Zap: Automated Theorem Proving for Software Analysis

Thomas Ball, Shuvendu K. Lahiri, and Madanlal Musuvathi

Microsoft Research
{tball, shuvendu, madanm}@microsoft.com

Abstract. Automated theorem provers (ATPs) are a key component that many software verification and program analysis tools rely on. However, the basic interface provided by ATPs (validity/satisfiability checking of formulas) has changed little over the years. We believe that program analysis clients would benefit greatly if ATPs were to provide a richer set of operations. We describe our desiderata for such an interface to an ATP, the logics (theories) that an ATP for program analysis should support, and present how we have incorporated many of these ideas in Zap, an ATP built at Microsoft Research.

1 Introduction

To make statements about programs in the absence of concrete inputs requires some form of *symbolic reasoning*. For example, suppose we want to prove that the execution of the assignment statement `x:=x+1` from a state in which the formula $(x < 5)$ holds yields a state in which the formula $(x < 10)$ holds. To do so, we need machinery for manipulating and reasoning about formulas that represent sets of program states.

Automated theorem provers (ATPs) provide the machinery that enables such reasoning. Many questions about program behavior can be reduced to questions of the validity or satisfiability of a first-order formula, such as $\forall x : (x < 6) \implies (x < 10)$. For example, given a program P and a specification S, a verification condition $VC(P, S)$ is a formula that is valid if and only if program P satisfies specification S. The validity of $VC(P, S)$ can be determined using an ATP. The basic interface an ATP provides takes as input a formula and returns a Boolean ("Valid", "Invalid") answer. Of course, since the validity problem is undecidable for many logics, an ATP may return "Invalid" for a valid formula.

In addition to this basic interface, ATPs may generate proofs witnessing the validity of input formulas. This basic capability is essential to techniques such as proof-carrying code [Nec97], where the ATP is an untrusted and potentially complicated program and the proof generated by the ATP can be checked efficiently by a simple program.

Through our experience with the use of ATPs in program analysis clients, we often want ATPs to provide a richer interface so as to better support program analysis tasks. We group these tasks into four categories:

G. Sutcliffe and A. Voronkov (Eds.): LPAR 2005, LNAI 3835, pp. 2–22, 2005.

- **Symbolic Fixpoint Computation.** For propositional (Boolean) formulas, binary decision diagrams (BDDs) [Bry86] enable the computation of fixpoints necessary for symbolic reachability and symbolic CTL model checking [BCM+92] of finite state systems. The transition relation of a finite state system can be represented using a BDD, as well as the initial and reachable states of the system. A main advantage of BDDs is that every Boolean function has a normal form, which makes various operations efficient. The basic operations necessary for fixpoint computation are a subsumption test (to test for convergence), quantifier elimination (to eliminate temporary variables used in image computation) and a join operation (to combine formulas representing different sets of states; this is simply disjunction in the case of Boolean logic). We would like to lift these operations to logics that are more expressive than propositional logic, so as to enable the computation of symbolic fixpoints over structures that more closely correspond to the types in programming languages (integers, enumerations, pointers, etc.). While normal forms may not be achievable, simplification of formula is highly desirable to keep formulas small and increase the efficiency of the fixpoint computation.
- **Abstract Transformers.** A fundamental concept in analyzing infinite-state systems (such as programs) is that of abstraction. Often, a system may be converted to a simpler abstract form where certain questions are decidable, such that proofs in the abstract system carry over to proofs in the original system. Abstract interpretation is a framework for mathematically describing program abstractions and their meaning [CC77]. A basic step in the process is the creation of *abstract transformers*: each statement in the original program must be translated to a corresponding abstract statement. This step often is manual. Predicate abstraction is a means for automating the construction of finite-state abstract transformers from infinite-state systems using an ATP [GS97]. ATPs can also be used to create *symbolic best transformers* for other abstract domains [YRS04]. Unfortunately, these approaches suffer from having to make an exponential number of calls to the ATP. If an ATP provides an interface to find all the *consequences* of a set of facts, the process of predicate abstraction and creation of symbolic best transformers can be made more efficient [LBC05]. Consequence finding [Mar99] is a basic operation for the automated creation of abstract transformers that ATPs could support.
- **Property-guided Abstraction Refinement.** If an abstraction is not precise enough to establish the correctness of a program with respect to some property, we wish to find a way to make the abstraction more precise with respect to the property of interest [Kur94, CGJ+00, BR01]. Recently, McMillan showed how interpolants naturally describe how to refine (predicate) abstractions with respect to a property of interest [McM03, HJMM04]. An interpolating ATP [McM04] can support the automated refinement of abstractions.
- **Test Generation.** Finally, we would like to use ATPs to prove the presence of a bug to the user through the automated generation of failure-inducing

inputs [Cla76]. In general, we wish to generate a test input to a program to meet some coverage criteria (such as executing a certain statement or covering a certain control path in the program). To do this, one can create from the program a formula that is satisfiable if and only if there is a test input that achieves the desired coverage criteria. We wish not only to determine the satisfiability of the input formula but also to generate a satisfying assignment that can be transformed into a test input. Model finding/generation is an important capability for ATPs in order to support test generation [ZZ96].

The paper is organized as follows. Section 2 presents more detail about the needs of (symbolic) program clients of ATPs. Section 3 describes the theories/logics that naturally arise from the analysis of programs. We have created an ATP called Zap to meet some of the needs described above. Section 4 gives background material necessary to understand Zap's architecture, which is based on the Nelson-Oppen combination procedure [NO79a, TH96]. We have found that the Nelson-Oppen method can be extended in a variety of ways to support the demands of program analysis clients mentioned above. Section 5 gives an overview of Zap's architecture and describes some of our initial results on efficient decision procedures for fragments of linear arithmetic that occur commonly in program analysis queries. Section 6 describes how we have extended Zap and the Nelson-Oppen combination framework to support richer operations such as interpolation and predicate abstraction. Finally, Section 7 discusses related work.

2 Symbolic Program Analysis Clients of ATPs

This section formalizes the requirements of symbolic program analysis clients of ATPs.

2.1 Notation

A program is a set $\mathcal{C}$ of guarded commands, which are logical formulas c of the form

$$c \;\equiv\; g(X) \wedge x_1' = e_1(X) \wedge \ldots \wedge x_m' = e_m(X)$$

where $X = \{x_1, x_2, \ldots, x_m\}$ are all the program variables. The variable x_i' stands for the value of x_i after the execution of the command. We write $g(X)$ to emphasize that g's free variables come only from X. A program state is a valuation of X. We have a transition of one state into another one if the corresponding valuation of primed and unprimed variables satisfies one of the guarded commands $c \in \mathcal{C}$.

In symbolic evaluation, a formula ϕ represents a set of states, namely, those states in which the formula ϕ evaluates true. Formulas are ordered by implication. We write $\phi \leq \phi'$ to denote that ϕ logically implies ϕ'.

The application of the operator post_c on a formula ϕ is defined as usual; its computation requires a quantifier elimination procedure.

$$\mathsf{post}_c(\varphi) \equiv (\exists X.\ \varphi \wedge g(X) \wedge x_1' = e_1(X) \wedge \ldots \wedge x_m' = e_m(X))[X/X']$$
$$\mathsf{post}(\varphi) \equiv \bigvee_{c \in \mathcal{C}} \mathsf{post}_c(\varphi)$$

In order to specify correctness, we fix formulas init and safe denoting the set of *initial* and *safe* states, respectively. A program is *correct* if no unsafe state is reachable from an initial state. The basic goal of a fixpoint analysis is to find a *safe inductive invariant*, which is a formula ψ such that

$$(\mathsf{init} \leq \psi) \wedge (\mathsf{post}(\psi) \leq \psi) \wedge (\psi \leq \mathsf{safe})$$

The correctness can be proven by showing that $\mathsf{lfp}(\mathsf{post}, \mathsf{init}) \leq \mathsf{safe}$, where $\mathsf{lfp}(\mathcal{F}, \phi)$ stands for the least fixpoint of the operator $\mathcal{F}$ above ϕ.

2.2 Fixpoint Computation

Figure 1 gives a very basic algorithm for (least) fixpoint computation using the post operator. Here we abuse notation somewhat and let ϕ and old be variables ranging over formulas. Initially, ϕ is the formula init and old is the formula `false`. The variable old represents the value of ϕ on the previous iteration of the fixpoint computation. As long as ϕ is not inductive (the test $\phi \leq \mathsf{old}$ fails) then old gets the value of ϕ and ϕ is updated to be the disjunction of current value of ϕ and the value of *post* applied to the current value of ϕ. If ϕ is inductive (the test $\phi \leq \mathsf{old}$ succeeds) then the algorithm tests if ϕ is inside the safe set of states. If so, then the algorithm returns "Correct". Otherwise, it returns "Potential error".

```
φ, old := init, false
loop
  if (φ ≤ old) then
      if (φ ≤ safe) then
          return "Correct"
      else
          return "Potential error"
  else
      old := φ
      φ := φ ∨ post(φ)
endloop
```

Fig. 1. Basic fixpoint algorithm

So, in order to implement a symbolic algorithm using an ATP, we require support for: (1) a subsumption test to test if ϕ is inductive under post ($\leq$); (2) quantifier elimination (to implement post); (3) disjunction of formulas (to collect the set of states represented by ϕ and $\mathsf{post}(\phi)$).

There are a number of interesting issues raised by the symbolic fixpoint client. First, it is well known that certain logics (for example, equality with uninterpreted functions) do not entail quantifier elimination. In these cases, we desire

the ATP to provide a "cover" operation, $\mathsf{cover}(\phi)$, that produces the strongest quantifier-free formula implied by ϕ.

Second, because the lattice of formulas may be infinite, to achieve termination it may be necessary to use an operator other than disjunction to combine the formulas ϕ and $\mathsf{post}(\phi)$. As in abstract interpretation, we desire that logics are equipped with "widening" operators. Given formulas ϕ_i and ϕ_{i+1} such that $\phi_i \leq \phi_{i+1}$, a widening operator widen produces a formula $\psi = \mathsf{widen}(\phi_i, \phi_{i+1})$ such that: (1) $\phi_{i+1} \leq \psi$; (2) the iterated application of widening eventually converges (reaches a fixpoint) [CC77].

The fixpoint algorithm computes a sequence of formulas as follows: $\phi_0 = \mathsf{init}$ and $\phi_{i+1} = \phi_i \vee \mathsf{post}(\phi_i)$. Widening typically is applied to consecutive formulas in this sequence: $\phi_{i+1} = \mathsf{widen}(\phi_i, \phi_i \vee \mathsf{post}(\phi_i))$. The type of widening operator applied may depend on the underlying logic as well as the evolving structure of formulas in the fixpoint sequence. An example of widening over the integer domains would be to identify a variable with an increasing value and widen to an open interval: $\mathsf{widen}(i = 1, i = 1 \vee i = 2) = i \geq 1$.

2.3 Finitary Abstract Transformers

As we have seen in the previous section, the symbolic fixpoint computation can diverge because the lattice of formulas may have infinite ascending chains. Widening is one approach to deal with the problem. Another approach is to a priori restrict the class of formulas under consideration so as to guarantee termination of the fixpoint computation.

For example, suppose we restrict the class of formulas we can assign to the variables ϕ and old in the fixpoint computation to be propositional formulas over a set P of finite atomic predicates. Let us denote this class of formulas by F_P. In this case, the number of semantically distinct formulas is finite.

However, there is a problem: this class of formulas is not closed under post (nor under pre, the backwards symbolic transformer, for that matter). Suppose that we have $\phi \in F_P$ and that $\mathsf{post}(\phi) \notin F_P$. We again require a cover operation $\mathsf{cover}_P(\phi)$ of the ATP, that produces the strongest formula in F_P implied by ϕ. Then, we modify the fixpoint computation by changing the assignment statement to variable ϕ to:

$$\phi := \phi \vee \mathsf{cover}_P(\mathsf{post}(\phi))$$

Note that cover_P is not the same operation as the cover operation from the previous section. cover_P is parameterized by a set of predicates P while the cover operation has no such restriction. The cover_P operation is the basic operation required for predicate abstraction [GS97].

The cover_P operation is related to the problem of consequence finding [Mar99]. Given a set of predicates P, the goal of consequence finding is to find all consequences of P. The $\mathsf{cover}_P(\phi)$ operation expresses all consequences of P that are implied by ϕ. As described later, we have shown that is possible to compute cover_P efficiently for suitably restricted theories [LBC05].

2.4 Abstraction Refinement

In the presence of abstraction, it often will be the case that the fixpoint computation will return "Potential error", even for correct programs. In such cases, we would like to refine the abstraction to eliminate the "potential errors" and guide the fixpoint computation towards a proof. In the case of predicate abstraction, this means adding predicates to the set P that defines the finite state space. Where should these new predicates come from?

Let us again consider the sequence of formulas computed by the abstract symbolic fixpoint: $\phi_0 = \mathsf{init}$; $\phi_{i+1} = \phi_i \vee \mathsf{cover}_P(\mathsf{post}(\phi_i))$. Suppose that ϕ_k is inductive (with respect to post) but does not imply safe. Now, consider the following sequence of formulas: $\psi_0 = \mathsf{init}$; $\psi_{i+1} = \mathsf{post}(\psi_i)$. If the program is correct then the formula $\psi_k \wedge \neg\mathsf{safe}$ is unsatisfiable. The problem is that the set of predicates P is not sufficient for the abstract symbolic fixpoint to prove this. One approach to address this problem would be to take the set of (atomic) predicates in all the ψ_j $(0 \leq j \leq k)$ and add them to P. However, this set may contain many predicates that are not useful to proving that $\psi_k \wedge \neg\mathsf{safe}$ is unsatisfiable.

Henzinger et al. [HJMM04] showed how Craig interpolants can be used to discover a more precise set of predicates that "explains" the unsatisfiability. Given formulas A and B such that $A \wedge B = \mathtt{false}$, an interpolant $\Theta(A, B)$ satisfies the three following points:

- $A \Rightarrow \Theta(A, B)$,
- $\Theta(A, B) \wedge B = \mathtt{false}$,
- $V(\Theta(A, B)) \subseteq V(A) \cap V(B)$

That is, $\Theta(A, B)$ is weaker than A, the conjunction of $\Theta(A, B)$ and B is unsatisfiable ($\Theta(A, B)$ is not too weak), and all the variables in $\Theta(A, B)$ are common to both A and B.

Let us divide the formula $\psi_k \wedge \neg\mathsf{safe}$ into two parts: a *prefix* $A_j = \mathsf{post}^j(\mathsf{init})$ and a *suffix* $B_j = \mathsf{post}^{k-j} \wedge \neg\mathsf{safe}$, where $0 \leq j \leq k$ and post^i denotes the i-fold composition of the post operator (recall that post is itself a formula).[1]

An interpolant $Q_j = \Theta(A_j, B_j)$ yields a set of predicates $\mathsf{p}(Q_j)$ such that $\mathsf{cover}_{\mathsf{p}(Q_j)}(A_j) \wedge B_j$ is unsatisfiable. This is because $A_j \Rightarrow Q_j$ and $Q_j \wedge B_j = \mathtt{false}$ (by the definition of interpolant) and because $A_j \Rightarrow \mathsf{cover}_{\mathsf{p}(Q_j)}(A_j)$ and $\mathsf{cover}_{\mathsf{p}(Q_j)}(A_j)$ is at least as strong as Q_j (by the definition of cover).

Thus, the union $Q = \bigcup_{j \in \{1,\cdots k\}} \mathsf{p}(Q_j)$ is sufficient for the abstract symbolic fixpoint to prove that it is not possible to reach an unsafe state (a state satisfying $\neg\mathsf{safe}$) in k steps.

2.5 Test Generation

We also would like to use ATPs to prove the *presence* of errors as well as their absence. Thus, it makes sense for ATPs to return three-valued results for validity/satisfiability queries: "yes", "no" and "don't know". Of course, because

[1] Note that $\psi_k = \mathsf{post}^k(\mathsf{init})$.

of undecidability, we cannot always hope for only "yes" or "no" answers. However, even for undecidable questions, it is more useful to separate out "no" from "don't know" when possible, rather than lumping the two together (as is usually done in program analysis as well as automated theorem proving). Much research has been done in using three-valued logics in program analysis model checking [SRW99, SG04].

The ultimate "proof" to a user of a program analysis tool that the tool has found a real error in their program is for the tool to produce a concrete input on which the user can run their program to check that the tool has indeed found an error. Thus, just as proof-carrying code tools produce proofs that are relatively simple to check, we would like defect-detection tools to produce concrete inputs that can be checked simply by running the target program on them. Thus, we desire ATPs to produce models when they find that a formula is satisfiable, as SAT solvers do. We will talk about the difficulty of model production later.

2.6 Microsoft Research Tools

At Microsoft Research, there are three main clients of the Zap ATP: *Boogie*, a static program verifier for the C# language [BLS05]; *MUTT*, a set of testing tools for generating test inputs for MSIL, the bytecode language of Microsoft's .Net framework [TS05]; and *Zing*, a model checker for concurrent object-oriented programs (written in the Zing modeling language) [AQRX04]. In the following sections, we describe each of the clients and their requirements on the Zap ATP.

Boogie. The Boogie static program verifier takes as input a program written in the Spec# language, a superset of C# that provides support for method specifications like pre- and postconditions as well as object invariants. The Boogie verifier then infers loop invariants using interprocedural abstract interpretation. The loop invariants are used to summarize the effects of loops. In the end, Boogie produces a verification condition that is fed to an ATP.

MUTT. MUTT uses a basic approach [Cla76] to white-box test generation for programs: it chooses a control-flow path p through a program P and creates a formula $F(P, p)$ that is satisfiable if and only if there is an input I such that execution of program P on input I traverses path p. A symbolic interpreter for MSIL traverses the bytecode representation of a program, creating a symbolic representation of the program's state along a control-flow path. At each (binary) decision point in the program, the interpreter uses the ATP to determine whether the current symbolic state constrains the direction of the decision. If it does not, both decision directions are tried (using backtracking) and appropriate constraints added to the symbolic state for each decision. This client generates formulas with few disjuncts. Furthermore, the series of formulas presented to the ATP are very similar. Thus, an ATP that accepts the incremental addition/deletion of constraints is desired. Finally, when a formula is satisfiable, the ATP should produce a satisfying model.

Zing. Zing is an explicit state model checker for concurrent programs written in an objected-oriented language that is similar to C#. Zing implements various optimizations such as partial-order reduction, heap canonicalization and procedure-level summarization. Recently, researchers at Microsoft have started to experiment with hybrid state representations, where some parts of the state (the heap) are represented explicitly and other parts (integers) are represented symbolically with constraints. Zing uses the Zap ATP to represent integer constraints and to perform the quantifier elimination required for fixpoint computation.

3 Theories for Program Analysis

Various program analyses involve reasoning about formulas whose structure is determined both by the syntax of the programs and the various invariants that the analyses require. This section identifies those logics that naturally arise when analyzing programs and thus should be supported by the ATP. We provide an informal description of these logics and emphasize those aspects that are particularly important for the clients of Zap. The reader should read [DNS03] for a more detailed description.

We restrict the discussion to specific fragments of first-order logic with equality. While we have not explored the effective support for higher order logics in Zap, such logics can be very useful in specifying certain properties of programs [GM93, ORS92, MS01, IRR+04]. For instance, extending first-order logic with transitive closure [IRR+04] enables one to specify interesting properties about the heap.

The control and data flow in most programs involve operations on integer values. Accordingly, formulas generated by program analysis tools have a preponderance of integer arithmetic operations. This makes it imperative for the ATP to have effective support for integers. In practice, these formulas are mostly linear with many *difference* constraints of the form $x \leq y + c$. While multiplication between variables is rarely used in programs, it is quite common for loop invariants to involve non-linear terms. Thus, some reasonably complete support for multiplication is desirable.

As integer variables in programs are implemented using finite-length bit vectors in the underlying hardware, the semantics of the operations on these variables differs slightly from the semantics of (unbounded) integers. These differences can result in integer-overflow related behavior that is very hard to reason about manually. An ATP that allows reasoning about these bounded integers, either by treating them as bit-vectors or by performing modular arithmetic, can enable analysis tools that detect overflow errors. In addition, the finite-implementation of integers in programs becomes apparent when the program performs bit operations. It is a challenging problem for the ATP to treat a variable as a bit-vector in such rare cases but still treat it as an integer in the common case.

Apart from integer variables, programs define and use derived types such as structures and arrays. Also, programs use various *collection* classes which can

be abstractly considered as maps or sets. It is desirable for the ATP to have support for theories that model these derived types and data structures.

Another very useful theory for program analysis is the theory of partial orders. The inheritance hierarchy in an object oriented program can be modeled using partial orders. The relevant queries involve determining if a particular type is a minimum element (base type) or a maximal element (final type), if one type is an ancestor (derived class) of another, and if two types are not ordered by the partial-order.

While the formulas generated during program analysis are mostly quantifier-free, invariants on arrays and collection data structures typically involve quantified statements. For instance, a tool might want to prove that all elements in an array are initialized to zero. Accordingly, the underlying ATP should be able to reason about quantified facts. In addition, supporting quantifiers in an ATP provides the flexibility for a client to encode domain-specific theories as axioms.

4 Background

In this section, we briefly describe the notations, the syntax and semantics of the logic, and a high-level description of the Nelson-Oppen combination algorithm for decision procedures. Our presentation of theories and the details of the algorithm is a little informal; interested readers are referred to excellent survey works [NO79a, TH96] for rigorous treatment.

4.1 Preliminaries

Figure 2 defines the syntax of a quantifier-free fragment of first-order logic. An expression in the logic can either be a *term* or a *formula*. A *term* can either be a variable or an application of a function symbol to a list of terms. A *formula* can be the constants `true` or `false` or an atomic formula or Boolean combination of other formulas. Atomic formulas can be formed by an equality between terms or by an application of a predicate symbol to a list of terms. A *literal* is either an atomic formula or its negation. A *monome* m is a conjunction of literals. We will often identify a conjunction of literals $l_1 \wedge l_2 \ldots l_k$ with the set $\{l_1, \ldots, l_k\}$.

The function and predicate symbols can either be *uninterpreted* or can be defined by a particular theory. For instance, the theory of integer linear arithmetic defines the function-symbol "+" to be the addition function over integers and "<" to be the comparison predicate over integers. For a theory T, the *signature* Σ denotes the set of function and predicate symbols in the theory. If an

$$
\begin{aligned}
\textit{term} &::= \textit{variable} \mid \textit{function-symbol}(\textit{term}, \ldots, \textit{term}) \\
\textit{formula} &::= \texttt{true} \mid \texttt{false} \mid \textit{atomic-formula} \\
&\quad \mid \textit{formula} \wedge \textit{formula} \mid \textit{formula} \vee \textit{formula} \mid \neg \textit{formula} \\
\textit{atomic-formula} &::= \textit{term} = \textit{term} \mid \textit{predicate-symbol}(\textit{term}, \ldots, \textit{term})
\end{aligned}
$$

Fig. 2. Syntax of a quantifier-free fragment of first-order logic

expression E involves function or predicate symbols from two (or more) theories T_1 and T_2, then E is said to be an expression over a combination of theories $T_1 \cup T_2$.

An *interpretation* $\mathcal{M} = \langle \mathcal{D}, \mathcal{J} \rangle$ consists of (i) a *domain* $\mathcal{D}$ and a (ii) mapping $\mathcal{J}$ from each function (or predicate) symbol in the theory to a function (or relation) over the domain $\mathcal{D}$. A formula ϕ is said to be *satisfiable* if there exists an interpretation $\mathcal{M}$ and an assignment ρ to the variables such that ϕ evaluates to `true` under $(\mathcal{M}, \rho)$. Such an interpretation is called a *model* of ϕ. A formula is *valid* if $\neg\phi$ is not satisfiable (or unsatisfiable). A satisfiability (or decision) procedure for Σ-theory T checks if a formula ϕ (over Σ) is satisfiable in T.

A theory T is *convex* if a quantifier-free formula ϕ in the T implies a disjunction of equalities over variables $x_1 = y_1 \vee x_2 = y_2 \ldots x_k = y_k$ if and only if ϕ implies at least one of the equalities $x_i = y_i$. A theory T is *stably-infinite* if a quantifier-free formula ϕ has a model if and only if ϕ has an infinite model, i.e., the domain of the model is infinite. Example of both convex and stably-infinite theories include the logic of Equality with Uninterpreted Functions (EUF) and rational linear arithmetic [NO79a]. Example of non-convex theories include the theory of *arrays* and the theory of integer linear arithmetic.

4.2 Nelson Oppen Combination

Given two *stably infinite* theories T_1 and T_2 with disjoint-signatures Σ_1 and Σ_2 respectively (i.e. $\Sigma_1 \cap \Sigma_2 = \{\}$), and a conjunction of literals ϕ over $\Sigma_1 \cup \Sigma_2$, we want to decide if ϕ is satisfiable under $T_1 \cup T_2$. Nelson and Oppen [NO79a] provided a method for modularly combining the satisfiability procedures for T_1 and T_2 to produce a satisfiability procedure for $T_1 \cup T_2$.

We describe the Nelson-Oppen procedure for convex theories.[2] The input ϕ is split into the formulas ϕ_1 and ϕ_2 such that ϕ_i only contains symbols from Σ_i and $\phi_1 \wedge \phi_2$ is equisatisfiable with ϕ. Each theory T_i decides the satisfiability of ϕ_i and returns unsatisfiable if ϕ_i is unsatisfiable in T_i. Otherwise, the set of equalities implied by ϕ_i over the variables common to ϕ_1 and ϕ_2 are propagated to the other theory T_j. The theory T_j adds these equalities to ϕ_j and the process repeats until the set of equalities saturate.

Therefore, in addition to checking the satisfiability of a set of literals, each theory also has to derive all the equalities over variables that are implied by the set of literals. The satisfiability procedure is called *equality generating* if it can generate all such equalities.

5 Zap

In this section, we start by describing the basic theorem proving architecture in Zap in Section 5.1. In Section 5.2, we present improvements to decision procedures for a restricted fragment of linear arithmetic that constitute most program analysis queries. In Section 5.3, we describe the handling of quantifiers in first-order formulas.

[2] For the description of the algorithm for non-convex case, refer to the [NO79a].

5.1 Basic Architecture

In this section, let us assume that we are checking the satisfiability of a quantifier-free first-order formula ϕ over theories $T_1, \ldots, T_k$. The basic architecture of Zap is based on a lazy proof-explicating architecture for deciding first-order formulas [ABC$^+$02, BDS02, FJOS03].

First, a Boolean abstraction of ϕ is generated by treating each atomic formula e in ϕ as an uninterpreted Boolean variable. The abstract formula is checked using a Boolean SAT solver. If the SAT solver determines that the formula is unsatisfiable, then the procedure returns unsatisfiable. Otherwise, the satisfying assignment from SAT (a monome m over the literals in ϕ) is checked for satisfiability using the Nelson-Oppen decision procedure for the combined theory $T_1 \cup \ldots \cup T_k$, as described in the last section. If ϕ is satisfiable over the first-order theories, the formula ψ is satisfiable and the procedure returns satisfiable. Otherwise, a "conflict clause" is derived from the theories that will prevent the same assignment being produced by the SAT solver. The process repeats until the Boolean SAT solver returns unsatisfiable or the Nelson-Oppen procedure returns satisfiable.

To generate a conflict clause, the decision procedure for the combined theories generates a proof of unsatisfiability when the monome m is unsatisfiable over $T_1 \cup \ldots \cup T_k$. The literals that appear in the proof constitute a conflict clause. In this framework, each theory generates the proof of (i) unsatisfiability of a monome in the theory and (ii) proof of every equality $x = y$ over the shared variables that are implied by the literals in the theory.

We use SharpSAT, a variant of the ZChaff [MMZ$^+$01] Boolean SAT solver developed at Microsoft by Lintao Zhang, as the underlying Boolean solver. In addition to checking satisfiability of a Boolean formula, SharpSAT also generates proof of unsatisfiability for unsatisfiable formulas. The theories present in Zap are the logic of equality with uninterpreted functions (EUF) and linear arithmetic. The decision procedure for EUF is based on the congruence closure algorithm [NO80]. For linear arithmetic, we have an implementation of proof-generating variant of the Simplex algorithm described in the Simplify technical report [DNS03]. We also have a decision procedure for Unit Two Variable Per Inequality (UTVPI) subset of linear arithmetic.

5.2 Restricted Linear Arithmetic Decision Procedures

Pratt [Pra77] observed that the arithmetic component in most program verification queries is mostly restricted to the difference logic ($x \leq y + c$) fragment. Recent studies by Seshia and Bryant [SB04] also indicate that more than 90% of the arithmetic constraints in some program analysis benchmarks are in difference logic fragment. We have also observed that structure of the constraints is *sparse*, i.e., if n is the number of variables in the queries, and m is the number of arithmetic constraints, then m is typically $O(n)$. In this section, we present our first step to obtain efficient decision procedure that exploit these observations.

The Unit Two Variable Per Inequality (UTVPI) logic is the fragment of integer linear arithmetic, where constraints are of the form $a.x + b.y \leq c$, where

a and b are restricted to $\{-1, 0, 1\}$ and c is an integer constant. This fragment (a generalization of difference constraints) is attractive because this is the most general class (currently known) of integer linear arithmetic for which the decision procedure enjoys a polynomial complexity [JMSY94]. Extending this fragment to contain three variables (with just unit coefficients) per inequality or adding non-unit coefficients for two variable inequalities make the decision problem NP-complete [Lag85]. Having an integer solver is often useful when dealing with variables for which rational solutions are unacceptable. Such examples often arise when modeling indices of an array or queues [FLL+02].

In [LM05a], we present an efficient decision procedure for UTVPI constraints. Our algorithm works by reducing the UTVPI constraints to a set of difference constraints, and then using negative cycle detection algorithms [CG96] to solve the resultant problem. Given m such constraints over n variables, the procedure checks the satisfiability of the constraints in $O(n.m)$ time and $O(n + m)$ space. This improves upon the previously known $O(n^2.m)$ time and $O(n^2)$ space algorithm provided by Jaffar et al. [JMSY94] based on transitive closure. The space improvement of our algorithm is particularly evident when m is $O(n)$, which occurs very frequently in practice, as the number of constraints that arise in typical verification queries have a sparse structure.

In addition to checking satisfiability of a set of UTVPI constraints, the decision procedure is also equality generating and proof producing. These requirements are in place because the decision procedure participates in the proof-explicating ATP described earlier. The decision procedure generates equalities between variables implied by a set of UTVPI constraints in $O(n.m)$ time. The algorithm can generate a proof of unsatisfiability and equalities implied in $O(n.m)$ time. Both these algorithms use linear $O(n+m)$ space. The algorithm for UTVPI generalizes our earlier results for difference logic fragment [LM05b].

We also provide a model generation algorithm for rational difference logic constraints (i.e. the variables are interpreted over rationals) in [LM05b]. The highlight of the algorithm is that the complexity of generating the model places a *linear* time and space overhead over the satisfiability checking algorithm. We also provide a model generation algorithm for integer UTVPI constraints in [LM05a]. The algorithm is currently based on transitive closure and runs in $O(n.m + n^2.logn)$ time and $O(n^2)$ space.

For many program analysis queries, having a UTVPI decision procedure suffice — more complex linear constraints often simplify to UTVPI constraints after propagating equalities and constants. We are also working on integrating the decision procedure UTVPI constraints within a general linear arithmetic decision procedure. This will enable us to exploit the efficient decision procedures for UTVPI even in the presence of (hopefully a few) general linear arithmetic constraints.

5.3 Quantifiers

Quantified statements naturally arise when analyzing invariants over arrays and data structures such as maps. To handle such quantified formulas, Zap uses an

approach very similar to Simplify based on heuristic instantiations. When the theories (that reason on quantifier-free literals in the formula) are not able to detect unsatisfiability, Zap uses various heuristics to instantiate quantified formulas with relevant terms from the input formula. Zap propagates the resulting quantifier-free formulas to the theories, which in turn try to detect unsatisfiability. This instantiation process continues for a few iterations (if necessary) after which Zap returns stating its inability to prove the unsatisfiability of the formula.

One challenge in supporting this instantiation based approach in a lazy proof explication setting is the following. Quantified formulas typically involve propositional connectives. As a result, quantifier instantiations performed during theory reasoning can dynamically introduce Boolean structure in the formula. This directly conflicts with the requirement that the Boolean structure be exposed statically to the SAT solver in a lazy proof explication setting. Moreover, most quantifier instantiations are not useful in proving the validity of the formula. Blindly exposing such redundant instantiations to the SAT solver could drastically reduce the performance of the propositional search.

To alleviate these problems, Zap implements a *two-tier* technique [LMO05] for supporting quantifiers. This technique involves two instances of a SAT solver, a *main* solver that performs the propositional reasoning of the input formula, and a *little* solver that reasons over the quantifier instantiations. When the main SAT solver produces a propositionally satisfying instance that is consistent with the decision procedures, the heuristic instantiation process generates a set of new facts that The little SAT solver, along with the decision procedures, tries to falsify the satisfying instance with the instantiations produced. If successful, the little SAT solver generates a blocking clause that only contains literals from the input formula. By thus separating the propositional reasoning of the input formula from that of the instantiated formulas, this technique reduces the propositional search space, with an eye toward improving performance.

In practice, we have found that our implementation is limited by the heuristics we use to instantiate quantifiers. These heuristics rely heavily on the "patterns" that the user specifies with each quantified statement. The performance of the ATP changes significantly even for slight changes to these patterns requiring several iterations to get them right. Moreover, we have found that it takes considerable effort to automate the process of generating these patterns. Ideally, we could use general purpose resolution-based ATPs (such as Vampire [RA01]) that are optimized to reason about quantified statements. However, these ATPs do not effectively support arithmetic reasoning, an important requirement for Zap. Combining a decision procedure for integers with a resolution-based ATP is a challenging open problem. Such an ATP would be very useful in our setting.

6 Richer Operations and Their Combinations

As described in Section 2, the main goal of Zap is to support a rich set of symbolic operations, apart from validity checking. These operations, such as

quantifier elimination and model generation are essential to support symbolic computation in Zap's clients. On the other hand, Zap needs to support a variety of theories (Section 3) that are useful for program analysis. Supporting these symbolic operations in the presence of multiple theories leads to an interesting challenge of *combining* these operations across theories.

Specifically, we seek a generalization of the Nelson-Oppen combination for decision procedures as follows. For a particular symbolic operation, assume that there exists a theory-specific procedure that performs the operation for formulas in that theory. Now, given such procedures for two different theories, the *combination* problem is to devise a procedure that performs the symbolic operation on formulas in the combination of the two theories, using the two theory-specific procedures as black boxes. When the symbolic operation in question is that of determining the satisfiability of a set of formulas then the general combination problem reduces to the well studied combination of decision procedures.

Such a combination procedure for supporting symbolic operations has several advantages over a monolithic procedure for a specific combination of theories. First, the combination approach provides the flexibility of adding more theories in the future. This is very important for Zap as enabling new applications might require supporting new theories. Second, the combination approach allows each theory-specific decision procedure to be independently designed, implemented and proven correct. The combination method itself needs to be proven once. The correctness of a particular combination directly follows from the correctness of each individual theory-specific procedure and the correctness of the combination method. Finally, the combination approach leads to a modular implementation of Zap that greatly simplifies the correctness of the implementation.

In the following sections, we describe how we extended the equality propagation framework of Nelson-Oppen combination to modularly combine procedures for interpolant-generation (in Section 6.1), and predicate abstraction (in Section 6.2). In Section 6.3, we present difficulties in modularly combining model generation procedures. The combination methods for other symbolic operations still remains open.

6.1 Interpolants

In [YM05], we presented a novel combination method for generating interpolants for a class of first-order theories. Using interpolant-generation procedures for component theories as black-boxes, this method generates interpolants for formulas in the combined theory. Provided the individual procedures for the component theories can generate interpolants in polynomial time, our method generates interpolants for the combined theory in polynomial time.

The combination method uses the fact that the proof of unsatisfiability produced in a Nelson-Oppen combination has a particular structure. In the Nelson-Oppen framework, the decision procedures for component theories communicate by propagating entailed equalities. Accordingly, the proof can be split into theory-specific portions that only involve inference rules from that theory. These theory-specific portions use literals from the input or equalities generated

by other theories. The crucial idea behind the combination method is to associate a *partial interpolant* with each propagated equality. Whenever a component theory propagates an equality, the combination method uses the interpolant-generation procedure for that theory to generate the partial interpolant for the equality. When a theory detects a contradiction, the combination method uses the partial interpolants of all propagated equalities to compute the interpolant for the input formulas.

The combination method places some restrictions on the theories that can be combined. The Nelson-Oppen combination requires that the component theories have disjoint signatures and be *stably-infinite* [NO79b, Opp80]. Our method naturally inherits these restrictions. Additionally, our combination method restricts the form of equalities that can be shared by the component theories. Specifically, the method requires that each propagated equality only involve symbols common to both input formulas A and B. We show that this restricted form of equality propagation is sufficient for a class of theories, which we characterize as *equality-interpolating* theories. Many useful theories including the quantifier-free theories of uninterpreted functions, linear arithmetic, and Lisp structures are equality-interpolating, and thus can be combined with our method.

While the restriction to equality-interpolating theories provides us a way to extend the existing Nelson-Oppen combination framework, the problem of generalizing the combination result to other theories remains open. Moreover, while our method generates *an* interpolant between two formulas, it is not clear if the interpolant generated is useful for the program analysis in question. Accordingly, we need to formalize the notion of *usefulness* of an interpolant to a particular analysis and design an algorithm that finds such interpolants.

6.2 Predicate Abstraction

Given a formula ϕ and a set of predicates P, the fundamental operation in predicate abstraction is to find the best approximation of ϕ using P. Let $\mathcal{F}_P(\phi)$ be the weakest expression obtained by a Boolean combination of the predicates that implies ϕ.[3]

In [LBC05], we describe a new technique for computing $\mathcal{F}_P(\phi)$ without using decision procedures, and provide a framework for computing $\mathcal{F}_P(\phi)$ for a combination of theories. We present a brief description of the approach in this section.

For simplicity, let us assume that ϕ is an atomic expression (for more general treatment, refer to [LBC05]). To compute $\mathcal{F}_P(\phi)$, we define a *symbolic decision procedure* (*SDP*) for a theory to be a procedure that takes as input a set of atomic expressions G and an atomic expression e and returns a symbolic *representation* of all the subsets $G' \subseteq G$ such that $G' \wedge \neg e$ is unsatisfiable. $SDP(G, e)$ symbolically simulates the execution of a decision procedure on every subset $G' \subseteq G$. Let $\tilde{P}$ be the set of negated predicates in P. If the formula ϕ and the predicates P belong to a theory T, then $SDP(P \cup \tilde{P}, \phi)$ represents $\mathcal{F}_P(\phi)$.

[3] Note that $\mathcal{F}_P(\phi)$ is the dual of $\mathsf{cover}_P(\phi)$ introduced earlier. It is easy to see that $\mathsf{cover}_P(\phi) = \neg\mathcal{F}_P(\neg\phi)$.

We present an algorithm for constructing *SDP* for a class of theories called *saturating theories*. For a theory T, consider the following procedure that repeatedly derives new facts from existing facts by applying the inference rules of the theory on the existing set of facts — Given a set of atomic expressions $H_0 \doteq H$, let $H_0, H_1, \ldots, H_i, \ldots$ denote a sequence of sets of atomic expressions in T such that H_{i+1} is the set of atomic expressions either present in H_i or derived from H_i using inference rules in the theory. A theory is saturating, if (i) each of the sets H_i is finite and, (ii) if H is inconsistent, then `false` is present in H_{k_H}, where k_H is a finite value that is a function of the expressions in H alone. For such a saturating theory one can construct $SDP(G, e)$ by additionally maintaining the derivation history for each expression in any H_i. The derivation history can be maintained as a directed acyclic graph, with leaves corresponding to the facts in H. Finally, the expression for $SDP(G, e)$ can be obtained by performing the above procedure for $H \doteq G \cup \{\neg e\}$ and returning (all) the derivations of `false` after k_H steps.

For two saturating theories T_1 and T_2 with disjoint signatures that also are convex and stably-infinite, we present a procedure for constructing *SDP* for the combined theory $T_1 \cup T_2$, by extending the Nelson-Oppen framework. Intuitively, we symbolically encode the operation of the Nelson-Oppen equality sharing framework for any possible input for the two theories. The *SDP* for the combined theory incurs a polynomial blowup over the *SDP* for either theory. For many theories that are relevant to program analysis, *SDP* can be computed in polynomial or pseudo-polynomial time and space complexity. Examples of such theories include EUF and difference logic (DIF). The combination procedure allows us to construct an *SDP* from these theories' *SDP*s.

We have implemented and benchmarked our technique on a set of program analysis queries derived from device driver verification [BMMR01]. Preliminary results are encouraging and the new predicate abstraction procedure outperformed decision procedure based predicate abstraction methods by orders of magnitude [LBC05]. It remains open how to extend this approach in the presence of more complex (non-convex) theories or quantifiers.

6.3 Model Generation

When Zap reports that a first-order formula ϕ is satisfiable, it is desirable to find a model for ϕ. Apart from serving as a witness to the satisfiability of ϕ, the model generated is very useful for generating test cases from symbolic execution of software. In this section, we present some of the issues in combining model generation for different theories.

To generate an assignment for the variables that are shared across two theories, each theory T_i needs to ensure that the variable assignment ρ for T_i assigns two shared variables x and y equal values if and only if the equality $x = y$ is implied by the constraints in theory T_i. Such a model where $\rho(x) = \rho(y)$ if and only if $T_i \cup \phi_i$ implies $x = y$, is called a *diverse* model. We have shown that for even (integer) difference logic, producing *diverse* models is NP-complete [LM05a].

The following example shows why diverse model generation is required in the Nelson-Oppen framework. Let $\phi = (f(x) \neq f(y) \wedge x \leq y)$ be a formula in the combined theory of EUF and UTVPI. An ATP based on the Nelson-Oppen framework will add $\phi_1 \doteq f(x) \neq f(y)$ to the EUF theory (T_1) and $\phi_2 \doteq x \leq y$ to the UTVPI theory (T_2). Since there are no equalities implied by either theory, and each theory T_i is consistent with ϕ_i, the formula ϕ is satisfiable. Now, the UTVPI theory could generate the model $\rho \doteq \langle x \mapsto 0, y \mapsto 0 \rangle$ for ϕ_2. However, this is not a model for ϕ, as it is not diverse.

Presently, Zap uses various heuristics for generating a model consistent with all the theories. As a last resort, we perform an equisatisfiable translation of ϕ to a Boolean formula using an *eager* encoding of first-order formulas [LS04, SB04] and use the SAT solver to search for a model[4].

7 Related Work

In this section, we describe some prior work on theorem proving and symbolic reasoning for program analysis.

Simplify [DNS03] is an ATP that was built to discharge verification conditions (VCs) in various program analysis projects including ESC/JAVA [FLL+02]. It supports many of the theories discussed in this paper along with quantifiers. It is based on the Nelson-Oppen framework for combining decision procedures. Apart from validity checking, Simplify allows for error localization by allowing the verification conditions to contain *labels* from the program. These labels help to localize the source locations and the type of errors when the validity check of a VC fails.

McMillan [McM04] presents an interpolating ATP for the theories of EUF and linear inequalities (and their combination). This ATP has been used in abstraction refinement for the BLAST [HJMM04] software model checker. Our work on combining interpolants for various equality-interpolating theories generalizes McMillan's work, and extends it to other theories. Lahiri et al. [LBC03] present an algorithm for performing predicate abstraction for a combination of various first-order theories by performing Boolean quantifier elimination. Unlike their approach, the use of symbolic decision procedures in our case allows us to perform predicate abstraction in a modular fashion.

Gulwani et al. [GTN04] present join algorithms for subclasses of EUF using *abstract congruence closure* [BTV03]. They show the completeness of he algorithm for several subclasses including the cases when the functions are *injective*. Chang and Leino [CL05] provide an algorithm for performing abstract operations (e.g. join, widen etc.) for a given *base* domain (e.g. the polyhedra domain for linear inequalities [CH78]) in the presence of symbols that do not belong to the theory. Their framework introduces names for each *alien* expression in the theory. A congruence closure abstract domain equipped with the abstract operations (join, widen etc.) is used to reason about the mapping of the names to

[4] Krishna K. Mehra implemented part of the model generation algorithm in Zap when he spent the summer in Microsoft.

the alien expressions. They instantiate the framework for the polyhedra domain and a domain for reasoning about heap updates.

There has also been a renewed interest in constructing decision procedures for first-order theories by exploiting SAT solver's backtracking search. Decision procedures based on lazy proof explicating framework (e.g. CVC [BDS02], Verifun [FJOS03], Mathsat [ABC+02]), eager approaches (e.g. UCLID [BLS02]), extending DPLL search to incorporate theory reasoning [GHN+04] have been proposed to exploit rapid advances in SAT solvers. Although Zap is closest to the lazy approaches in its architectures, we are also investigating the best match of these approaches for the nature of queries generated by the various clients of program analysis.

References

[ABC+02] G. Audemard, P. Bertoli, A. Cimatti, A. Korniłowicz, and R. Sebastiani. A SAT-based approach for solving formulas over Boolean and linear mathematical propositions. In *CADE 02: Conference on Automated Deduction*, LNCS 2392, pages 195–210. Springer-Verlag, 2002.

[AQRX04] T. Andrews, S. Qadeer, S. K. Rajamani, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In *CONCUR 04: Concurrency Theory*, LNCS 3170, pages 1–15. Springer-Verlag, 2004.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BDS02] C. W. Barrett, D. L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *CAV 02: Computer-Aided Verification*, LNCS 2404, pages 236–249. Springer-Verlag, 2002.

[BLS02] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *CAV 02: Computer-Aided Verification*, LNCS 2404, pages 78–92. Springer-Verlag, 2002.

[BLS05] M. Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 04: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, LNCS 3362, pages 49–69. Springer-Verlag, 2005.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*, pages 203–213. ACM, 2001.

[BR01] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 01: SPIN Workshop*, LNCS 2057, pages 103–122. Springer-Verlag, 2001.

[Bry86] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[BTV03] Leo Bachmair, Ashish Tiwari, and Laurent Vigneron. Abstract congruence closure. *J. Autom. Reasoning*, 31(2):129–168, 2003.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *POPL 77: Principles of Programming Languages*, pages 238–252. ACM, 1977.

[CG96] B. V. Cherkassky and A. V. Goldberg. Negative-cycle detection algorithms. In *European Symposium on Algorithms*, pages 349–363, 1996.

[CGJ+00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV 00: Computer Aided Verification*, LNCS 1855, pages 154–169. Springer-Verlag, 2000.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL 78: Principles of Programming Languages*, pages 84–96. ACM, 1978.

[CL05] B-Y. E. Chang and K. R. M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI 05: Verification, Model Checking, and Abstract Interpretation*, LNCS 3385, pages 147–163. Springer-Verlag, 2005.

[Cla76] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, September 1976.

[DNS03] D. L. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical report, HPL-2003-148, 2003.

[FJOS03] C. Flanagan, R. Joshi, X. Ou, and J. Saxe. Theorem proving using lazy proof explication. In *CAV 03: Computer-Aided Verification*, LNCS 2725, pages 355–367. Springer-Verlag, 2003.

[FLL+02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI 02: Programming Language Design and Implementation*, pages 234–245. ACM, 2002.

[GHN+04] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *CAV 04: Computer Aided Verification*, LNCS 3114, pages 175–188. Springer-Verlag, 2004.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *CAV 97: Computer Aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.

[GTN04] S. Gulwani, A. Tiwari, and G. C. Necula. Join algorithms for the theory of uninterpreted functions. In *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science*, LNCS 3328, pages 311–323. Springer, 2004.

[HJMM04] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL 04: Principles of Programming Languages*, pages 232–244. ACM, 2004.

[IRR+04] N. Immerman, A. Rabinovich, T. Reps, S. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *CSL 04: Conference on Computer Science Logic*, LNCS 3210, pages 160–174. Springer-Verlag, 2004.

[JMSY94] J. Jaffar, M. J. Maher, P. J. Stuckey, and H. C. Yap. Beyond finite domains. In *PPCP 94: Principles and Practice of Constraint Programming*, LNCS 874, pages 86–94. Springer-Verlag, 1994.

[Kur94] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994.

[Lag85] J. C. Lagarias. The computational complexity of simultaneous diophantine approximation problems. *SIAM Journal of Computing*, 14(1):196–209, 1985.

[LBC03] S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In *CAV 03: Computer-Aided Verification*, LNCS 2725, pages 141–153. Springer-Verlag, 2003.

[LBC05] S. K. Lahiri, T. Ball, and B. Cook. Predicate abstraction via symbolic decision procedures. In *CAV 05: Computer Aided Verification*, LNCS 3576, pages 24–38. Springer-Verlag, 2005.

[LM05a] S. K. Lahiri and M. Musuvathi. An efficient decision procedure for UTVPI constraints. In *FroCos 05: Frontiers of Combining Systems*, LNCS 3717, pages 168–183. Springer-Verlag, 2005.

[LM05b] S. K. Lahiri and M. Musuvathi. An efficient Nelson-Oppen decision procedure for difference constraints over rationals. Technical Report MSR-TR-2005-61, Microsoft Research, 2005.

[LMO05] K. R. M. Leino, M. Musuvathi, and X. Ou. A two-tier technique for supporting quantifiers in a lazily proof-explicating theorem prover. In *TACAS 05: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 3440, pages 334–348. Springer-Verlag, 2005.

[LS04] S. K. Lahiri and S. A. Seshia. The UCLID decision procedure. In *CAV 04: Computer Aided Verification*, LNCS 3114, pages 475–478. Springer-Verlag, 2004.

[Mar99] P. Marquis. Consequence-finding algorithms. In *Handbook of Defeasible Reasoning and Uncertainty Management Systems (vol. 5): Algorithms for Defeasible and Uncertain Reasoning*, pages 41–145. Kluwer Academic Publishers, 1999.

[McM03] K.L. McMillan. Interpolation and SAT-based model checking. In *CAV 03: Computer-Aided Verification*, LNCS 2725, pages 1–13. Springer-Verlag, 2003.

[McM04] K.L. McMillan. An interpolating theorem prover. In *TACAS 04: Tools and Algorithms for Construction and Analysis of Systems*, pages 16–30. Springer-Verlag, 2004.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC 01: Design Automation Conference*, pages 530–535. ACM Press, 2001.

[MS01] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *PLDI 01: Programming Language Design and Implementation*, pages 221–231. ACM Press, 2001.

[Nec97] G.C. Necula. Proof carrying code. In *POPL 97: Principles of Programming Languages*, pages 106–119. ACM, 1997.

[NO79a] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):245–257, 1979.

[NO79b] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[NO80] G. Nelson and D. C. Oppen. Fast decision procedures based on the congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[Opp80] Derek C. Oppen. Complexity, convexity and combinations of theories. In *Theoretical Computer Science*, volume 12, pages 291–302, 1980.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *CADE 92: Conference on Automated Deduction*, LNCS 607, pages 748–752. Springer-Verlag, 1992.

[Pra77] V. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, Cambridge, Mass., September 1977.

[RA01] A. Riazanov and A.Voronkov. Vampire 1.1 (system description). In *IJCAR '01: International Joint Conference on Automated Reasoning*, LNAI 2083, pages 376–380. Springer-Verlag, 2001.

[SB04] S. A. Seshia and R. E. Bryant. Deciding quantifier-free Presburger formulas using parameterized solution bounds. In *LICS 04: Logic in Computer Science*, pages 100–109. IEEE Computer Society, July 2004.

[SG04] S. Shoham and O. Grumberg. Monotonic abstraction-refinement for CTL. In *TACAS 04: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2988, pages 546–560. Springer-Verlag, 2004.

[SRW99] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL 99: Principles of Programming Languages*, pages 105–118. ACM, 1999.

[TH96] C. Tinelli and M. T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In *FroCos 96: Frontiers of Combining Systems*, Applied Logic, pages 103–120. Kluwer Academic Publishers, 1996.

[TS05] N. Tillmann and W. Schulte. Parameterized unit tests. In *FSE 05: Foundations of Software Engineering*, pages 253–262. ACM Press, 2005.

[YM05] G. Yorsh and M. Musuvathi. A combination method for generating interpolants. In *CADE 05: Conference on Automated Deduction*, LNCS 3632, pages 353–368. Springer-Verlag, 2005.

[YRS04] G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS 04: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2988, pages 530–545. Springer-Verlag, 2004.

[ZZ96] H. Zhang and J. Zhang. Generating models by SEM. In *CADE 96: Conference on Automated Deduction*, LNAI 1104, pages 308–312. Springer-Verlag, 1996.