Interpolants for Linear Arithmetic in SMT

Christopher Lynch and Yuefeng Tang

Department of Mathematics and Computer Science
Clarkson University
Potsdam, NY, 13676 USA
clynch@clarkson.edu, tangy@clarkson.edu

Abstract. The linear arithmetic solver in Yices was specifically designed for SMT provers, providing fast support for operations like adding and deleting constraints. We give a procedure for developing interpolants based on the results of the Yices arithmetic solver. For inequalities over real numbers, the interpolant is computed directly from the one contradictory equation and associated bounds. For integer inequalities, a formula is computed from the contradictory equation, the bounds, and the Gomory cuts. The formula is not exactly an interpolant because it may contain local variables. But local variables only arise from Gomory cuts, so there will not be many local variables, and the formula should thereby be useful for applications like predicate abstraction. For integer equalities, we designed a new procedure. It accepts equations and congruence equations, and returns an interpolant. We have implemented our method and give experimental results.

1 Introduction

In 1957 [5] Craig presented the idea of interpolant. But, only recently interpolant are used as an important part of program verification. Interpolants are especially useful for finding inductive invariants and for predicate abstraction [1,10,11,12]. Given two sets of logical formulas A and B where $A \cup B$ is unsatisfiable, a formula P is an interpolant of (A, B) if A implies P, P contradicts B , and P only contains variables common to A and B. Informally, an interpolant gives the reason why A and B contradict, in a language common to both of them. Invariants are created iteratively, and the common variable restriction keeps the number of variables from exploding.

In this paper, we are concerned with linear arithmetic (in)equalities, either over the reals or the integers. Our method, called YAI (Yices Arithmetic Interpolants), is based on the linear arithmetic algorithm of Yices[8]. This algorithm was specially designed to work inside an SMT prover. It is able to quickly add and remove single constraints. Yices' linear arithmetic solver reduces sets of inconsistent constraints into an unsatisfiable equation. Then, our method can simply construct an interpolant from that unsatisfiable equation. Therefore, the method is not proof-based, and after detecting unsatisfiability, it can find an interpolant without much additional work.

Cha et al. (Eds.): ATVA 2008, LNCS 5311, pp. 156–170, 2008.

Simplex methods are believed to be faster than Fourier Motzkin methods. Yices does not use the Simplex method, but it uses essential ideas from that method, with the idea that incremental changes must be fast. Given a set of constraints, Yices will first flatten the constraints using extension variables. What is left are equations that are pivoted and updated in a Simplex manner, and bounds on extension variables. The third component of the Yices algorithm is a set of variable assignments, which is updated as bounds are added. If the constraint set is unsatisfiable, then the algorithm will halt with an equation that conflicts with the bounds on the variables it contains. For real (in)equations, our method creates the interpolant from that equation and those bounds.

For integer (in)equations, the Yices algorithm needs to apply Gomory cuts, which introduce new variables. In this case, our formula uses the bounds, the contradicting equation, and the Gomory cut definitions to create a conditional formula which satisfies the first two conditions of the definition of interpolant. However, it is possible that the formula may contain variables that are not common to both A and B. Therefore, the formula is not exactly an interpolant. However, these offending variables are only from the Gomory cuts. So, there will not be many of them. When the purpose of requiring common variables is to keep the number of variables from exploding when the interpolant construction is iterated, we believe that these formulas are still useful in practice.

We have developed a special algorithm for interpolants over integer equations. This algorithm takes as input a set of equations and congruence equations. It is more of a Fourier Motzkin method, since new equations can be added, but in a controlled fashion. However, we still use the extensions and bounds from the Yices algorithm. This will retain the advantages of ease of adding and deleting equations. When the algorithm determines unsatisfiability, we will again have an equation which conflicts the bounds on the variables, and we will create a congruence equation from that information, which will be an interpolant.

We have implemented each of these methods. We created some random constraints and compared our implementation, YAI, with Yices. Our algorithm is the same as Yices except for integer equations, and the fact that we construct an interpolant. The time to construct the interpolant does not add significantly to the running time. However, due to clever coding techniques of the Yices programmers, YAI is slower than Yices. Our method could be incorporated into Yices without increasing its running time. For integer equations, our implementation is faster than Yices on the tested examples.

We are aware of two other implementations which create interpolants for linear arithmetic: FOCI[11] is which a proof-based method, and CLP-Prover[13] which is not proof-based. Those methods also handle uninterpreted function symbols and disjunctions. However, CLP-Prover does not handle integers, and FOCI only approximates them. Since FOCI is proof-based, it is more complicated to find an interpolant. The main difference between our implementation and those other two is that our implementation is designed to work in an SMT theorem prover which supports fast operations for adding and deleting constraints. Recently, we became aware of MATHSAT-ITP[3] and INT2[9]. For linear arithmetic

over reals, both MATHSAT-ITP and YAI are based on the simplex method. MATHSAT-ITP explicitly constructs a proof of unsatisfiability and uses McMillan's method[11] to build an interpolant from the proof of unsatisfiability. But, YAI does a similar thing by directly computing an interpolant from the unsatisfiable equation with its bounds. MATHSAT-ITP converts an equality into two inequalities, which creates more constraints. However, YAI can directly handle equalities. INT2 creates interpolants for equalities over integers, but it does not use the same method as YAI. INT2 is not designed to work in SMT.

YAI cannot handle uninterpreted function symbols and disjunctions. But, YAI can easily be extended to handle those problems by using the combination methods proposed by Yorsh and Musuvathi[14] or McMillan[11]. Notice that a disequality $t \neq 0$ can be rewritten into $t > 0 \vee t < 0$.

Our paper is organized as follows. In Section 2, we introduce the preliminaries. That is followed by a brief introduction to the linear arithmetic solver of Yices. Next, we introduce YAI for linear equalities and inequalities over rational numbers. In this section, the linear arithmetic solver is the same as Yices. After that, we introduce YAI for linear equalities over integers. In this section, a new complete linear arithmetic solver is introduced, which is different from Yices. Following that, we introduce YAI for linear equalities and inequalities over integers. The linear arithmetic solver of YAI is the same as Yices. Finally, we conclude and mention future work.

2 Preliminaries

An *atom* is a single variable. A *term* is defined as a constant, an atom or cx where c is a constant and x is a variable. If c is negative, we call cx a *negative term*, otherwise cx is a *positive term.* An expression is a summation of terms. $t_1 - t_2$ is an abbreviation of $t_1 + (-t_2)$ where t_1 and t_2 are terms.

In our paper, an equation $t_1 = t_2$ is called a *standard equation* (or simply equation) where t_1 and t_2 are expressions. $t_1 \equiv_m t_2$ is a *congruence equation* where t_1 and t_2 are expressions and m is a constant. An *inequality* is of the form $t_1 \phi t_2$, where $\phi \in \{>, <, \geq, \leq\}$. A *constraint* is a standard equation, an inequality, or a congruence equation. A congruence equation $t_1 \equiv_m t_2$ is *satisfiable* if there is an assignment for the variables such that t_1 and t_2 are congruent modulo m. Any other constraint is *satisfiable* if there is an assignment for the variables which satisfies the constraint. A set of constraints is *satisfiable* if there is an assignment for the variables which satisfies all the constraints. A set of constraints is *unsatisfiable* if it is not satisfiable.

We will consider two sets of constraints A and B, and we wish to determine if $A \cup B$ is satisfiable. A variable occurring only in A or only in B is called a *local variable.* Variables occurring in both A and B are called *global variables.* If $A \cup B$ is unsatisfiable, then P is an (A, B) interpolant if A implies P, $P \cup B$ is unsatisfiable, and all variables in P are global.

3 A Linear Arithmetic Solver for DPLL(T)

Yices is an SMT solver that can efficiently check satisfiability of linear arithmetic. If the problem is satisfiable, Yices generates an assignment for each variable; else unsatisfiability is detected. Yices first converts the formula into a conjunction of equalities and bounds. The new equalities and bounds can be derived by introducing *extension* variables. For instance, given an constraint $t\phi c$ where t is an expression, c is a constant and $\phi \in \{\geq, \leq, =, >, <\}$, a new equality $s = t$ and a *bound* $s\phi c$ are generated by introducing an *extension* variable s. t is called a *definition* of s, c is a *bound value* and ϕ is a *bound operator*. This process of replacing a constraint $t\phi c$ by an extension equation $s = t$ and a bound $s\phi c$ is called *flattening*. Later on, we will need to replace the extension variables back by their definitions, so we introduce a substitution σ to do that.

Definition 1. *σ is a substitution that replaces extension variables by their definitions.*

After flattening, a set of new equalities $\sum_{i=1}^{m} s_i = t_i$ forms a constraint matrix, and a set of bounds $\sum_{i=1}^{n} s_i \phi_i c_i$ is generated too. Notice that the conversion preserves satisfiability. Here is an example. Given two linear arithmetic sets: $A = \{x - y = 0, 2y \geq 1\}$, $B = \{x \leq 0\}$.

This will be flattened into extension equations $\{s_1 = x - y,\ s_2 = 2y\}$ and bounds $\{\ s_1 = 0,\ s_2 \geq 1,\ x \leq 0\ \}$ where s_1 and s_2 are extension variables. Notice that no extension variable is introduced for the bound $x \leq 0$ because x is an atom. So, we can directly treat $x \leq 0$ as a bound. After introducing extension variables, the new set derived from A is called A_e, and the new set derived from B is called B_e. It is not hard to see that $\sigma(A_e)$ is A and $\sigma(B_e)$ is B. In this example, A_e is $\{s_1 = x - y,\ s_2 = 2y,\ s_1 = 0,\ s_2 \geq 1\ \}$, and B_e is $\{x \leq 0\}$.

The equations in the constraint matrix always maintain the following form:

$$x_i = \sum_{x_j \in \mathcal{N}} a_{ij} x_j \qquad x_i \in \mathcal{B}$$

where x_i is a *basic variable*, which only appears on the left side of the equations, x_j are *non basic variables*, which appears only on the right side of the equations, $\mathcal{B}$ is the set of basic variables, $\mathcal{N}$ is the set of non-basic variables, and a_{ij} are the coefficients. Each variable x_i has two bounds l_i(lower bound) and u_i(upper bound). If no constant bound exists, we assume the bound is $-\infty$ or $+\infty$. If $l_i = -\infty$ and $u_i = +\infty$ then x_i is a *free variable*; else if $l_i = u_i$ then x_i is a *fixed variable*. We introduce a variable assignment β that maps each variable to a constant value. Initially $\beta(x) = 0$ for all variables x. The procedure continually updates β, so that all equations and bounds will be true.

The main algorithm of this solver contains two parts. The first part containing two asserting procedures in Figure 3 [8] is to assert bounds. The second part is to resolve bound violations using the main procedure *check*() in Figure 2[8]. Initially, for each variable x, $\beta(x) = 0$. Then, each bound will be asserted one by one. After asserting a bound, the variable assignment may be

procedure update(x_i, v)
  for each $x_j \in \mathcal{B}$, $\beta(x_j) := \beta(x_j) + a_{ji}(v - \beta(x_i))$
  $\beta(x_i) := v$

procedure pivotAndUpdate(x_i, x_j,v)
  $\theta := \frac{v-\beta(x_i)}{a_{ij}}$
  $\beta(x_i) := v$
  $\beta(x_j) := \beta(x_j) + \theta$
  for each $x_k \in \mathcal{B} \setminus x_i$, $\beta(x_k) := \beta(x_k) + a_{kj}\theta$
  pivot(x_i, x_j);

Fig. 1. Auxiliary procedures

1. **procedure** check()
2. **loop**
3. select the smallest basic variable x_i such that $\beta(x_i) < l_i$ or $\beta(x_i) > u_i$
4. **if** there is no such x_i **then** return *satisfiable*
5. **if** $\beta(x_i) < l_i$ **then**
6. select the smallest non-basic variable x_j such that
7. ($a_{ij} > 0$ and $\beta(x_j) < u_j$) or ($a_{ij} < 0$ and $\beta(x_j) > l_j$)
8. **if** there is no such x_j **then return** *unsatisfiable*
9. pivotAndUpdate(x_i, x_j, l_i)
10. **if** $\beta(x_i) > u_i$ **then**
11. select the smallest non-basic variable x_j such that
12. ($a_{ij} < 0$ and $\beta(x_j) < u_j$) or ($a_{ij} > 0$ and $\beta(x_j) > l_j$)
13. **if** there is no such x_j **then return** *unsatisfiable*
14. pivotAndUpdate(x_i, x_j, u_i)
15. **end loop**

Fig. 2. Check procedure

1. **procedure** AssertUpper($x_i \leq c_i$)
2. **if** $c_i \geq u_i$ **then return** *satisfiable*
3. **if** $c_i < l_i$ **then return** *unsatisfiable*
4. $u_i := c_i$
5. **if** x_i is non-basic variable and $\beta(x_i) > c_i$ **then** update(x_i, c_i)
6. **return** *ok*

1. **procedure** AssertLower($x_i \geq c_i$)
2. **if** $c_i \leq l_i$ **then return** *satisfiable*
3. **if** $c_i > u_i$ **then return** *unsatisfiable*
4. $l_i := c_i$
5. **if** x_i is non-basic variable and $\beta(x_i) < c_i$ **then** update(x_i, c_i)
6. **return** *ok*

Fig. 3. Assertion procedures

updated. Then, it is possible that the assignment of some variable could violate some asserted bound after updating the variable assignment. Thus, the procedure *check*() will be immediately called to resolve bound violations after the assignment is updated in the asserting procedures. If a bound violation is resolved by $pivotAndUpdate(x_i, x_j, v)$ or no violation is detected, a temporary assignment satisfies all equations in the constraint matrix and the asserted bounds. So, after all bounds are asserted, if the original problem is satisfiable then a model is generated. If the algorithm determines unsatisfiability because an equation e is unsatisfiable with the bounds of its variables, then we call e an *unsat* equation. The termination of the algorithm has been proved in [8].

To handle strict inequalities over reals, Yices converts strict inequalities into non-strict inequalities by introducing a variable δ representing an infinitely small positive real number [8]. For instance, $x > 2$ is converted into $x \geq 2 + \delta$.

The linear arithmetic solver of Yices is complete to decide satisfiability for linear arithmetic over real numbers. But, it is incomplete for linear arithmetic over integers. In the next section we will introduce a method to build an interpolant when the solver of Yices detects that the problem is unsatisfiable.

4 YAI for Linear Equalities and Inequalities over Reals

In this section, our method to check satisfiability of linear arithmetic is the same as Yices. Since any disequality $t \neq 0$ can be converted into $t > 0 \vee t < 0$ [8], YAI can be easily extended to handle disequalities. As mentioned in the previous section, the solver can detect unsatisfiability in the assert and check procedures. In the assert procedures, unsatisfiability means two asserted bounds contradict each other. For instance, the bound $x \geq 5$ contradicts the bound $x \leq 0$. So, in this case the way to construct an interpolant is simple, and is presented in Figure 4. The input b_i and b_j of this procedure represents two inconsistent bounds.

```
1. procedure bound_interpolant(b_i, b_j)
2.   if(both bounds are from A_e) then return FALSE;
3.   if(both bounds are from B_e) then return TRUE;
4.   if(b_i is from A_e) then return σ(b_i);
5.   if(b_j is from A_e) then return σ(b_j);
```

Fig. 4. Bound Interpolant procedure

If the algorithm returns unsatisfiable in the checking procedure, that means that a bound violation is detected in an equation and the solver is not able to resolve the bound violation. Thus, in this case an unsat equation is detected. Notice that in an unsat equation there are no free variables because if a free variable exists then the bound violation can always be resolved by calling $pivotAndUpdate(x_i, x_j, v)$. We can rewrite the unsat equation in the form $t = 0$. Since that equation contradicts the bounds, there must exist a minimal set of bounds K such that K implies $t > 0$ or K implies $t < 0$. Let ax be a term

Table 1.

ϕ	inverse(ϕ)
$\geq$	$\leq$
$>$	$<$
$\leq$	$\geq$
$<$	$>$
$=$	$=$

in t, and let $x\phi d \in K$. If ax is a positive term, then $ax\phi a * d$ is called an *active bound*. The function *inverse* maps an operator to an operator and is defined in Table 1. If ax is a negative term, then $ax \;\; inverse(\phi)a * d$ is called an *active bound*. We say that a set of bounds has the *same direction* if all bound operators of active bounds exist only in the set $\{>, \geq, =\}$ or the set$\{<, \leq, =\}$. Since K implies $t > 0$ or K implies $t < 0$, then all active bounds in the unsat equation must have the same direction.

```
1. procedure real_interpolant(i)
2.   move the basic variable x_i to the right side of the equation
3.   for each variable x_j in the equation
4.     if the active bound of x_j is from A_e then
5.       s := s + a_ij x_j
6.       c+ = a_ij * b_value(x_j)
7.     if a_ij <0 then op = inverse( b_operator(x_j))
8.     else op = b_operator(x_j)
9.     if op ≻ φ then φ =op
10. return σ(s φ c)
```

Fig. 5. Interpolant procedure over reals

Our method collects all active bounds from A_e in that equation to build an interpolant. Later on we will prove that the summation of all active bounds from A_e is an interpolant. The procedure $real\_interpolant(x_i)$ to build an interpolant over the reals is shown in Figure 5.

$real\_interpolant(i)$ will be immediately called after the linear arithmetic solver of Yices fails to find an assignment for the basic variable x_i in the unsat equation. The procedure takes as input an index i indicating the i^{th} equation(x_i is a basic variable). We move basic variable x_i to the right hand side of the equation to express it in the form $0 = t$. The variable s, initialized to the empty string, is the sum of the left hand sides of active bounds from A_e, c, initialized to zero, is the sum of corresponding bound values and ϕ is the active bound operator. $b\_value(x_j)$ returns the active bound value of x_j, and $b\_operator(x_j)$ returns the bound operator of x_j. In the procedure an ordering is placed on operators $' >' \succ ' \geq' \succ ' ='$, and $' <' \succ ' \leq' ' ='$. When adding active bounds together, the procedure selects the biggest operator among bound operators.

Example 1. Let's construct an interpolant from the example introduced in Section 3. The solver detects an unsat equation $2x = 2s_1 + s_2$ with its active bounds

$\{x \leq 0, s_1 = 0, s_2 \geq 1\}$. Running *real_interpolant*(i), we can derive $s = 2s_1 + s_2$, $c = 1$ and $\phi = '\geq'$ because $' \geq' \succ \ ' = '$. So, $s\phi c$ will be $2s_1 + s_2 \geq 1$. Then, the interpolant $\sigma(2s_1 + s_2 \geq 1)$ is $2x \geq 1$ because $s_1 = x - y$ and $s_2 = 2y$.

Lemma 1. *$\sigma(s\phi c)$ derived in real_interpolant(i) is an (A, B)-interpolant.*

5 YAI for Linear Equalities over Integers

Our method to check satisfiability of linear equations over integers is different from Yices. Yices employs the branch-and-bound and Gomory cut strategies to handle linear arithmetic over integers after a real assignment of variables is generated. Since in this section we only focus on integer equalities, then the Gomory cut strategy is not applicable. So, Yices only can use the branch-bound strategy. However, the branch-bound strategy is not suitable for generating interpolants because it repeatedly splits the problem into subproblems. Thus, this strategy not only increases the difficulty of constructing the interpolant, but it also makes the format of the interpolant complex involving the composition of disjunction and conjunction formulas. Therefore, we present a new linear arithmetic solver for linear equalities over integers and a simple method to construct interpolants.

Our Interpolants are standard equations or congruence equations because in some cases standard equations are not powerful to express the interpolant. For example, $A = \{2x = y\}$ and $B = \{2z = y + 1\}$. In this example, it is difficult to express the interpolant as standard equations because the interpolant could be a disjunction of infinitely many standard equations. But, if we introduce congruence equations, we can simply represent the interpolant as $y \equiv_2 0$. Another nice feature is that our method can treat congruence equations as inputs. This will be useful to find inductive invariants, when the interpolant may be an input to another call of the theorem prover.

We still use Yices' method to build the constraint matrix and bounds. In the constraint matrix we specify that all the coefficients of variables are integers. Basically, our method consists of two main procedures *reduce_matrix*() and *reduce_congruence_matrix*(). In those procedures some equations are removed from the matrices, and some equations are added into the matrices. If no equation is left After running those two procedures , then the original problem is satisfiable; else, the problem is unsatisfiable and an interpolant is constructed.

In our paper, we assume every congruence equation is simplified. The algorithm to simplify congruence equations is based on the Euclidian algorithm [4]. The definition of simplification of a congruence equation is given as follows.

Definition 2. *A congruence equation $ax \equiv_m t$ is simplified if a is a factor of m, x is a variable, t is an expression, and a and m are constants.*

The gcd test[4] is employed to check if an individual equation has an integer solution. The algorithm of the gcd test is presented in Figure 6. Notice that in the procedure, if equation i is a congruence equation, we consider the modulo as the coefficient of a free variable. Thus, *mod*(i) returns the modulo of Equation

1. **procedure** gcd_test(i)
2. **if**(all variables in Equation i are fixed) **then** $g = 0$
3. **else** g = gcd of coefficients of free variables in Equation i
4. **if** (Equation i is a congruence equation) **then** $g = gcd(g, mod(i))$
5. c is the summation of bound values of fixed variables
6. **if** (c is not a multiple of g) **then return** *fail*
7. **if** (all variables in Equation i are fixed) **then** remove Equation i
8. **return** *success*

Fig. 6. gcd test procedure

i. In certain cases, it is possible that there is no free variable in equation i. In this case Equation i could be removed if c is a multiple of g. It is straightforward that removing Equation i preserves satisfiability.

1. **procedure** reduce_matrix()
2. assert all the bounds at once
3. **if**(bound violation exists during assert) **then return** $bound\_interpolant(b_1, b_2)$
4. **if**(Equation p fails gcd test) **then return** $integer\_interpolant(p)$
5. **while**(the matrix is not empty)
6. select a free variable x_j from Equation i
7. build a set M of equations containing a variable x_j
8. **if**(for each $m \in M$ $|gcd(coef(x_j, i), coef(x_j, m))| > 1$) **then**
9. add $\sum_{k=1}^{n} a_{ik}x_k \equiv_{a_{ij}} 0$ into the congruence matrix
11. apply Superposition Rule to replace x_j in each equation in M by Equation i
12. **if**(Equation p fails gcd test) **then return** $integer\_interpolant(p)$
13. apply Congruence Superposition Rule to replace x_j by Equation i
14. **if**(Congruence Equation p fails gcd test) **then return** $integer\_interpolant(p)$
15. remove Equation i from the matrix
16. **if**(the congruence matrix is empty) **then return** *satisfiable*
17. **return** $reduce\_congruence\_matrix()$;

Fig. 7. Reduce_matrix procedure

Yices asserts bounds one by one. But, in our procedure $reduce\_matrix()$, we assert all the bounds at once. If a bound violation occurs while asserting bounds, $bound\_interpolant$ (b_1, b_2) is called to construct an interpolant where b_1 and b_2 are two inconsistent bounds. Notice that the element of the set M could be a standard equation or a congruence equation. $coef(x_j, i)$ is a function that returns the coefficient of x_j in equation i. Thus, $gcd(coef(x_j, i), coef(x_j, m))$ returns the gcd of coefficients of x_j in Equation i and Equation m. To preserve satisfiability, a congruence equation $\sum_{k=1}^{n} a_{ik}x_k \equiv_{a_{ij}} 0$ may be added into the congruence matrix where $\sum_{k=1}^{n} a_{ik}x_k$ is the summation of all the terms in equation i and a_{ij} is the coefficient of x_j in equation i. Then, the procedure replaces x_j in the other equations by Equation i. In the following inference rules, we treat the left premise as Equation i and the right premise as a updated equation.

The way to replace x_j in a standard equation by Equation i is given in **Superposition Rule**.

$$\frac{ax_j = t_1 \quad bx_j = t_2}{bt_1 = at_2}$$

where x_j is a free variable, t_1 and t_2 are expressions, and a and b are integers.

The idea to replace x_j in a congruence equation by a standard equation is given in **Congruence Superposition Rule**:

$$\frac{ax_j = t_1 \quad bx_j \equiv_c t_2}{bt_1 \equiv_{ac} at_2}$$

where x_j is a free variable, t_1 and t_2 are expressions, and a, b, and c are integers.

The above two inference rules represent updates, meaning that the right premise is replaced by the conclusion in the presence of the left premise. The left premise will not be removed from the matrix until all the other equations are updated. Thus, for each iteration in the *while* loop, an equation is removed from the constraint matrix in *reduce_matrix*(). Thus, in general our method is efficient because the number of constraints in the matrix is reduced and the cost to update standard equations is dramatically decreased.

Example 2. Given $A = \{2x = y\}$ and $B = \{y = 2z + 1\}$, this is flattened into equations $\{s_1 = 2x - y, s_2 = y - 2z\}$ and bounds $\{s_1 = 0, s_2 = 1\}$. Suppose that the ordering of the selected free variables in *reduce_matrix*() is x from $s_1 = 2x - y$ and y from $s_2 = y - 2z$. For the first pass of the while loop, $y \equiv_2 s_1$ is added into the congruence matrix. For the second pass of the while loop, the congruence superposition rule is applied for $s_2 = y - 2z$ and $y \equiv_2 s_1$. The inference result is $s_2 + 2z \equiv_2 s_1$ which can be simplified to $s_2 \equiv_2 s_1$. The equation $s_2 \equiv_2 s_1$ fails gcd test because $s_1 = 0$ and $s_2 = 1$. Thus, unsatisfiability is detected.

1. **procedure** reduce_congruence_matrix()
2. **while**(the congruence matrix is not empty)
3. select a free variable x_j from Congruence equation i
4. build a set M of congruence equations containing x_j
5. for an equation $m \in M$ **if** $|coef(x_j, m)| > 1$ **then**
6. add congruence Equation $\sum_{k=1}^{n} a_{mk}x_k \equiv_{a_{mj}} 0$
7. apply CC Superposition Rule replacing x_j for each pair of equations in M
8. remove all the equations in M
9. **if**(an equation m fails the gcd test) **then return** *integer_interpolant*(p)
10. **return** satisfiable

Fig. 8. Reduce congruence matrix procedure

Next, we will study the procedure *reduce_congruence_matrix*() which is shown in Figure 8. In the procedure, a set M is built in which each congruence equation contains x_j. For an equation $j \in M$, if the absolute value of the coefficient of x_j is greater than 1 then $\sum_{k=1}^{n} a_{mk}x_k \equiv_{a_{mj}} 0$ is added where $\sum_{k=1}^{n} a_{mk}x_k$

is a summation of all terms in Congruence Equation m and a_{mj} is the coefficient of x_j in congruence Equation m. Then **CC Superposition Rule** is applied for each pair of equations in M to replace x_j.

$$\frac{ax_j \equiv_c t_1 \quad bx_j \equiv_d t_2}{bt_1 \equiv_{gcd(b*c,a*d)} at_2}$$

After applying the CC Superposition Rule, all the equations containing x_j are removed. Next, gcd test is applied on those new generated equations. An interpolant is constructed if unsatisfiability is detected. This procedure is terminated when an interpolant is constructed or the congruence matrix is empty.

Since for each equation $a_i x \equiv_{m_i} t_i$ in M, $a_i | t_i (t_i$ is divisible by $a_i)$ because a congruence equation may be added. Thus, in some sense our method is the successive substitution method which is a method of solving problems of simultaneous congruences by using the definition of the congruence equation. Usually, the successive substitution method is applied in cases where the conditions of the Chinese Remainder Theorem are not satisfied.

Example 3. Given $A = \{6m = 3x - y, 4n = 3x - z\}$ and $B = \{3z = y + 1\}$, this is flattened into extension equations $\{s_1 = 6m - 3x + y, s_2 = 4n - 3x + z, s_3 = 3z - y\}$ and bounds $\{s_1 = 0, s_2 = 0, s_3 = 1\}$. Suppose the ordering of the selected variables in *reduce_matrix*() is m from $s_1 = 6m - 3x + y$, n from $s_2 = 4n - 3x + z$ and z from $s_3 = 3z - y$. After running *reduce_matrix*(), all those standard equations are removed and $\{3x - y + s_1 \equiv_6 0, 3x - z + s_2 \equiv_4 0, y + s_3 \equiv_3 0\}$ is added into the congruence matrix. So, the next step is to call *reduce_congruence_matrix*(). In this procedure, let's assume that x is selected. First, $3x - y + s_1 \equiv_6 0$ and $3x - z + s_2 \equiv_4 0$ can be rewritten to $3x \equiv_6 y - s_1$ and $3x \equiv_4 z - s_2$. Second, since the coefficient of x in those two congruence equations is 3, $y \equiv_3 s_1$ and $z \equiv_3 s_2$ are generated. Third, after applying the CC Superposition Rule for $3x \equiv_6 y - s_1$ and $3x \equiv_4 z - s_2$, the inference result is $y - s_1 \equiv_2 z - s_2$. So, at this point the congruence matrix contains $y + s_3 \equiv_3 0$, $y \equiv_3 s_1$, $z \equiv_3 s_2$ and $y - s_1 \equiv_2 z - s_2$. Suppose, the next selected variable is y. Then, applying the CC Superposition Rule for $y + s_3 \equiv_3 0$ and $y \equiv_3 s_1$, we can derive $s_1 + s_3 \equiv_3 0$ which is unsatisfiable because $s_1 = 0$ and $s_3 = 1$.

So far, we have introduced a new linear arithmetic solver different from Yices for linear equations over integers. Next is to show a procedure to construct an interpolant. In *integer_interpolant*(i), we consider that the modulo of Equation i is a coefficient of a free variable. s is a sum of the left hand sides of active bounds and c is a sum of corresponding bound values.

Example 4. In example 2, our method detects that $s_2 \equiv_2 s_1$ with $s_2 = 1$ and $s_1 = 0$ is unsatisfiable. Running *integer_interpolant*(i), we can derive $s = s_1$, $g = 2$ and $c = 0$ because s_1 is from A_e. Since $s_1 = 2x - y$ then an interpolant $\sigma(s_1 \equiv_2 0)$ could be $y \equiv_2 0$ which is simplified from $2x - y \equiv_2 0$. Similarly, in example 3, running *integer_interpolant*(i), we can derive an interpolant $y \equiv_3 0$.

```
1. procedure integer_interpolant(i)
2.   g = gcd of coefficients of free variables in Equation i
4.   if (Equation i is a congruence equation) then g = gcd(mod(i), g)
5.   for each variable x_j in the equation
6.     if the active bound of x_j is from A_e then
7.       s := s + a_ij x_j
8.       c+ = a_ij * b_value(x_j)
9.   σ(s ≡_g c)
10. return satisfiable
```

Fig. 9. Interpolant over integer equations

Notice that our arithmetic solver of YAI is different from Yices. However, our solver uses the same flattening procedure as Yices does to separate constraints and bounds. Thus, our solver is also able to quickly add and delete a single constraint. To delete a constraint, we can simply delete its bound. To add a constraint, first we assert its bound. Then, we re-run $gcd\_test(i)$ on the equations, including the deleted equations which contain the corresponding extension variable for that constraint. Actually, our method saves those deleted constraints for adding and deleting operations.

In our implementation, YAI does not construct a model for a satisfiable problem because our interest is to construct interpolants. However, theoretically our method is able to construct a model. We can build a model from the reverse order of the deleted constraints using the successive substitution method [7].

Lemma 2. *$\sigma(s \equiv_g c)$ derived in integer_interpolant(i) is an (A, B)-interpolant.*

Lemma 3. *Each pass of the while loop in reduce_matrix() preserves satisfiability.*

Lemma 4. *Each pass of the while loop in reduce_congruence_matrix() preserves satisfiability.*

6 YAI for Linear Equalities and Inequalities over Integers

The linear arithmetic procedure of Yices tries to find an assignment for each variable and if an assignment is not an integer then branch-bound and Gomory cut strategies are employed. In this section we use the linear arithmetic solver in Yices to check satisfiability for linear arithmetic over integers. But, in our implementation only the Gomory cut strategy is employed and the reason is explained in the previous section. Notice that the linear arithmetic solver in Yices is incomplete for integer problems. So, our method to construct interpolants based on Yices is also incomplete.

Given an equation $x_i = \sum_{x_j \in \mathcal{N}} a_{ij} x_j$ where the assignment of the basic variable x_i is a rational number, then a Gomory cut [8] is generated to restrict the solution of x_i. Gomory cut implied by constraints is an inequality. A Gomory cut is a *pure cut* when the equation used to generate the cut is from A_e or B_e.

Otherwise, it is a *mixed cut*. We treat all pure Gomory cuts as constraints from A_e or B_e. For a mixed Gomory cut, a conjunction of constraints from B_e, which are used to generate the cut, will be stored with the corresponding cut.

If unsatisfiability is detected in the asserting procedure, then we can use $bound\_\ interpolant(b_i, b_j)$ to construct the interpolant; else $cut\_interpolant(i)$ is called to construct the interpolant.

1. **procedure** cut_interpolant(i)
2. move the basic variable x_i to the left side of the equation
3. **for each** variable x_j in the equation
4. **if** the active bound of x_j is mixed cut or from A_e **then**
5. s += $a_{ij}x_j$
6. c += $(a_{ij} * b\_value(x_j)$
7. **if** a_{ij} <0 **then** op = inverse($b\_operator(x_j)$
8. **else** op = $b\_operator(x_j)$
9. **if** op $\succ \phi$ **then** ϕ =op
10. **if** the active bound of x_j is a mixed cut **then** b = b $\wedge$ $cut\_info(x_j)$
11. **return** σ (b $\Rightarrow$ (s ϕ c))

Fig. 10. Interpolant for linear arithmetic over integers

$cut\_interpolant(i)$ is quite similar to $real\_inerpolant(i)$. $cut\_info(x_j)$ is a set of constraints from B_e which are used to generate the cut of x_j. However, the strategy of Gomory cut is employed. So, if a mixed cut is involved in the unsatisfiability proof, then extracting information of A_e from the mixed cut is not enough because the mixed cut is derived from a combination of constraints from A_e and B_e. Thus, in the procedure we use b to collect some constraints from B_e which are used to generate corresponding cuts. We use $\Rightarrow$ as implication. Thus, we treat b as a condition to conclude $s\phi c$.

Since b is a set of some constraints from B_e, then b may contains some variables local to B_e. Thus, σ (b $\Rightarrow$ (s ϕ c)) is not an interpolant. But, it does satisfy the other two properties of (A, B)-interpolant.

Example 5. Given $A = \{5x = y + z, y \geq 0, y \leq 1\}$ and $B = \{z \geq 1, z \leq 2\}$, this can be flattened into extension equations $\{s_1 = 5x - y - z\}$ and bounds $\{s_1 = 0, y \geq 0, y \leq 1, z \geq 1, z \leq 2\}$. Running the linear arithmetic procedure in Yices, an assignment is generated but the value of x is $1/5$ which can be derived from $5x = y + z + s_1$ with $y \geq 0$, $z \geq 1$ and $s_1 = 0$. Then, a Gomory cut $y + z \geq 5$ is generated from that equation with its bounds. Notice that the constraint from B_e used to construct the Gomory cut is $z \geq 1$. That Gomory cut is flattened into an extension equation $s_2 = y + z$ and a bound $s_2 \geq 5$. The extension equation is added into matrix and the bound is asserted. Finally, unsatisfiability is detected in $z = -y + s_2$ with $z \leq 2$, $y \leq 1$ and $s \geq 5$. Thus, running $cut\_interpolant(i)$, we derive that an interpolant $\sigma(z \geq 1 \Rightarrow -y + s2 \geq 4)$ is $z \geq 1 \Rightarrow z \geq 4$ because $s_2 = y + z$. The left side of the conditional formula is the conjunction of constraints from B_e used to construct Gomory cuts and the right side of the conditional formula is the sum of constraints from A_e and Gomory cuts.

Lemma 5. *$\sigma(b \Rightarrow s\phi c)$ in cut_interpoant(i) is implied by A and contradicts B.*

7 Experimental Results

We implemented YAI using Microsoft Visual Studio 2008 C++ on the Windows XP operating system. All experiments are conducted on a Lenovo Think Pad T60 with the configuration of Intel 2.0GHz CPU and 2.0GB memory. YAI is available on *http://people.clarkson .edu/∼tangy/*.

Since YAI for linear real inequalities and integer inequalities is based on the linear arithmetic solver in Yices and constructing interpolants does not add too much additional cost to Yices, we only compared the running time of YAI with Yices for linear integer equalities.

Most SMT benchmarks contain disjunctions and function symbols. Thus, we wrote a C program to randomly generate sample examples without containing disjunctions and function symbols. Each sample example set contains 100 examples. Let $P(e_1, e_2)$ be a pair of integers and let e_1 be the number of constraints and e_2 be the number of variables in an example. In each example of *equality_set_*1, *equality_set_*2, and *equality_set_*3, the corresponding pairs are $P(10, 5)$, $P(50, 25)$ and $P(100, 50)$. For each constraint in the example the operator is $' ='$, the number of terms is randomly selected from $[1, 10]$, the coefficient for each term is randomly selected from $[1, 100]$, the variable name is randomly generated, and a constant is randomly selected from $[1, 100]$. The results in the table are the total runtime of 100 examples. The running time of YAI for linear integer equalities mainly depends on the number of congruence equations generated from the examples because the expensive operation of YAI is to reduce congruence equations using the successive substitution method.

	Number of Examples	YAI (seconds)	Yices(seconds)
*equality_set_*1	100	4	5
*equality_set_*2	100	8	10
*equality_set_*3	100	6	67

Fig. 11. Comparison between YAI and Yices for linear equalities over integers

8 Conclusion and Future Work

We have shown how interpolant construction fits easily into the SMT framework in the theory of linear arithmetic. In particular, we produce an interpolant directly from the single contradictory equation. Therefore, an interpolant can be generated at no additional cost to the satisfiability procedure. We have implemented YAI and compared our results against Yices.

We have several directions to work on. First, we plan to extend our method to handle uninterpreted function symbols and disjunctions using the methods proposed by Yorsh and Musuvathi[8] and McMillan [11]. Second, We would like

to compare against FOCI, CLP-prover, MATHSAT-ITP and INT2 in the size of interpolants and the efficiency of solvers. Finally, we would like to apply our method to generate invariants.

References

1. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The Software Model Checker Blast: Applications to Software Engineering. *International Journal on Software Tools for Technology Transfer (STTT)* 9(5-6), 505–525 (2007)
2. Chvatal, V.: *Linear Programming*. W. H. Freeman, New York (1983)
3. 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. Springer, Heidelberg (2008)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: *Introduction to Algorithms*, 2nd edn., pp. 856–862. MIT Press and McGraw-Hill (2001)
5. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *J. Symbolic Logic* 22(3), 269–285 (1957)
6. Dantzig, G.B., Curtis, B.: Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory*, 288–297 (1973)
7. Dickson, L.E.: History of the Theory of Numbers, vol. 2. Chelsea, New York (1957)
8. Dutertre, B., de Moura, L.: Integrating Simplex with DPLL(T). CSL Technical Report SRI-CSL-06-01 (2006)
9. Jain, H., Clarke, E.M., Grumberg, O.: Efficient Craig Interpolation for Linear Diophantine (Dis)Equations and Linear Modular Equations. In: *Proc. CAV* (2008)
10. 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)
11. McMillan, K.L.: An interpolating theorem prover. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 16–30. Springer, Heidelberg (2004)
12. McMillan, K.L.: Applications of Craig interpolants in model checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 1–12. Springer, Heidelberg (2005)
13. Rybalchenko, A., Stokkermans, V.S.: Constraint Solving for Interpolation. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 346–362. Springer, Heidelberg (2007)
14. Yorsh, G., Musuvathi, M.: A combination method for generating interpolants. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 353–368. Springer, Heidelberg (2005)