11. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In John Launchbury and John C. Mitchell, editors, *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL '02)*, pages 58–70, 2002.
12. S. K. Lahiri. An efficient decision procedure for the logic of Counters, Constrained Lambda expressions, Uninterpreted Functions and Ordering. Master's thesis, ECE Department, Carnegie Mellon University, May 2001.
13. S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In W. A. Hunt, Jr. and F. Somenzi, editors, *Computer-Aided Verification (CAV 2003)*, LNCS 2725, pages 141–153, 2003.
14. S. K. Lahiri, R. E. Bryant, A. Goel, and M. Talupur. Revisiting positive equality. Technical Report CMU-CS-03-196, Carnegie Mellon University, November 2003.
15. S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In J. W. O'Leary M. Aagaard, editor, *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, pages 142–159, Nov 2002.
16. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *38th Design Automation Conference (DAC '01)*, 2001.
17. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):245–257, 1979.
18. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, June 1992.
19. A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small-domain instantiations. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 455–469. Springer-Verlag, July 1999.
20. A. Pnueli, Y. Rodeh, O. Strichman, and M. Siegel. The Small Model Property: How Small Can It Be? Information and Computation. *Information and Computation*, 178(1):279–293, 2002.
21. Y. Rodeh and O. Strichmann. Finite Instantiations in Equivalence Logic with Uninterpreted Functions. In G. Berry, H. Comon, and A. Finkel, editors, *Computer-Aided Verification (CAV '01)*, LNCS 2102, pages 144–154, 2001.
22. R. E. Shostak. Deciding Combinations of Theories. *Journal of the ACM*, 31(1):1–12, 1984.

An Interpolating Theorem Prover

K.L. McMillan

Cadence Berkeley Labs

Abstract. We present a method of deriving Craig interpolants from proofs in the quantifier-free theory of linear inequality and uninterpreted function symbols, and an interpolating theorem prover based on this method. The prover has been used for predicate refinement in the BLAST software model checker, and can also be used directly for model checking infinite-state systems, using interpolation-based image approximation.

1 Introduction

A Craig interpolant [1] for an inconsistent pair of logical formulas (A, B) is a formula ϕ that is implied by A, inconsistent with B and refers only to variables common to A and B. If A and B are propositional formulas, and we are given a refutation of $A \wedge B$ by resolution steps, we can derive an interpolant for (A, B) in linear time [5,12]. This fact has been exploited in a method of over-approximate image computation based on interpolation [7]. This provides a complete symbolic method of model checking finite-state systems with respect to linear temporal properties. The method is based entirely on a proof-generating Boolean satisfiability solver and does not rely on quantifier elimination or reduction to normal forms such as binary decision diagrams (BDD's) or conjunctive normal form. In practice it was found to be highly effective in proving localizable properties of large circuits.

Here we present a first step in expanding this approach from propositional to first-order logic, and from finite-state to infinite-state systems. We present an interpolating prover for a quantifier-free theory that includes linear inequalities over the rationals and equality with uninterpreted function symbols. As in [2] the prover combines a Boolean satisfiability solver with a proof-generating ground decision procedure. After generating a refutation for a pair of formulas (A, B), the prover derives from this refutation an interpolant ϕ for the pair. The main contribution of this work is to show how interpolants can be derived from proofs in the combined theories of linear inequality and equality with uninterpreted function symbols (LIUF). This extends earlier work that handles only linear inequalities [12]. The combination of theories is useful, for example, for applications in software model checking.

The interpolating prover has been applied in the BLAST software model checking system [3]. This system is based on predicate abstraction, and uses interpolants as a guide in generating new predicates for abstraction refinement. The approach resulted in a substantial reduction in abstract state space size

K. Jensen and A. Podelski (Eds.): TACAS 2004, LNCS 2988, pp. 16–30, 2004.

relative to earlier methods. Further, using the method of [7], the prover can be used directly to verify some infinite-state systems, such as the Fischer and "bakery" mutual exclusion protocols. In principle, it can also be applied to the model checking phase of predicate abstraction.

The paper is organized as follows. In section 2, we introduce a simple proof system for LIUF, and show how refutations in this system can be translated into interpolants. Section 3 discusses the practicalities of constructing an efficient interpolating prover using this system. Finally, section 4 discusses actual and potential applications of the interpolating prover.

2 Interpolants from Proofs

We now describe a system of rules that, given a refutation of a pair of clause sets (A, B), derive an interpolant ϕ for the pair. For the sake of simplicity, we begin with a quantifier-free logic of with linear inequalities (LI). Then we treat a logic with equality and uninterpreted functions (EUF). Finally, we combine the two theories.

2.1 Linear Inequalities

A *term* in this logic is a linear combination $c_0+c_1v_1+\cdots c_nv_n$, where $v_1 \ldots v_n$ are distinct individual variables, $c_0 \ldots c_n$ are rational constants, and further $c_1 \ldots c_n$ are non-zero. When we perform arithmetic on terms, we will assume they are reduced to this normal form. That is, if x is a term and c is a non-zero constant, we will write cx to denote the term obtained by distributing the coefficient c inside x. Similarly, if x and y are terms, we will write $x + y$ to denote the term obtained by summing like terms in x and y and dropping resulting terms with zero coefficients. Thus, for example, if x is the term $1+a$ and y is the term $b-2a$ then $2x + y$ would denote the term $2 + b$.

An *atomic predicate* in the logic is either a propositional variable or an inequality of the form $0 \leq x$, where x is a term. A literal is either an atomic predicate or its negation. A clause is a disjunction of literals. We will write the clause containing the set of literals Γ as $\langle\Gamma\rangle$. In particular, we will distinguish syntactically between a literal l and the clause $\langle l\rangle$ containing just l. The empty clause, equivalent to false, will be written $\langle\rangle$.

A *sequent* is of the form $\Gamma \vdash \Delta$, where Γ and Δ are sets of formulas (in this case, either literals or clauses). The interpretation of $\Gamma \vdash \Delta$ is that the conjunction of the formulas in Γ entails the disjunction of the formulas in Δ. In what follows, lower case letters generally stand for formulas and upper case letters for sets of formulas. Further, a formula in a place where a set is expected should be taken as the singleton containing it, and a list of sets should be taken as their union. Thus, for example, the expression $\Gamma, \phi \vdash p, A$ should be taken as an abbreviation for $\Gamma \cup \{\phi\} \vdash \{p\} \cup A$.

Our theorem prover generates refutations for sets of clauses using the following proof rules:

$$\textsc{Hyp}\frac{}{\Gamma \vdash \phi}\ \phi \in \Gamma \qquad \textsc{Comb}\frac{\Gamma \vdash 0 \leq x \quad \Gamma \vdash 0 \leq y}{\Gamma \vdash 0 \leq c_1 x + c_2 y}\ c_{1,2} > 0$$

$$\textsc{Contra}\frac{l_1, \ldots, l_n \vdash \bot}{\Gamma \vdash \langle \neg l_1, \ldots, \neg l_n \rangle} \qquad \textsc{Res}\frac{\Gamma \vdash \langle l, \Theta \rangle \quad \Gamma \vdash \langle \neg l, \Theta' \rangle}{\Gamma \vdash \langle \Theta, \Theta' \rangle}$$

In the above, $\bot$ is a shorthand for $0 \leq -1$. All Boolean reasoning is done by the resolution rule Res. This system is complete for refutation of clause systems over the rationals. As in [10], we can obtain an incomplete system for the integers rather than the rationals by systematically translating the literal $\neg(0 \leq x)$ to $0 \leq -1 - x$.

We will use the notation $\phi \preceq \Gamma$ to indicate that all variables occurring in ϕ also occur in Γ. A term x is *local* with respect to a pair (A, B) if it contains a variable not occurring in B (in other words $x \not\preceq B$) and global otherwise.

In order to represent the rules for deriving interpolants from proofs, we will define several classes of *interpolations*. These have the general syntactic form $(A, B) \vdash \phi\ [X]$, where the exact form of X depends on the class. Intuitively, X is a representation of an "interpolant" associated with the deduction of ϕ from A and B. In the case where ϕ is the empty clause, X should in fact be an interpolant for (A, B). For each class of interpolation, we will define a notion of validity, and introduce derivation rules that are sound, in the sense that they derive only valid interpolations from valid interpolations. To save space, we will sketch the more difficult soundness arguments here, and leave the more straightforward ones to the reader.

Definition 1. *An* inequality interpolation *has the form* $(A, B) \vdash 0 \leq x\ [x', \rho, \gamma]$, *where* A *and* B *are clause sets,* x *and* x' *are terms, and* ρ *and* γ *are formulas, such that* $\rho, \gamma \preceq B$ *and* $x', \rho, \gamma \preceq A$. *It is said to be* valid *when:*

- $A, \rho \models 0 \leq x' \wedge \gamma$
- $B \models \rho$ *and* $B, \gamma \models 0 \leq x - x'$ *and,*
- *for all individual variables* v, *such that* $v \not\preceq B$, *the coefficients of* v *in* x *and* x' *are equal.*

For the current system, the formulas ρ and γ are always $\top$. They will play a role later, when we combine theories. The intuition behind this definition is that $0 \leq x$ is a linear combination of inequalities from A and B, where x' represents the contribution to x from A. We now begin with the interpolation rule for introduction of hypotheses. Here, we distinguish two cases, depending on whether the hypothesis is from A or B:

$$\textsc{HypLeq-A}\frac{}{(A, B) \vdash 0 \leq x\ [x, \top, \top]}\ (0 \leq x) \in A$$

$$\textsc{HypLeq-B}\frac{}{(A,B) \vdash 0 \leq x\ [0,\top,\top]}\ (0 \leq x) \in B$$

The soundness of these rules (*i.e.*, validity of their consequents, given the side conditions) is easily verified. The rule for combining inequalities is as follows:

$$\textsc{Comb}\frac{(A,B) \vdash 0 \leq x\ [x',\rho,\gamma] \quad (A,B) \vdash 0 \leq y\ [y',\rho',\gamma']}{(A,B) \vdash 0 \leq c_1x + c_2y\ [c_1x' + c_2y', \rho \wedge \rho', \gamma \wedge \gamma']}\ c_{1,2} > 0$$

In effect, we derive the interpolant for a linear combination of inequalities by taking the same linear combination of the contributions from A. Again, the reader may wish to verify that the validity conditions for inequality interpolations are preserved by this rule.

Example 1. As an example, let us derive an interpolant for the case where A is $(0 \leq y - x)(0 \leq z - y)$ and B is $(0 \leq x - z - 1)$. For clarity, we will abbreviate $(A,B) \vdash \phi\ [x,\top,\top]$ to $\vdash \phi\ [x]$. We first use the HypLeq-A rule to introduce two hypotheses from A:

$$\textsc{HypLeq-A}\frac{}{\vdash 0 \leq y - x\ [y-x]} \qquad \textsc{HypLeq-A}\frac{}{\vdash 0 \leq z - y\ [z-y]}$$

Now, we sum these two inequalities using the Comb rule:

$$\textsc{Comb}\frac{\vdash 0 \leq y - x\ [y-x] \quad \vdash 0 \leq z - y\ [z-y]}{\vdash 0 \leq z - x\ [z-x]}$$

Now we introduce a hypothesis from B:

$$\textsc{HypLeq-B}\frac{}{\vdash 0 \leq x - z - 1\ [0]}$$

Finally, we sum this with our previous result, to obtain $0 \leq -1$, which is false:

$$\textsc{Comb}\frac{\vdash 0 \leq z - x\ [0 \leq z-x] \quad \vdash 0 \leq x - z - 1\ [0 \leq 0]}{\vdash 0 \leq -1\ [z-x]}$$

You may want to check that all the interpolations derived are valid. Also notice that in the last step we have derived a contradiction, and that $0 \leq z - x$ is an interpolant for (A,B).

Now we introduce an interpolation syntax for clauses, to handle Boolean reasoning. If Θ is a set of literals, we will denote by $\Theta \downarrow B$ the literals of Θ occurring in B and by $\Theta \setminus B$ the literals *not* occurring in B.

Definition 2. *A* clause interpolation *has the form* $(A,B) \vdash \langle\Theta\rangle\ [\phi]$*, where* A *and* B *are clause sets,* Θ *is a literal set and* ϕ *is a formula. It is said to be* valid *when:*

- $A \models \phi \vee \langle \Theta \setminus B \rangle$, *and*
- $B, \phi \models \langle \Theta \downarrow B \rangle$, *and*
- $\phi \preceq B$ *and* $\phi \preceq A$.

Notice that if Θ is empty, ϕ is an interpolant for (A, B). Two rules are needed for introduction of clauses as hypotheses:

$$\textsc{HypC-A}\frac{}{(A,B) \vdash \langle \Theta \rangle \ [\langle \Theta \downarrow B \rangle]} \ \langle \Theta \rangle \in A \quad \textsc{HypC-B}\frac{}{(A,B) \vdash \langle \Theta \rangle \ [\top]} \ \langle \Theta \rangle \in B$$

Note that the derived interpolations are trivially valid, given the side conditions. Now, we introduce two interpolation rules for resolution of clauses. The first is for resolution on a literal *not* occurring in B:

$$\textsc{Res-A}\frac{\begin{array}{c}(A,B) \vdash \langle l, \Theta \rangle \ [\phi] \\ (A,B) \vdash \langle \neg l, \Theta' \rangle \ [\phi']\end{array}}{(A,B) \vdash \langle \Theta, \Theta' \rangle \ [\phi' \vee \phi']} \ l \not\preceq B$$

Soundness. For the first condition, we know that A implies $\phi \vee l \vee \langle \Theta \setminus B \rangle$ and $\phi' \vee \neg l \vee \langle \Theta' \setminus B \rangle$. By resolution on l we have A implies $(\phi \vee \phi') \vee \langle \Theta, \Theta' \setminus B \rangle$. For the second condition, given B, we know that $\phi \implies \langle \Theta \downarrow B \rangle$ and $\phi' \implies \langle \Theta' \downarrow B \rangle$. Thus, $\phi \vee \phi'$ implies $\langle \Theta, \Theta' \downarrow B \rangle$. The third condition is trivial.

The second rule is for resolution on a literal occurring in B:

$$\textsc{Res-B}\frac{\begin{array}{c}(A,B) \vdash \langle l, \Theta \rangle \ [\phi] \\ (A,B) \vdash \langle \neg l, \Theta' \rangle \ [\phi']\end{array}}{(A,B) \vdash \langle \Theta, \Theta' \rangle \ [\phi \wedge \phi']} \ l \preceq B$$

Soundness. For the first validity condition, we know that A implies $\phi \vee \langle \Theta \setminus B \rangle$ and $\phi' \vee \langle \Theta' \setminus B \rangle$. These in turn imply $(\phi \wedge \phi') \vee \langle \Theta, \Theta' \setminus B \rangle$. For the second condition, given B, we know that $\phi \implies l \vee \langle \Theta \downarrow B \rangle$ while $\phi' \implies \neg l \vee \langle \Theta' \downarrow B \rangle$. By resolution, we have that $\phi \wedge \phi'$ implies $\langle \Theta, \Theta' \downarrow B \rangle$. The third condition is trivial.

Example 2. As an example, we derive an interpolant for (A, B), where A is $\langle b \rangle, \langle \neg b \vee c \rangle$ and B is $\langle \neg c \rangle$. First, using the HypC-A rule, we introduce the two clauses from A as hypotheses:

$$\textsc{HypC-A}\frac{}{\vdash \langle b \rangle \ [\bot]} \qquad \textsc{HypC-A}\frac{}{\vdash \langle \neg b, c \rangle \ [c]}$$

We now resolve these two clauses on the variable b.

$$\textsc{Res-A}\frac{\vdash \langle b \rangle \ [\bot] \qquad \vdash \langle \neg b, c \rangle \ [c]}{\vdash \langle c \rangle \ [\bot \vee c]}$$

We then use the Hyp-B rule to introduce the clause from B.

$$\textsc{Hyp-B}\frac{}{\vdash \langle \neg c \rangle \ [\top]}$$

Finally, we resolve the last two clauses on c. We use the RES-B rule, since c occurs in B.

$$\textsc{Res-B}\frac{\vdash \langle c\rangle\ [c] \qquad \vdash \langle \neg c\rangle\ [\top]}{\vdash \langle\rangle\ [c \wedge \top]}$$

Thus c is an interpolant for (A, B).

Finally, we introduce a rule to connect inequality reasoning to Boolean reasoning:

$$\textsc{Contra}\frac{(\{a_1, \ldots, a_k\}, \{b_1, \ldots, b_m\}) \vdash 0 \leq -1\ [x', \rho, \gamma]}{(A, B) \vdash \langle \neg a_1, \ldots, \neg a_k, \neg b_1, \ldots, \neg b_m\rangle\ \ [\rho \implies (0 \leq x' \wedge \gamma)]}$$

where the b_i are literals occurring in B, and the a_i are literals *not* occurring in B.
Soundness. By the first condition of Definition 1, $\bigwedge a_i$ implies $\rho \implies (0 \leq x' \wedge \gamma)$. Moreover, $\bigvee \neg a_i$ is precisely $\langle \Theta \setminus B\rangle$ in Definition 2. This means that $\phi \vee \langle \Theta \setminus B\rangle$ is a tautology, satisfying the first validity condition. For the second condition, suppose that $\rho \implies (0 \leq x' \wedge \gamma)$ holds. From the first two conditions of Definition 1, we infer that $\bigwedge b_i$ implies $0 \leq -1$. Thus, $\langle \neg b_1, \ldots, \neg b_m\rangle$ holds. Finally, the third validity condition is guaranteed by the third condition of Definition 1.

2.2 Equality and Uninterpreted Functions

In our logic of equality and uninterpreted functions, a term is either an individual variable or a function application $f^n(x_1, \ldots, x_n)$ where f^n is a n-ary function symbol and $x_1 \ldots x_n$ are terms. An atomic predicate is a propositional variable or an equality of the form $x = y$ where x and y are terms. Refutations are generated using the following proof rules (in addition to the HYP rule):

$$\textsc{Refl}\frac{}{\Gamma \vdash x = x} \qquad \textsc{Symm}\frac{\Gamma \vdash x = y}{\Gamma \vdash y = x}$$

$$\textsc{Trans}\frac{\Gamma \vdash x = y \quad \Gamma \vdash y = z}{\Gamma \vdash x = z} \qquad \textsc{Cong}\frac{\Gamma \vdash x_1 = y_1 \quad \ldots \quad \Gamma \vdash x_n = y_n}{\Gamma \vdash f^n(x_1, \ldots, x_n) = f^n(y_1, \ldots, y_n)}$$

$$\textsc{EqNeq}\frac{\Gamma \vdash x = y}{\Gamma \vdash \bot}\ \neg(x = y) \in \Gamma$$

Boolean reasoning can be added to the system by adding the CONTRA and RES rules of the previous system.

Definition 3. *An* equality interpolation *has the form* $(A, B) \vdash x = y\ [x', y', \rho, \gamma]$*, where* A *and* B *are clause sets,* x, y, x', y' *are terms, and* ρ *and* γ *are formulas, such that* $\rho, \gamma \preceq B$*. It is said to be* valid *when:*

- $A, \rho \models x = x' \wedge y = y' \wedge \gamma$,
- $B \models \rho$ *and* $B, \gamma, x = x', y = y' \models x = y' \wedge y = x'$,
- *either* $x', y' \preceq B$ *or* $x' = y$ *and* $y' = x$ *(the* degenerate *case), and*
- *if* $x \preceq A$ *then* $x' \preceq A$*, else* $x' = x$*, and similarly for* y, y'.

The intuition behind this definition is that x' and y' solutions for local terms x and y in terms of common variables (except in the degenerate case). The formula ρ is always $\top$ in the present theory. The interpolation rules for equality are as follows:

$$\textsc{HypEq-A}\frac{}{(A,B) \vdash x = y\ [y, x, \top, \top]}\quad (x = y) \in A$$

$$\textsc{HypEq-B}\frac{}{(A,B) \vdash x = y\ [x, y, \top, \top]}\quad (x = y) \in B$$

$$\textsc{Refl}\frac{}{(A,B) \vdash x = x\ [x, x, \top, \top]} \qquad \textsc{Symm}\frac{(A,B) \vdash x = y\ [x', y', \rho, \gamma]}{(A,B) \vdash y = x\ [y', x', \rho, \gamma]}$$

$$\textsc{Trans}\frac{\begin{array}{c}(A,B) \vdash x = y\ [x', y', \rho, \gamma] \\ (A,B) \vdash y = z\ [y'', z', \rho', \gamma']\end{array}}{(A,B) \vdash x = z\ [x', z', \rho \wedge \rho' \wedge y' = y'', \gamma \wedge \gamma']}\quad x' \neq y, z' \neq y$$

$$\textsc{Trans}'\frac{\begin{array}{c}(A,B) \vdash x = y\ [x', y', \rho, \gamma] \\ (A,B) \vdash y = z\ [y'', z', \rho', \gamma']\end{array}}{(A,B) \vdash x = z\ [x'(y''/y), z'(y'/y), \rho \wedge \rho', \gamma \wedge \gamma']}\quad x' = y \text{ or } z' = y$$

where $x(y/z)$ denotes y if $x = z$ else x. Of these, only the transitivity rules require a detailed soundness argument. The Trans rule is the for the case when neither antecedent is degenerate. The first condition of Definition 3 is trivial. For the second, suppose B, $\rho, \rho', y' = y'', x = x'$ and $x = z'$ hold. By inductive hypothesis, we know that $x = y'$ and $z = y''$. This, since $y' = y''$, we have $x = z'$ and $z = x'$. For the third condition, the side condition of the rule ensures that $x', z' \preceq B$. The Trans$'$ rule handles the case when one of the two antecedents is degenerate. Suppose that $x' = y$ and $z' \neq y$. Again, the first condition is trivial. For the second condition, suppose B, $\rho \wedge \rho'$, $x = y''$ and $z = z'$. By inductive hypothesis, $z = y''$, hence $x = z'$. On the other hand, if $x' = y$ and $z' = y$, then we have immediately $x = z$ (*i.e.*, the consequent is also degenerate). The third condition is straightforward.

Now we consider the Cong rule for uninterpreted functions symbols. Suppose that from $x = y$ we deduce $f(x) = f(y)$ by the Cong rule. Except in the degenerate case, we must obtain solutions for $f(x)$ and $f(y)$ in terms of variables occurring in B. We can obtain this by simply substituting the solutions for x and y into f, as follows:

$$\textsc{Cong}\frac{(A,B) \vdash x = y\ [x', y', \rho, \gamma]}{(A,B) \vdash f(x) = f(y)\ [f(x'), f(y'), \rho, \gamma]}$$

Note that if the antecedent is degenerate, the consequent is also degenerate. The generalization of this rule to n-ary function is somewhat complex, and we omit it here for lack of space.

Example 3. Suppose A is $x = y$ and B is $y = z$ and we wish to derive an interpolation for $f(x) = f(z)$. After introducing our two hypotheses, we use the Trans′ rule to get $x = z$:

$$\textsc{Trans}'\frac{\vdash x = y\ [y, x, \top, \top] \quad \vdash y = z\ [y, z, \top, \top]}{\vdash x = z\ [y, z, \top, \top]}$$

We then apply the Cong rule to obtain $f(x) = f(z)$:

$$\textsc{Cong}\frac{\vdash x = z\ [y, z, \top, \top]}{\vdash f(x) = f(z)\ [f(y), f(z), \top, \top]}$$

Finally, we deal with the EqNeq rule, which derives false from an equality and its negation. First, we consider the case where the disequality is contained in A:

$$\textsc{EqNeq-A}\frac{(A, B) \vdash x = y\ [x', y', \rho, \gamma]}{(A, B) \vdash 0 \leq -1\ [0, \rho, \gamma \wedge (x' \neq y')]}\ (x \neq y) \in A,\ y' \neq x$$

Notice that we derive an inequality interpolation here so that we can then apply the Contra rule. The idea is to translate the disequality over local terms to an equivalent disequality over global terms. We handle the degenerate case separately:

$$\textsc{EqNeq-A}'\frac{(A, B) \vdash x = y\ [y, x, \rho, \gamma]}{(A, B) \vdash 0 \leq -1\ [0, \rho, \bot]}\ (x \neq y) \in A$$

The case where the disequality comes from B is handled as follows:

$$\textsc{EqNeq-B}\frac{(A, B) \vdash x = y\ [x', y', \rho, \gamma]}{(A, B) \vdash 0 \leq -1\ [0, \rho, \gamma \wedge x = x' \wedge y = y']}(x \neq y) \in B$$

In the above, if x and x' are syntactically equal, we replace $x = x'$ with $\top$, and similarly for y and y'. This fulfills the requirement that $\gamma \preceq A$, in the case when $x \not\preceq A$ or $y \not\preceq A$.

2.3 Combining LI and EUF

A term in the combined logic is a linear combination $c_0 + c_1 v_1 + \cdots c_n v_n$, where $v_1 \ldots v_n$ are distinct individual variables and $c_0 \ldots c_n$ are integer constants, or a function application $f^n(x_1, \ldots, x_n)$ where f^n is a n-ary function symbol and $x_1 \ldots x_n$ are terms. An atomic predicate is either a propositional variable, an inequality of the form $0 \leq x$, where x is a linear combination, or an equality of the form $x = y$ where x and y are terms.

Our proof system consists of all the previous proof rules, with the addition of the following two rules that connect equality and inequality reasoning:

$$\textsc{LeqEq}\frac{\Gamma \vdash x = y}{\Gamma \vdash 0 \leq x - y}$$

$$\textsc{EqLeq}\frac{\Gamma \vdash 0 \leq x - y \quad \Gamma \vdash 0 \leq y - x}{\Gamma \vdash x = y}$$

The LeqEq rule, inferring an inequality from an equality, can be handled by the following interpolation rule:

$$\textsc{LeqEq}\frac{(A, B) \vdash x = y\ [x', y', \rho, \gamma]}{(A, B) \vdash 0 \leq x - y\ [x - x' - y + y', \rho, \gamma]}\ y' \neq x$$

The idea here is that B and γ give us $x' = y'$, thus $0 \leq x - x' - y + y'$ gives $0 \leq x - y$. However, we must deal separately with the special case where the antecedent is degenerate:

$$\textsc{LeqEq}'\frac{(A, B) \vdash x = y\ [y, x, \rho, \gamma]}{(A, B) \vdash 0 \leq x - y\ [x - y, \rho, \gamma]}$$

We now consider the EqLeq rule, which derives an equality from a pair of inequalities. We distinguish three cases, depending on whether x and y are local or global. The first case is when both x and y are global, and is straightforward:

$$\textsc{EqLeq-BB}\frac{\begin{array}{c}(A, B) \vdash 0 \leq x - y\ [x', \rho, \gamma] \\ (A, B) \vdash 0 \leq y - x\ [y', \rho', \gamma']\end{array}}{(A, B) \vdash x = y\ [x, y, \rho, 0 \leq x' \wedge 0 \leq y']}\quad x \preceq B, y \preceq B$$

Note that x and y in the above rule may be arbitrary linear combinations.

The case when x is local and y is global is more problematic. Suppose, for example, that A is $(0 \leq a - x)(0 \leq z - a)$ and B is $(0 \leq b - z)(0 \leq x - b)$. From this we can infer $0 \leq b - a$ and $0 \leq a - b$, using the Comb rule. Thus, using the EqLeq rule, we infer $a = b$. To make an interpolation for this, we must have a solution for a in terms of global variables, implied by A. Unfortunately, there are no equalities that can be inferred from A alone. In fact, it is not possible in general to express interpolants in the combined theory as conjunctions of atomic predicates as we have done up to now, because the combined theory is no longer convex in the sense of [10].[1]

This is the reason the parameter ρ was introduced in the interpolation syntax. In our example, we will have

$$(A, B) \vdash a = b\ [z, 0 \leq x - z, 0 \leq z - x]$$

[1] Consider the atomic formulas $0 \leq a - x$, $0 \leq y - a$, $f(a) = f(q)$. These imply the disjunction $f(x) = f(q) \vee x < y$, but do not imply either disjunct by itself.

That is, A proves $a = z$, under the condition ρ that $0 \leq x - z$. This interpolation is valid, since from B we can prove $0 \leq x - z$. Using A and this fact, we can infer $a = z$. From A we can also infer $0 \leq z - x$, which, with B, gives us $z = b$, hence $a = b$. This approach can be generalized to the following rule:

$$\textsc{EqLeq-AB}\frac{\begin{array}{c}(A,B) \vdash 0 \leq y - x\ [x', \rho, \gamma] \\ (A,B) \vdash 0 \leq x - y\ [x'', \rho', \gamma']\end{array}}{\begin{array}{r}(A,B) \vdash x = y\ [x + x', y, \rho \wedge \rho' \wedge 0 \leq -x' - x'', \\ \gamma \wedge \gamma' \wedge 0 \leq x' + x'']\end{array}} \quad x \not\preceq B, y \preceq B$$

The final case for the EqLeq rule is when $x \not\preceq B$ and $y \not\preceq B$:

$$\textsc{EqLeq-AA}\frac{\begin{array}{c}(A,B) \vdash 0 \leq y - x\ [\rho, x', \gamma] \\ (A,B) \vdash 0 \leq x - y\ [\rho', x'', \gamma']\end{array}}{\begin{array}{r}(A,B) \vdash x = y\ [y, x, \rho \wedge \rho' \wedge 0 \leq y - x - x' \\ \wedge\, 0 \leq x - y - x'', \gamma \wedge \gamma']\end{array}} \quad x \not\preceq B, y \not\preceq B$$

Soundness. Given B, we know from Definition 1 that $0 \leq y - x - x'$ and $0 \leq x - y - x''$. Note that all variables not occurring in B cancel out in these terms. Moreover, A gives us $0 \leq x'$ and $0 \leq x''$. Combining with the above, we get $x = y$.

2.4 Soundness and Completeness

The following two theorems state the soundness and completeness results for our interpolation system:

Theorem 1 (Soundness). *If a clause interpolation of the form* $(A, B) \vdash \langle\rangle\ [\phi]$ *is derivable, then* ϕ *is an interpolant for* (A, B).

Proof sketch. Validity of the interpolation is by the soundness of the individual interpolation rules and induction over the derivation length. By Definition 2 we know that A implies ϕ, that B and ϕ are inconsistent and that $\phi \preceq B$.

Theorem 2 (Completeness). *For any derivable sequent* $A, B \vdash \psi$, *there is a derivable interpolation of the form* $(A, B) \vdash \psi\ [X]$.

Proof sketch. We split cases on the rule used to derive the sequent, and show in each case that there is always a rule to derive an interpolation for the consequent from interpolations for the antecedents.

In effect, the proof of the completeness theorem gives us an algorithm for constructing an interpolant from a refutation proof. This algorithm is linear in the proof size, and the result is a formula (not in CNF) whose circuit size is also linear in the proof size.

3 An Interpolating Prover

Thus far we have described a proof system for a logic with linear inequalities and uninterpreted functions, and set of rules for deriving interpolants from proofs in this system. There are two further problems that we must address: constructing an efficient proof-generating decision procedure for our system, and translating interpolation problems for general formulas into interpolation problems in clause form.

3.1 Generating Proofs

The prover combines a DPLL style SAT solver, similar to Chaff [9], for propositional reasoning, with a proof-generating Nelson-Oppen style ground decision procedure[2] for theory reasoning. They are combined using the "lazy" approach of [2]. That is, the SAT solver treats all atomic predicates in a given formula f as free Boolean variables. When it finds an assignment to the atomic predicates that satisfies f propositionally, it passes this assignment to the ground decision procedure in the form of a set of literals $l_1 \ldots l_n$. The ground decision procedure then attempts to derive a refutation of this set of literals. If it succeeds, the literals used as hypotheses in the refutation are gathered (call them $m_1, \ldots, m_k$). The CONTRA rule is then used to derive the new clause $\langle \neg m_1, \ldots, \neg m_k \rangle$. This clause is added to the SAT solver's clause set. We will refer to it as a *blocking clause*. Since it is in conflict in the current assignment, the SAT solver now backtracks, continuing where it left off. On the other hand, if the ground decision procedure cannot refute the satisfying assignment, the formula f is satisfiable and the process terminates.

The SAT solver is modified in a straightforward way to generate refutation proofs by resolution (see [8] for details). When a conflict occurs in the search (*i.e.*, when all the literals in some clause are assigned to false), the solver resolves the conflicting clause with other clauses to infer a so-called "conflict clause" (a technique introduced in the GRASP solver [14] and common to most modern DPLL solvers). This inferred clause is added to the clause set, and in effect prevents the same conflict from occurring in the future. The clause set is determined to be unsatisfiable when the empty clause (false) is inferred as a conflict clause. To derive a proof of the empty clause, we have only to record the sequence of resolutions steps used to derive each conflict clause.

The SAT solver's clause set therefore consists of three classes of clauses: the original clauses of f, blocking clauses (which are tautologies proved by the ground decision procedure) and conflicts clauses (proved by resolution). When the empty clause is derived, we construct a refutation of f using the stored proofs of the blocking clauses and the conflict clauses.

[2] The current implementation uses the VAMPYRE proof-generating decision procedure (see `http://www.eecs.berkeley.edu/~rupak/Vampyre`). Proofs in its LF-style proof system are translated to the system used here.

3.2 Interpolants for Structured Formulas

Of course, the interpolation problem (A, B) is not in general given in the clause form required by our proof system. In general, A and B have arbitrary nesting of Boolean operators. We now show how to reduce the problem of finding an interpolant for arbitrary formulas (A, B) into the problem of finding an interpolant for (A_c, B_c) where A_c and B_c are in clause form.

It is well known that *satisfiability* of an arbitrary formula f can be reduced in linear time to satisfiability of a clause form formula [11]. This transformation uses a set V of fresh Boolean variables, containing a variable v_g for each non-atomic propositional subformula g of f. A small set of clauses is introduced for each occurrence of a Boolean operator in f. For example, if the formula contains $g \wedge h$, we add the clauses $\langle v_g, \neg v_{g\wedge h}\rangle$, $\langle v_h, \neg v_{g\wedge h}\rangle$ and $\langle \neg v_g, \neg v_h, v_{g\wedge h}\rangle$. These clauses constrain $v_{g\wedge h}$ to be the conjunction of v_g and v_h. We will refer to the collection of these clauses for all non-atomic subformulas of f as $\textsc{Cnf}_V(f)$. We then add the clause $\langle v_f\rangle$ to require that the entire formula is true. The resulting set of clauses is satisfiable exactly when f is satisfiable.

In fact, we can show something stronger, which is that any formula implied by $CNF_V(f) \wedge v_f$ that does not refer the fresh variables in V is also implied by f. This gives us the following result:

Theorem 3. *Let $A_c = \textsc{Cnf}_U(A), \langle u_A\rangle$ and $B_c = \textsc{Cnf}_V(B), \langle v_B\rangle$, where U,V are disjoint sets of fresh variables, and A, B are arbitrary formulas. An interpolant for (A_c, B_c) is also an interpolant for (A, B).*

This theorem allows us to compute interpolants for structured formulas by using the standard translation to clause form.

4 Applications

The interpolating prover described above has a number of possible applications in formal verification. These include refinement in predicate abstraction, and model checking infinite-state systems, with and without predicate abstraction.

4.1 Using Interpolation for Predicate Refinement

Predicate abstraction [13] is a technique commonly used in software model checking in which the state of an infinite-state system is represented abstractly by the truth values of a chosen set of predicates. Typically, when the chosen predicates are insufficient to prove the property in question, the abstraction is refined by adding predicates. For this purpose, the Blast software model checker uses the interpolating prover in a technique due to Ranjit Jhala [3].

The basic idea of the technique is as follows. A counterexample is a sequence of program locations (a path) that leads from the program entry point to an error location. When the model checker finds a counterexample in the abstract model, it builds a formula that is satisfiable exactly when the path is a counterexample

in the concrete model. This formula consists of a set of constraints: equations that define the values of program variables in each location in the path, and predicates that must be true for execution to continue along the path from each location (these correspond to program branch conditions).

Now let us divide the path into two parts, at state k. Let A_k be the set of constraints on transitions preceding state k and let B_k be the set of constraints on transitions subsequent to state k. Note that the common variables of A and B represent the values of the program variables at state k. An interpolant for (A_k, B_k) is a fact about state k that must hold if we take the given path to state k, but is inconsistent with the remainder of the path. In fact, if we derive such interpolants for every state of the path from the same refutation of the constraint set, we can show that the interpolant for state k is sufficient to prove the interpolant for state $k+1$. As a result, if we add the interpolants to the set of predicates defining the abstraction, we are guaranteed to rule out the given path as a counterexample in the abstract model. Alternatively, we use the set of atomic predicates occurring in the interpolants, with the same result.

This interpolation approach to predicate refinement has the advantage that it tells us which predicates are relevant to each program location in the path. By using at each program location only predicates that are relevant to that location, a substantial reduction in the number of abstract states can be achieved, resulting in greatly increased performance of the model checker [3]. The fact that the interpolating prover can handle both linear inequalities and uninterpreted functions is useful, since linear arithmetic can represent operations on index variables, while uninterpreted functions can be used to represent array lookups or pointer dereferences, or to abstract unsupported operations (such as multiplication).[3]

4.2 Model Checking with Interpolants

Image computation is the fundamental operation of symbolic model checking [4]. This requires quantifier elimination, which is generally the most computationally expensive aspect of the technique. In [7] a method of approximate image computation is described that is based on interpolation, and does not require quantifier elimination. While the method is over-approximate, it is shown that it can always be made sufficiently precise to prevent false negatives for systems of finite diameter. While [7] treats only the propositional case, the same theory applies to interpolation for first order logic. Thus, in principle the interpolating prover can be used for interpolation-based model checking of infinite-state systems whose transition relation can be expressed in LIUF.

[3] Unfortunately, array updates cannot be handled directly, since the theory of *store* and *select* does not have the Craig interpolation property. Suppose, for example that A is $M' = \mathit{store}(M, a, x)$ and B is $(b \neq c) \wedge (\mathit{select}(M', b) \neq \mathit{select}(M, b)) \wedge (\mathit{select}(M', c) \neq \mathit{select}(M, c))$. The common variables here are M and M', but no facts expressible using only these variables are implied by A (except true), thus there is no interpolant for this pair.

One potential application would be model checking with predicate abstraction. This is a case where the transition relation is expressible in first order logic and the state space is finite, guaranteeing convergence. That is, the state is defined in terms of a set of Boolean variables $v_1 \ldots v_k$ corresponding to the truth values of first-order predicates $p_1 \ldots p_k$. The abstraction relation α is characterized symbolically by the formula $\bigwedge_i v'_i \leftrightarrow p_i$. If the concrete transition relation is characterized by R, the abstract transition relation can be written as the relational composition $\alpha^{-1} \circ R \circ \alpha$. Note that the relational composition can be accomplished by a simple renaming, replacing the "internal" variables with fresh variables that are implicitly existentially quantified. That is, $R \circ S$ can be written as $R\langle U/V' \rangle \wedge S\langle U/V \rangle$ where V and V' are the current and next-state variables respectively, and U is a set of fresh variables. Thus, if the concrete transition relation can be written as a formula in LIUF, then so can the abstract transition relation.

This formula can in turn be rewritten as a satisfiability-equivalent Boolean formula, as is done in [6]. This allows the application of finite-state methods for image computation, but has the disadvantage that it introduces a large number auxiliary boolean variables, making BDD-based image computations impractical. Although SAT-based quantifier elimination techniques are more effective in this case, this approach limits the technique to a small number of predicates. On the other hand, the interpolation-based approach does not require quantifier elimination or translation of the transition relation to a Boolean formula, and thus avoids these problems.

Another possible approach would be to model check the concrete, infinite-state system directly using the interpolation method. For infinite state systems in general this process is not guaranteed to converge. However, in the special case when the model has a finite bisimulation quotient, convergence is guaranteed. This is the case, for example, for timed automata. Since the transition relation of a timed automaton can be expressed in LI, it follows that reachability for timed automata can be verified using the interpolation method. As an example, a model of Fischer's timed mutual exclusion protocol has been verified in this way. Similarly, a simple model of Lamport's "bakery" mutual exclusion, with unbounded ticket numbers, has been modeled and verified. It is possible, in principle to apply the method to software model checking. Convergence is not guaranteed, but of course, this can also be said of predicate abstraction.

5 Conclusions and Future Work

The primary contribution of this work is a method of computing Craig interpolants from refutations in a theory that includes linear inequalities and uninterpreted functions. This extends earlier results that apply only to linear inequalities. This procedure has been integrated with a proof generating decision procedure, combining a SAT solver and a Nelson Oppen style prover to create an interpolating prover.