f(c)$. Process p_1 processes a message from p_3 to tell it to change the direction of its rewrite edge because $b \succ g(f(c))$. Process p_4 detects an SUR configuration $conf3$ and sends it to p_2. Process p_1 receives and processes the SUR configuration $conf1$. Process p_2 receives and processes the SUR configuration $conf3$. On the graph, we notice a cycle of subterm edges. To prevent this critical case, we use *Time Stamps*. They are not used for synchronization, but to avoid using old information.

Time stamp definition and usage

Definition 3. The time stamp of a process is a counter initialized to 0. We denote the current time stamp of a process p by TS_p.

Each process p contains a time stamp TS_p and a table of time stamps that we denote TS_Table_p. The table has n entries, where n is the arity of the symbol of p. For process p', $TS\_Table_p[p']$ is the last time stamp $TS_{p'}$ of process p' known by p, initially 0. When p sends a message to one of its parent processes, it sends its current time stamp TS_p in the message.

A unification edge between processes p and p' has two time stamps associated with it: the time stamps of the processes at the ends of the edge. These time stamps are stored in p and p'. The same thing is done for orientation edges.

Use of time stamps for the creation of configurations: Consider a semi-configuration containing a rewrite edge r from process p_3 to process p_4 and a unification edge between processes p_2 and p_3. This semi configuration is sent to p_2 only if the calculation of the direction of the rewrite edge r gives that r is from p_3 to p_4 and the calculation of Unification for the unification edge u between p_2 and p_3 gives a true unification constraint and the time stamp of p_3 used for calculating Unification is the same as the time stamp of p_3 used for calculating Orientation. The message *SEMICONF* is sent to p_2 containing the time stamp of p_2 used to calculate Unification. Let t be this time stamp. When p_2 receives the message *SEMICONF*, it sends to process p_1 a message *CONFIG* only if $TS_{p_2} = t$. If process p_1 receives a message *CONFIG* containing an SUR configuration, it processes it only if the subterm edge in the message still exists and $TS\_Table_{p_1}[p_2] = t$ or $TS\_Table_{p_1}[p_2] < t$. Then, if $TS\_Table_{p_1}[p_2] < t$, TS_{p_1} is incremented and $TS\_Table_{p_1}$ is updated so that $TS\_Table_{p_1}[p_2] = t$.

When process p_1 receives a message *CONFIG* containing an RUR or an *RUR-rhs* configuration, it processes it only if the incoming or the outgoing rewrite edge respectively still exists.

Use of time stamps for Unification (It is similar for Orientation): Suppose that we want to calculate the Unification constraint of unification edge u between process p_0 and process p_1, such that p_1 decides the calculation.

Suppose that process p_2 (k^{th} child process of p_1) receives a request asking if there is a unification edge with constraint true between processes p_2 and p_3 (k^{th} child process of p_0). Answers will be sent to p_1 (for conditions of this sending see section 3.2). Answer messages contain t_2 and t_3 the last time stamps of p_2 and p_3 used in the calculation of the unification constraint between p_2 and p_3.

When p_1 receives an answer from p_2 containing time stamps t_2 and t_3, the answer message is processed only if the following conditions are respected:

— There is a subterm edge going from p_1 to p_2 with index k.

— $TS\_Table_{p_1}[p_2] = t_2$ or $TS\_Table_{p_1}[p_2] < t_2$. For this second case, $TS\_Table_{p_1}$ is updated, such that $TS\_Table_{p_1}[p_2] = t_2$ and the current time stamp of p_1 is incremented.

If an answer message permits us to determine that the unification constraint of the unification edge u between p_0 and p_1 is true i.e. n answer messages have been received by p_1 where n is the arity of the symbol of p_1, then the message *INITU* corresponding to the fact that the unification constraint of u is true, is sent to p_0 containing t_3.

When p_0 receives from p_1 a message *INITU* containing the time stamp t_3, this message is processed only if the following conditions are respected:

— There is a subterm edge going from p_0 to p_3 with index k.

— $TS\_Table_{p_0}[p_3] = t_3$ or $TS\_Table_{p_0}[p_3] < t_3$. For this second case, $TS\_Table_{p_0}$ is updated, such that $TS\_Table_{p_0}[p_3] = t_3$ and the current time stamp of p_0 is incremented.

Semantics of a time stamped ground SOUR Graph

Definition 4. We define the semantics of the term representing process p by $T(p, TS_p)$, where TS_p is the current time stamp of process p. We have:

$T(p, TS_p) = f(T(p_1, t_1), \ldots, T(p_n, t_n))$ where:
— f is the symbol of process p.
— p_i are processes such that there exists a subterm edge going out from p to p_i with index i, when the time stamp of p is TS_p.
— t_i is the last time stamp of p_i known by p, $\forall i \in \{1, \ldots, n\}, TS\_Table_p[p_i] = t_i$.

Using time stamps, we have the following results.

Lemma 5. *For all time stamps t and t' such that $t' < t$, and for all processes p, we have: $T(p, t') \succeq T(p, t)$.*

Definition 6. The term of a process is persisting, if the time stamp of the process is persisting.

Corollary 7. *For each process, its term becomes persisting.*

Theorem 8. *Concurrent ground completion using SOUR graphs terminates.*

3.5 Soundness and Completeness Results

In this section, we give results for the soundness and completeness of our concurrent calculation of Unification and Orientation.

We call *CWD* the concurrent implementation of completion using SOUR Graphs and all the concurrent operations that we describe. We give results for the soundness and the completeness of *CWD*. Let E be the equational theory the graph represents.

Concurrent unification is *sound* if whenever message INITU is sent saying the unification constraint between terms s and t is true, then $E \models s \approx t$ is true and remains true.

Concurrent unification is *complete* if whenever s and t are persisting terms and s and t are unifiable, then at some time the unification edge between the processes representing terms s and t will have a true unification constraint. The concurrent computation of orientation is *complete* if whenever s and t are persisting terms and $s \succ_{lpo} t$ then, at some time, the orientation edge between processes representing terms s and t goes from s to t and does not change.

Theorem 9. *The concurrent computation of Unification is sound and complete, and the concurrent computation of Orientation is complete.*

CWD is *sound* if (i) after a rewrite edge is added between two processes with terms s and t, then $E \models s \approx t$ is true and remains true, and (ii) no subterm edge is added so that a cycle of subterm edges is created.

CWD is complete, if there are no configuration between persisting nodes.

Theorem 10. CWD *is sound and complete.*

4 Implementation and Experimental Results

Implementation : *CWD* is currently implemented in *C++* using *LEDA* (Library of Efficient Data types) and *PVM* (Parallel Virtual Machine). It is tested on a network of Unix and Solaris Sun4SparcStations, and Sgi workstations and on a Power Challenge Array (PCA) with 8 R8000 processors.

Experimental Results : Measurements have been made with many benchmark examples. *Table 1* shows a selection of results of measurements obtained on different platforms for two particular problems. *Table 1* summarizes, for each problem, the platform we use, the total number of messages sent between processes, the total number of *NOTIFY* messages (used for detecting termination) , and the real time of execution given in seconds.

Counter5 Example [GNP+93]: Precedence : $f \succ g \succ c$. Set of equations : $E = \{f(c) = g(c)\ ,\ f(g(c)) = g(f(c)),\ f(g(g(c)) = g(f(f(c))),\ f(g(g(g(c))) = g(f(f(f(c)))),\ f(g(g(g(g(c))))) = g(f(f(f(f(c)))))\}$

Number of child processes (the number of vertices of the initial graph representing the set of equations to complete) : 17

Expf6 Example: Precedence : $aa \succ ab \succ ac \succ ad \succ ae \succ af \succ ag \succ f \succ b$.

Set of equations : $E = \{aa = f(ab, ab, ab, ab, ab, ab), ab = f(ac, ac, ac, ac, ac, ac), ac = f(ad, ad, ad, ad, ad, ad), ad = f(ae, ae, ae, ae, ae, ae), ae = f(af, af, af, af, af, af), af = f(ag, ag, ag, ag, ag, ag),\ f(aa, aa, aa, aa, aa, aa) = b\}$.

On this example, *OTTER-3.0* runs out of memory after 20 minutes on a Digital server.

Number of child processes: 15

Problem	Platform	Completion Messages	Termination Messages	Real time
Counter5	1 Sun4	582	627	13.00 s
Counter5	5 Sun4	470	514	4.00 s
Counter5	10 Sun4	648	696	3.62 s
Counter5	15 Sun4	616	663	3.23 s
Counter5	5 Sgi	500	545	4.77 s
Counter5	PCA	594	639	1.88 s
Expf6	1 Sun4	724	822	11.00 s
Expf6	5 Sun4	737	835	5.52 s
Expf6	10 Sun4	734	832	3.00 s
Expf6	15 Sun4	729	827	3.00 s
Expf6	5 Sgi	715	813	4.56 s
Expf6	PCA	722	820	2.32 s

Table 1

5 Conclusion

We have presented a concurrent algorithm for completion using SOUR graphs. The initial set of equations is divided up into small pieces, stored in a graph. Initially each vertex represents one of the subterms in the initial set of equations.

Edges represent relations between those pieces. Inferences are local graph transformations, which remove existing edges and create new edges constrained by constraints and renamings. In the concurrent implementation, vertices represent processes and edges represent communication links between processes.

For simplicity, the algorithm we have given in this paper is for ground completion. However, we have summarized in the preliminaries how the theory of SOUR graphs is lifted to the non-ground case. The modification of our algorithm to the non-ground case is simple. Unification messages now contain a satisfiable equational constraint instead of just an assertion that a unification problem is true. Orientation messages contain a satisfiable ordering constraint. Configuration messages pass around these constraints, plus renamings. And new edges that are added are labelled with the combination constraint and renaming. This gives the processes more work to do. Whenever a node receives constraints, it must combine them, determine the satisfiability of the combination, and pass along the solved form.

Our presentation and algorithm has several desirable properties. As far as we know, it is the first fine-grained completion or theorem proving procedure, in the sense that each process represents a subterm of the equational system.

Another feature of most concurrent completion and theorem proving techniques is that they either require a global memory which all processes need to read and write to, or else a master process which controls all the other processes. In contrast, our algorithm works in a completely local fashion. All processes operate independently, because the algorithm is based on the local transformations of the SOUR graph algorithm. The work of a process is based on receiving a message from another process. However, a process does not need to wait, because there are lots of different things that are done by one process, and the message passing is asynchronous. Our algorithm has no need for any consistency checks, and two processes never do the same thing.

We feel that the property of having no global memory or global control is important. As mentioned in [BH94], contraction-based strategies are the most efficient strategies, but they are the most difficult to parallelize. Their parallelization almost always requires a global control or global memory, because anything can be a simplifier, and it is not known which simplifiers will simplify which equations. Our approach allows a contraction based strategy with all local operations.

Acknowledgements This work is partially supported by the Esprit Basic Research working group 6028, CCL. We would like to thank Polina Strogova, Ilies Alouini, Dominique Fortin and the anonymous referees for their comments.

References

[ADF95] J. Avenhaus, J. Denzinger, and M. Fuchs. DISCOUNT: A system for distributed equational deduction. In Hsiang [Hsi95], pages 397–402.

[BGK95] R. Bündgen, M. Göbel, and W. Küchlin. Parallel ReDuX → PaReDuX. In Hsiang [Hsi95], pages 408–413.

[BH94] M. P. Bonacina and J. Hsiang. Parallelization of deduction strategies: an analytical study. *Journal of Automated Reasoning*, 13:1–33, 1994.

[BH95] M. P. Bonacina and J. Hsiang. Distributed deduction by clause-diffusion: Distributed contraction and the aquarius prover. *Journal of Symbolic Computation*, 19:245–267, March 1995.

[DFvG83] E. Dijkstra, W. Feijen, and A. van Gasteren. Derivation of a termination detection algorithm for distributed computation. *Information Processing Letters*, 16:217–219, 1983.

[DJ90] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990. Also as: Research report 478, LRI.

[DKM84] C. Dwork, P. Kanellakis, and J. C. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1(1):35–50, 1984.

[GNP+93] J. Gallier, P. Narandran, D. Plaisted, S. Raatz, and W. Snyder. Finding canonical rewriting systems equivalent to a finite set of ground equations in polynomial time. *Journal of Association for Computing Machinery*, 40(1):1–16, 1993.

[Hsi95] J. Hsiang, editor. *Rewriting Techniques and Applications, 6th International Conference, RTA-95*, LNCS 914, Kaiserslautern, Germany, April 5–7, 1995. Springer-Verlag.

[KLS96] C. Kirchner, C. Lynch, and C. Scharff. Fine-grained concurrent completion. Technical Report 96-R-058, CRIN, 1996.

[LS90] R. Letz and J. Schumann. Partheo: A high-performance parallel theorem prover. In *Proceedings 10th International Conference on Automated Deduction, Kaiserslautern (Germany)*, volume 446 of *Lecture Notes in Artificial Intelligence*, pages 40–56. Springer-Verlag, 1990.

[LS95] C. Lynch and P. Strogova. Sour graphs for efficient completion. Technical Report 95-R-343, CRIN, 1995.

[Lyn95] C. Lynch. Paramodulation without duplication. In D. Kozen, editor, *Proceedings 10th IEEE Symposium on Logic in Computer Science, San Diego (Ca., USA)*, pages 167–177, San Diego, June 1995. IEEE.

[Sny93] W. Snyder. A fast algorithm for generating reduced ground rewriting systems from a set of ground equations. *Journal of Symbolic Computation*, 15:415–450, 1993.

[SS93] C. Suttner and J. Schumann. Parallel automated theorem proving. *Parallel Processing for Artificial Intelligence*, 1993.

This article was processed using the LaTeX macro package with LLNCS style.