Solving Quantified First Order Formulas in Satisfiability Modulo Theories

by

Yeting Ge

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

January 2010

Clark Barrett

To my wife and our baby.

ACKNOWLEDGEMENTS

In writing this dissertation, I have benefited from the support, advice and good will from many people to whom I am eternally grateful.

I would like to extend my first thanks to my adviser, Dr. Clark Barrett, for his support, guidance and patience, as well as for his painstakingly editing my dissertation draft to the point of completion. I also want to acknowledge my gratitude toward Dr. Leonardo de Moura for mentoring me at Microsoft Research and working with me on a major part of this dissertation. I would like to thank Dr. Amir Pneuli who over the years had given me great encouragements and had been on my dissertation defense committee. I also would like to express my appreciations to my dissertation defense committee members: Dr. Benjamin Goldberg, Dr. Ernest Davis, and Dr. Morgan Deters. Many thanks to Dr. Morgan Deters for his careful proof reading. I would like to thank other members of the ACSys group for many excellent suggestions on this dissertation.

ABSTRACT

Design errors in computer systems, i.e. bugs, can cause inconvenience, loss of data and time, and in some cases catastrophic damages. One approach for improving design correctness is formal methods: techniques aiming at mathematically establishing that a piece of hardware or software satisfies certain properties. For some industrial cases in which formal methods are utilized, a huge number of extremely large mathematical formulas are generated and checked for satisfiability. For these applications, high-performance solvers, which automatically check the formulas, play a crucial role.

For example, propositional logic (SAT) solvers are very popular. However, it is rather expensive to encode certain problems in propositional logic and the encoding is tricky and hard to understand. Recently, Satisfiability Modulo Theories (SMT) solvers have been developed to handle formulas in a more expressive first order logic. In contrast to general theorem provers that check satisfiability under all models, SMT solvers check satisfiability with regard to some background theories, such as theories of arithmetic, arrays and bit-vectors. SMT solvers are efficient and automatic like SAT solvers, while accepting much more general formulas.

For some applications, SMT formulas with quantifiers are useful. Tradi-

tional SMT solvers only consider quantifier-free formulas. In general, deciding SMT formulas containing quantifiers is undecidable. In other words, there are no complete and sound algorithms for solving a quantified SMT formulas.

This dissertation proposes several novel techniques for solving quantified SMT formulas. For general quantifier reasoning in SMT, the practical approach adopted by most state-of-the-art SMT solvers is heuristics-based instantiation. We propose a number of new heuristics. Most important is the notion of "instantiation level" that solves several challenges of general quantifier reasoning at the same time. These new heuristics have been implemented in solver CVC3, and experimental results show that a significant number of additional benchmarks can be solved than could be solved by CVC3 before.

When only considering formulas restricted to be within certain fragments of first order logic, it is possible to have complete algorithms based on instantiation. We propose a series of new fragments, and prove that formulas in these new fragments can be solved by complete algorithm based on instantiation.

Finally, this dissertation addresses the correctness of solvers. A practical method to improve correctness is to ask an SMT solver to produce a proof for a case it solves, and then proceed to check the proof. The problem is that such a proof checker will be rather complicated because it has to deal with a lot of

proof rules. Thus the correctness of the proof checker becomes questionable. We propose a proof translator that translates a proof from SMT solver CVC3 into a trusted solver HOL Light. Experimental results show that this approach is feasible. When translating proofs, two faulty proof rules in CVC3 and two mis-labeled benchmarks in SMT-LIB were discovered.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer systems are pervasive and affect virtually every aspect of modern societies. In addition to hundreds of millions of servers and personal computers, which nearly every business depends on, even more embedded computers of all kinds of sizes and capabilities are used to regulate or control an astronomical number of devices. With the foreseeable increase of capabilities and drop of prices in the near future, computer systems will become even more complicated and will be applied in more fields. For example, it is estimated [12] that a cell phone will contain 20 million lines of code and a passenger car will have 100 million lines of code in 2014.

Design errors in computer systems, i.e. bugs in hardware or software, can

cause inconvenience and loss of data and time, and in some cases result in catastrophic damages. In a report [63] by the National Institute of Standards and Technology in 2002, software failures in the U.S. cost about 59.5 billion dollars every year. This figure does not yet include the loss due to design errors in mission-critical applications because these cases are unpredictable. For example, the Pentium 4 bug [16] cost Intel 475 million US dollars in 1994. In 1996, the first Ariane 5 rocket [25] exploded because of a software bug, with direct cost of 375 million dollars. In 2000 the FDA investigated a case [32] in which twenty-eight people were administered an excessive dosage and five people died because of a bug in the program for a radiotherapy machine. In 2003, a bug in an alarm system contributed a crucial link in a chain of events that led to the Northeastern blackout, with an estimated economic cost of around 6 billion [17]. In 2005, Toyota [13] recalled 160,000 cars for a bug in the software that controlled the engines.

Many techniques have been devised to improve the correctness of computer systems and ensure that computer systems behave as desired. The traditional and most widely used technique is testing. One of the problems of the testing techniques is most of the time only a fraction of all possible inputs cases can be tested. The formal methods technique consider correctness with regard to

all possible input cases.

1.1 Formal methods

Formal methods are mathematically-based techniques for the specification, development and verification of software and hardware systems [46]. Formal methods aim at mathematically establishing that a piece of hardware or software satisfies certain properties in a formal model of it. Unlike other approaches, formal techniques consider correctness with regard to all possible input cases. Generally speaking, formal methods can be classified into two categories, verification and synthesis. Formal verification techniques can prove or disprove that certain properties hold for a given piece of hardware or software. Synthesis techniques can guarantee a piece of hardware or software is correct by construction.

Since it is usually expensive or impossible in practice to prove total correctness, formal verification techniques are often applied to detect certain types of bugs, such as dead-lock, starvation, out-of-bound array access, etc. In general, the application of formal verification involves modeling, specification, and proving. Modeling involves representing the system being checked in some

formal model of it, usually simplified and abstracted at the same time. Specification means describing, in formal languages sometimes, the desired property to be checked for correctness. Proving will establish whether the model of the system satisfies the property or not. The main techniques of formal verification include model checking, abstract interpretation, and theorem proving.

1.1.1 Model checking

Model checking [27] has been successful, especially in hardware verification where it has become a standard technique. The piece of hardware or software being checked is usually represented by a Kripke structure. A Kripke structure is based on the possible worlds semantics, in which a node represents a state and an edge represents a transition. The properties are usually specified by formulas in a variant of temporal logic. Given the property, the model checking algorithm exhaustively checks all states reachable from initial states. If any state that represents an error is reachable, then the path from an initial state to the error state represents a counter-example in which the property is violated. If no error states are reachable, then the property holds. In explicit model checking, a state is explicitly represented. In symbolic model checking, a set of states is represented symbolically. For one approach of symbolic model

checking, a logic formula which represents both the property and the system is computed and discharged to a solver.

Traditional model checking can only handle systems with a finite number of states. Many techniques have been proposed to overcome this limitation. One of these techniques is predicate abstraction, in which a concrete system being checked is abstracted into a finite one. A popular technique of predicate abstraction is counter-example-guided abstraction refinement (CEGAR). A counter-example in the abstracted system is checked for feasibility in the concrete system. If the counter-example is infeasible, the abstract system is automatically refined based on some heuristics. Usually the checking of feasibility is done by discharging a logic formula to a solver. Another approach to deal with infinite state systems is bounded model checking, in which a loop is unrolled a finite number of times.

1.1.2 Abstract interpretation

Abstract interpretation [18] is traditionally a technique for static analysis that is used to check properties of a program.

The basic idea is to transform programs with operations in a concrete domain into an abstract domain. The computation in the abstract domain

is much easier than the computation in the concrete domain, but the results in the abstract domain still provide enough information about the concrete domain. When a suitable abstract domain is utilized, certain properties can be easily checked for very large program.

1.1.3 Theorem proving

In theorem proving, both the system and the property are represented by some logic formulas. The logics can be first order logic, higher order logic or others, and usually a set of axioms about the system is needed. Because logic formulas can represent nearly any system and property that can be formalized, theorem proving techniques in principle can handle any verification task. For example, traditionally [14] theorem proving has been used to deal with infinite state systems.

When first order logic or higher order logic are employed, in principle the proving of formulas in theories (such as the theory of arithmetic) commonly used in verification cannot be fully automated and thus requires manual effort. Because the proving usually requires human guidance, it is expensive to employ it for verification of large systems.

1.1.4 Extended static checkers

An extended static checker [49] behaves like a compiler that checks common errors. Unlike abstract interpretation techniques that rely on operations of abstract domains to derive useful information about the system, an extended static checker generates verification conditions based on preconditions and post-conditions. The preconditions and post-conditions are usually derived from the axiomatic semantics of the program (Floyd-Hoare logic). The verification conditions are usually expressed in first order logic with background theories and discharged to some solvers. If the verification condition is valid, then the desired property holds. The user can annotate the system to help the checker.

1.2 Verification engines

Many formal verification applications need to check the satisfiability of logic formulas. It is common that a huge number of extremely large formulas need to be checked in a single run. For these formulas, high performance solvers play a crucial role. A number of such solvers are available. This section briefly reviews common solvers used in verification, and more relevant details will be

discussed in later chapters.

1.2.1 SAT solvers

Solvers for propositional logic, known SAT solvers, are popular. The DPLL-based [19] SAT solvers have improved significantly in the last decade. In terms of complexity, any algorithm for satisfiability problem for propositional logic is NP-complete. The basic idea of DPLL-based algorithms is to guess a value for a propositional variable and then proceed to simplify the formula. If the guess is not successful, then backtrack. Modern SAT solvers employ clever and efficient heuristics for guessing, backtracking and simplification. Some details of the DPLL-based algorithm will be discussed in later chapters. Most modern DPLL-based SAT solvers only deal with formulas in conjunctive normal form (CNF).

1.2.2 First order logic solvers

First order logic allows quantifiers that range over individuals. Traditional solvers of first order logic only deal with formulas of pure first order logic (with equality in some cases). To employ these traditional solvers for verification applications, a set of axioms must be provided. However, some useful theories

such as the theory of integer arithmetic are not finitely axiomatizable and have non-standard models. Because of quantifiers, first order logic is semi-decidable, which means there is no guarantee of termination for any complete algorithm.

1.2.3 Higher order logic solvers

Higher order logic allows quantifiers to range over any object. Higher order logics are more expressive than first order logic and are convenient for describing certain systems. Since there is no sound and complete algorithm to decide a formula in higher order logic, any higher order logic solver inevitably needs human guidance. For this reason, higher order logic solvers sometimes are called proof assistants or interactive solvers.

1.2.4 Satisfiability Modulo Theories solvers

For some problems, the encoding into propositional logic is unintuitive and inefficient. Satisfiability Modulo Theories (SMT) [56] solvers are developed to accept formulas in first order logic. In contrast to general first order logic solvers that check satisfiability under all models, SMT solvers check satisfiability with regard to some background theories (such as theories of arithmetic,

arrays and bit-vectors). In other words, the interpretation of some symbols appearing in the formulas is restricted. SMT solvers are automatic and complete like SAT solvers and yet accept much more general formulas. Traditional SMT solvers can only solve ground formulas.

1.3 Quantifier reasoning in SMT

For some verification applications, quantified SMT formulas are useful. For example, quantified formulas are convenient for capturing what does not changed over loops (frame conditions), for summarizing invariants, and for axiomatizing theories for which decision procedures of ground formulas have not been implemented in an SMT solver. However, reasoning about quantified formulas in SMT is a long-standing challenge.

Broadly speaking, quantifier reasoning has been thoroughly studied in two situations. When every symbol is interpreted, for certain theories quantifier elimination procedures can be used. If every symbol is uninterpreted, techniques from general first order theorem proving can be applied. The challenge for quantifiers in SMT is that most quantified SMT formulas contain both interpreted and uninterpreted symbols. It is difficult to have a general decision

procedure for quantifiers in SMT. For example, there is no sound and complete procedure for first-order logic formulas of linear arithmetic with uninterpreted functions [41].

Some SMT solvers [22] integrate a general first order solver, say a superposition calculus solver, for quantifier reasoning. These solvers are refutation complete for formulas in pure first order logic with equality, but cannot guarantee completeness when other interpreted symbols appear in quantified formulas. Several first-order calculi have been proposed based on the idea of theory resolution [59]. These calculi provide nice theoretical results, yet no efficient implementations, because the computation of theory unifiers is too expensive or impossible for some background theories of interest.

Most state-of-the-art SMT solvers with support for quantifiers use heuristic-based quantifier instantiation [34, 26, 21] for quantifier reasoning. The idea is as follows: Suppose F^g is an instantiation of universally quantified formula F, and G is a ground formula. If ground formula $F^g \wedge G$ is unsatisfiable, then so is $F \wedge G$. Because $F^g \wedge G$ is a ground formula, its satisfiability can be easily checked by decision procedures for ground formulas that already exist in SMT solvers. The trick is to pick the right instantiation F^g. Of course, we can only hope that some good heuristics will help us, because the problem in

general is undecidable. A well known heuristic instantiation-based approach was introduced by the Simplify theorem prover [24].

Some fragments of first order logic with background theories are decidable. For some of these fragments, it is possible to have a complete decision procedure for quantified formulas based on instantiation. Given a formula F in these fragments, a conjunction of a finite number of instantiations of F can be constructed and shown to be equi-satisfiable to F. This is called complete instantiation.

1.4 Contributions

For general heuristic-based quantifier reasoning, we propose several new heuristics. The most important one is a heuristic called "instantiation level" that solves several challenges at the same time. We implemented the new heuristics in the SMT solver CVC3. Experimental results show that a number of additional benchmark can be solved than could be solved by CVC3 before.

For complete instantiation, we propose a series of new fragments and prove that formulas in these new fragments can be solved by algorithms based on complete instantiation.

This dissertation also proposes a proof translator that translates proofs from CVC3 into a trusted solver HOL Light. Experimental results show that a proof translator is a feasible solution to improve the correctness of SMT solvers. When translating proofs, we found two faulty proof rules in CVC3 and two mis-labeled benchmarks in the benchmark library SMT-LIB.

1.5 Organization

Chapter 2 introduces the background theory needed for this dissertation, namely the syntax and semantics of first order logic, a formal framework for SAT and SMT solvers called DPLL(T), and some heuristics employed in modern SAT and SMT solvers which are needed for discussion in later chapters.

Chapter 3 discusses heuristic-based quantifier instantiation, a general quantifier reasoning method in SMT. It introduces various improvements and the implementation details in the SMT solver CVC3. Some experimental results are given at the end of the chapter.

Chapter 4 is for complete instantiation. This chapter proposes several new fragments of first order logic, and proves that formulas in these new fragments can be solved by algorithms based on complete instantiation.

As SMT solvers become more sophisticated, the correctness of the solver itself becomes questionable. Since it is nearly impossible to prove an SMT solver is correct, a feasible solution is to check the proofs. Chapter 5 proposes a proof checker that translates a proof from CVC3 into a proof in a trusted solver. The proof checking is then done by the trusted solver.

Finally Chapter 6 concludes and gives some directions for future work.

Chapter 2

Background

This chapter introduces the theories and notations needed for the following discussion. This chapter can be skipped for those who are familiar with these concepts.

2.1 First order logic

First order logic is a well-studied logic and has found numerous applications in computer science and mathematics. The study of first order logic consists of a formal language, the semantics of the formal language, and the deduction and reasoning methods. The syntax of the formal language of first order logic

in this dissertation is as follows.

2.1.1 Syntax

The alphabet consists of a set of variable symbols, a set of predicate and function symbols, and a set of logical symbols. Usually it is assumed that a countable set of variable symbols is available. Variables are denoted by x, y, x_1, y_1, etc. The set of predicate and function symbols is called the *signature* of the language. It suffices to discuss a finite set of function and predicate symbols here. Each function symbol and predicate symbol is associated with a non-negative integer, the *arity*. A function or predicate is an n-ary one if the arity of it is n. Intuitively speaking, an n-ary function or predicate can take n arguments. Function symbols are usually denoted by f and g, and predicate symbols by p and q. The set of logical symbols contains quantifiers, propositional connectives, and parentheses. The quantifiers are $\forall$ and $\exists$. The propositional connectives are $\wedge$, $\neg$ and $\vee$.

A *term* is defined by the following recursive rules: A variable is a term. $f(t_1, t_2, \ldots, t_n)$ is a term if f is a n-ary function symbol and $t_1, t_2, \ldots, t_n$ are all terms. For example, x, $f(x)$, $f(g(x))$ are terms.

An *atomic formula* is of the form $p(s_1, s_2, \ldots, s_n)$ where p is a n-ary predi-

cate symbol and $s_1, s_2, \ldots, s_n$ are terms. A *formula* is defined by the following recursive rules: An atomic formula is a formula. If ϕ is a formula then $(\neg\phi)$ is a formula. If ϕ and ψ are formulas, then $(\phi \vee \psi)$ and $(\phi \wedge \psi)$ are formulas. If ϕ is a formula and x is a variable, then $(\forall x.\phi)$ and $(\exists x.\phi)$ are formulas.

Where there is no confusion, parentheses are often dropped. For example, $(\forall x.((\neg p(x)) \vee q(x)))$ is usually written as $\forall x.(\neg p(x) \vee q(x))$. A Σ-formula is a formula that only contains functions and predicate symbols in Σ.

In $\exists x.\phi$ and $\forall x.\phi$, x is called a *bound variable* and ϕ is the *body* of the quantified formula. If a variable is not a bound variable, then it is a *free variable*. A formula is a *sentence* if it does not contain any free variables.

Define $\forall\overline{x}.\varphi$ as the abbreviation of $\forall x_1 \forall x_2 \ldots \forall x_n \varphi$. The notation $\exists\overline{x}.\varphi$ is defined similarly.

If φ is a formula or a term, t is a term, and x is a variable, $\varphi[x/t]$ denotes the result of substituting t for all free occurrences of x in φ. For tuple $\overline{x}$ of variables and $\overline{t}$ of terms of the same length, $\varphi[\overline{x}/\overline{t}]$ is defined as the result of simultaneously substituting each x in $\overline{x}$ by the corresponding t in $\overline{t}$.

2.1.2 Semantics

The semantics of the first order language is defined as follows: Given a signature Σ, a Σ-structure M consists of $|M|$ (the domain) and an interpretation for variables and symbols in Σ. For a variable x, the interpretation x^M is an element in $|M|$. For an n-ary function symbol f, the interpretation f^M is an n-ary function on $|M|$. For a n-ary predicate p, the interpretation p^M is a subset of $|M|^n$, where $|M|^n$ is the n-ary Cartesian product over $|M|$. An element of $|M|^n$ is of the form $\langle v_1, v_2, \ldots, v_n \rangle$. The interpretation of an arbitrary term t is defined by the following recursive rules: If t is a variable x, then t^M is x^M. If t is of the form $f(s_1, s_2, \ldots, s_n)$ where f is an n-ary function symbol, then t^M is $f^M(s_1^M, s_2^M, \ldots, s_n^M)$. A 0-ary f is often called a *constant*, because f^M is a fixed element in $|M|$ in any given Σ-structure M.

$M\{x \mapsto v\}$ denotes a structure in which the variable symbol x is interpreted as v, $v \in |M|$, and all other variables, function symbols and predicate symbols have the same interpretation as in M. That is $x^{M\{x \mapsto v\}}$ is v. $M\{\overline{x} \mapsto \overline{v}\}$ denotes $M\{x_1 \mapsto v_1\}\{x_2 \mapsto v_2\} \ldots \{x_n \mapsto v_n\}$.

Given a signature Σ, a relation between a Σ-structure M and a Σ-formula ϕ, denoted by $M \models \phi$, is recursively defined as follows:

- $M \models p(t_1, \ldots, t_n)$, if $\langle t_1^M, \ldots, t_n^M \rangle \in p^M$, where $p(t_1, \ldots, t_n)$ is an atomic formula.

- $M \models (\psi \vee \phi)$, if $M \models \psi$ or $M \models \phi$.

- $M \models (\psi \wedge \phi)$, if $M \models \psi$ and $M \models \phi$.

- $M \models (\neg\psi)$, if it is not the case that $M \models \psi$.

- $M \models (\forall x.\psi)$, if for all $v \in |M|$, $M\{x \mapsto v\} \models \psi$, where ψ is a formula.

- $M \models (\exists x.\psi)$, if there is a $v \in |M|$ such that $M\{x \mapsto v\} \models \psi$.

$\forall$ is called the universal quantifier. A formula of the form $(\forall x.\psi)$ is a universally quantified one, which intuitively means ψ holds for all elements in the domain of the structure. $\exists$ is the existential quantifier and $(\exists x.\psi)$ is an existentially quantified formula. As can be seen from the semantics, $M \models (\forall x.\psi)$ if and only if $M \models (\neg\exists x.\neg\psi)$. If a formula does not have any quantifiers, then it is *quantifier free*. If a quantifier-free formula does not contain any variables, it is a *ground* formula.

$\neg$, $\vee$ and $\wedge$ are primitive propositional connectives. $\neg$ stands for negation, $\vee$ for disjunction and $\wedge$ for conjunction. Other propositional connectives are possible and can be defined using $\neg$, $\vee$ and $\wedge$. For example, $\phi \rightarrow \psi$ can be

defined as $(\neg\phi) \vee \psi$, which intuitively means ϕ implies ψ. $\phi \vee \psi$ is sometimes called the disjunction of ϕ and ψ, and similarly $\phi \wedge \psi$ is called the conjunction of ϕ and ψ. The conjunction and disjunction of s, where s is a set of formulas, are defined in the obvious way. A *literal* is an atomic formula or the negation of one.

Given a signature Σ and a formula ϕ, if there is a Σ-structure M such that $M \models \phi$, then ϕ is *satisfiable* under M. The structure M is called a *model* of ϕ. A set S of formulas is satisfiable if the conjunction of formulas in S is satisfiable. A model of a set of formulas is defined in the obvious way. If there does not exist an M such that $M \models \phi$, then ϕ is *unsatisfiable*. If for every Σ-structure M, $M \models \phi$, then ϕ is *valid*. If ϕ is not valid, then it is *invalid*. For example $\forall x.p(x)$ is satisfiable, $\forall x.p(x) \vee \exists x.\neg p(x)$ is valid, and $\forall x.(p(x) \wedge \neg p(x))$ is unsatisfiable. The satisfiability problem is to determine whether a formula is satisfiable or not. Similarly the validity problem is to determine the validity of a formula. In first order logic, it is easy to see that a formula ϕ is unsatisfiable if and only if $\neg\phi$ is valid. If formula ϕ if satisfiable if and only if formula ψ is, then ϕ and ψ are *equi-satisfiable*. Given formulas ψ and ϕ, $\psi \models \phi$ means that any model of ψ is also a model of ϕ.

2.1.3 Deduction

A central problem in the study of a logic is to devise a deductive calculus that can be used to determine the satisfiability (validity) of a formula in the language. A calculus is complete if any valid formula can be derived by the calculus. A calculus is sound if all derived formulas are valid. A number of sound and complete calculi for first order logic have been invented. For first order logic, there is no algorithm that can completely determine the validity of a formula in a finite amount of time. For any complete algorithm, there exists some invalid formula on which it does not terminate.

A number of general purpose solvers for first order logic are available. To use these solvers in verification applications, the system and the property being checked must be formalized by some first order logic formulas. Suppose the system is represented by formula ϕ and the property by p. The task is then to show that $\phi \rightarrow p$ is valid.

2.1.4 SAT solvers

Propositional logic can be seen as a restriction of first order logic in which all predicates are 0-ary ones. Because 0-ary predicates take no arguments at

all, variables, functions and quantifiers are unnecessary in propositional logic. Intuitively, formulas in propositional logic can only formalize properties for a finite domain. In propositional logic, an atomic formula is called a *propositional variable*. A propositional variable can have a *truth value*, either *true* or *false*. The negation of *true* is *false* and vice-versa. Given an assignment of truth values to all variables in a propositional formula, the formulas can be evaluated in the obvious way. It is easy to see that if there is an assignment that makes a formula evaluated to be *true*, then this propositional formula is satisfiable. The problem of deciding the satisfiability of a formula in propositional logic, the SAT problem, was the first problem proved to be in the *NP-complete* complexity class[15].

Propositional logic solvers, commonly known as SAT solvers, have improved significantly in the last decade. Modern DPLL-based solvers can solve industrial cases that contain millions of clauses and hundreds of thousands of propositional variables in a reasonable amount of time on normal computer servers.

Most DPLL-based SAT solvers only deal with formulas in conjunctive normal form (CNF). A *clause* is a disjunction of a set of literals. A CNF formula is a conjunction of a set of clauses. Any formula of propositional logic can be

```
1.  FUNCTION DPLL(F)
2.     F1 := Simplify(F);
3.     IF false = F1
4.     THEN RETURN (UNSAT);
5.     IF (all variables in F1 are assigned)
6.     THEN RETURN (SAT) ;
7.     pick a var v;
8.     IF (SAT) = DPLL(F1 ∧ v)
9.     THEN RETURN (SAT) ;
10.    ELSE RETURN DPLL(F1 ∧ ¬ v);
```

Figure 2.1: Naive DPLL algorithm

converted into an equivalent one in CNF. A naive algorithm is as follows: First use the De Morgon's laws [28] to push inward all negations. Next, use the distributive laws of conjunction and disjunction connectives to put the formula into CNF. The naive algorithm will result in an exponential number of clauses in terms of the number of variables in the original formula for some cases. More efficient algorithms for CNF conversion that only preserves satisfiability have been studied and used in most modern SAT solvers.

A naive DPLL algorithm is shown as a recursive function in Figure 2.1. The function accepts a CNF formula F and returns either *UNSAT* or *SAT*, which respectively means F is unsatisfiable or satisfiable. The key ideas of the DPLL algorithm are in the simplification (line 2) and splitting (lines 7-10). In the simplification step, the formula is simplified into a equi-satisfiable

one. If the resulting formula is *false* then the original formula is obviously unsatisfiable (lines 3,4). If the resulting formula is not *false*, then pick up a variable v that has not been assigned a truth value (line 7) and proceed under the assumption that v is *true* (line 8) or v is *false* (line 10). If there are no more un-assigned variables, then the original formula is satisfiable (line 5, 6). Intuitively speaking, the DPLL algorithm can be seen as a search on a tree with each node of in the tree representing a split. If every branch of the search tree leads to formula equivalent to *false*, then the problem is unsatisfiable.

For simplification, if a clause contains a literal evaluated to be *true*, then this clause is evaluated to *true* and thus can be removed for the set of clauses. If a clause c contains a literal l evaluated to be *false*, then the clause is equivalent to the clause c with l removed. An empty clause means *false* because all literals in it have been evaluated to be *false*. If there is an empty clause, then the set of clauses must be evaluated to *false*.

The original DPLL algorithm uses two rules, the *pure literal rule* and the *unit propagation rule*, for further simplification. A literal is *positive* if it is a propositional variable. A literal is *negative* if it is the negation of a propositional variable. If a propositional variable appears only in positive literals or appears only in negative literals, then the literal containing the

propositional variable is a *pure* literal. Suppose a positive pure literal l appears in formula F, then it is easy to see that F is equi-satisfiable to the formula in which l has been assigned *true*. If a negative pure literal l appears, then F is equi-satisfiable to the formula in which l is evaluated to be *false*. The pure literal rule identifies any pure literals and removes any clauses that contains a pure literal. Since in modern SAT solvers it is rather expensive to identify pure literals, the pure literal rule is usually not used as often as the unit propagation rule.

A clause is a *unit* clause if it only contains one literal. Suppose, without loss of generality, a unit clause containing a positive literal l appears in F. Then F is equi-satisfiable to the formula in which l has be assigned to be *true*. The unit propagation rule finds a unit clause and simplifies the set of clauses accordingly. In many industrial cases, unit clauses appear frequently and one application of the unit propagation rule can result in a cascade of unit clauses. Modern SAT solvers utilize efficient data structures to keep track of unit clauses which makes the unit propagation rule rather cheap to carry out. Therefore, the unit propagation rule is aggressively applied until no further unit clauses, and this process is often called *boolean constraint propagation* or *BCP* for short. Some reports [68] show that most of the time, about 80%

or more, is spent on BCP in modern SAT solvers.

Other heuristics employed in modern SAT solvers including learning and non-chronological back-tracking. For more details please refer to [68].

If a propositional formula is satisfiable, a SAT solver can return a concrete assignment, i.e. *true* or *false*, for every propositional variable. The assignment is usually useful. For example, the assignment can be used to verify that a formula reported as satisfiable is indeed satisfiable.

2.2 Satisfiability modulo theories

A lot of verification applications need to check formulas that involve arithmetic or other operations that cannot be formalized by a finite or any decidable set of formulas. Therefore, it is difficult to utilize general first order logic solvers for these applications. What is needed in these applications are tailored methods to determine the satisfiability with regard to the models in which certain symbols are interpreted as intended. For example, given the formula $a + b < 0 \wedge a < 0$, the intended interpretation for a and b are some numbers and $<$ is the less-than relation over numbers. In other words, the interpretations of certain symbols are restricted here. The symbols with re-

stricted interpretations are called *interpreted* ones, while other symbols are *uninterpreted*.

Satisfiability with regard to theories is defined as follows: A theory T is a (possibly infinite) set of sentences, which is believed to have some models. Given a formula ϕ that may contain some interpreted symbols in T, ϕ is T-satisfiable if there is structure M^T such that M^T satisfies all sentences in T and $M^T \models \phi$, i.e. $M^T \models T \cup \{\phi\}$. A formula is T-valid if it is satisfiable under all structures that satisfy all sentences in T.

Research on *Satisfiability Modulo Theories* (SMT) studies the satisfiability problem with regard to theories. The theories considered in SMT are called *background theories*. A solver that checks satisfiability in SMT is referred to as an SMT solver. A formula that contains interpreted symbols from the background theories is called an SMT formula. Common background theories include the theory of equality, integer and real arithmetic, arrays, and bit-vectors. From now on, interpreted symbols from the theory of equality and arithmetic are assumed. $=$ is used to denote the interpreted predicate symbol for equality, and $\equiv$ denotes the equality in the meta-language. The SMT-LIB [8] is a library of benchmarks for SMT solvers.

2.2.1 SMT solvers

An SMT solver is a special kind of constraint solver. There are other types of constraint solver that deal with formulas in theories considered in current SMT research. For example, commercial linear programming solvers can solve cases that contain millions of integer or real variables. Unlike formulas for other constraint solvers, SMT formulas usually contain uninterpreted symbols and show highly complex propositional structures. Some SMT solvers can deal with formulas with interpreted symbols from several background theories.

Generally speaking, algorithms for SMT solvers can be categorized into two kinds, lazy and eager. In the eager approach, a formula is transformed into an equi-satisfiable propositional formula which then is sent to a SAT solver. Any off-the-shelf SAT solver can be used for the eager approach. For example, difference logic formulas can be transformed into propositional ones by bit-blasting [65]. The advantage of the eager approach is that any off-the-shelf SAT solvers can be used as is. However, the encoding of the problem into propositional logic may be awkward for some theories of interest. The model of a satisfiable case or the proof of a unsatisfiable, when needed, have to be "de-encoded".

In the lazy approach, an SMT solver integrates a SAT solver and one or

```
1.  IF (UNSAT = SAT_SOLVE(F^B))
2.  THEN RETURN UNSAT
3.  ELSE FOR_EACH assignment ASSN of F^B DO
4.           IF (SAT = THEORY_SOLVE(ASSN))
5.           THEN RETURN SAT
6.       END_FOR
7.       RETURN UNSAT
```

Figure 2.2: Naive lazy SMT algorithm

more *theory solvers* that are capable of deciding whether a conjunction of literals in the theory is satisfiable or not. Most theory solvers can only deal with quantifier-free formulas. In a lazy SMT solver, an SMT formula F is first abstracted into a propositional formula F^B, and a naive method of abstraction is to simply replace every atomic formula by a fresh propositional variable. A naive lazy algorithm is shown in Figure 2.2.

In the beginning F^B is sent to a SAT solver (line 1). If F^B is propositionally unsatisfiable, then clearly F is unsatisfiable (line 2). If F^B is satisfiable, the SAT solver provides an assignment for every propositional variable in F^B, and the theory solver is called(line 3). If the theory solver determines that the assignment, which represents a conjunction of first order literals, is T-satisfiable, then F is satisfiable (line 4,5). If the assignment is T-unsatisfiable, then the SAT solver is asked to produce another possible assignment for F^B.

If there are no more possible propositional assignments for F^B, then F is unsatisfiable (line 7).

Suppose the formula is $a < 2 \wedge \neg(a < 3))$ where a is an integer. The formula is abstracted into $B_1 \wedge \neg B_2$, and it is satisfiable with $B_1 \equiv \mathit{true}$ and $B_2 \equiv \mathit{false}$. For the theory solver, this means $a < 2$ and $a \not\equiv 3$, which is unsatisfiable. Since there are no more possible propositional assignments, the formula is unsatisfiable.

Needless to say, a lot of improvements are possible for the naive lazy SMT algorithm. DPLL(T) is popular schema for the interactions between the SAT solver and the theory solvers in the DPLL-based lazy SMT solvers. A key point of the DPLL(T) schema is, instead of passively checking if a propositional assignment is satisfiable in the theory, a theory solver should actively deduce constraints on the abstract propositional variables to improve the overall efficiency.

2.3 Abstract DPLL Modulo Theories

The Abstract DPLL Modulo Theories framework [56] is a formalism for lazy SMT solvers that integrate a DPLL-based SAT solver. This framework pro-

vides a general, yet precise, model for modern lazy SMT solvers based on the DPLL(T) structure. Using this framework, some key properties, say completeness and soundness, of modern SMT solvers can be proved.

The framework describes SMT solvers as transition systems. A transition system consists of a set of *states* and a binary relation $\Longrightarrow$, called the *transition relation*, over the states. The transition relation is defined declaratively by a set of *transition rules*. A state is either the *Fail* (denotes T-unsatisfiability) state or a pair of the form $M \parallel F$, where F is a CNF formula and M is a sequence of literals that appears in F. Intuitively, F is the formula of which the satisfiability is being checked and M represents a partial assignment for literals in F. A literal l in M may contain the special annotation l^{d} which indicates it is a *decision* literal. Where no confusion, M is sometimes used to denote the sequence of literals and F is used to denote the CNF formula in a state.

The transition rules of the Abstract DPLL Modulo Theories are shown in Figure 2.3. In each rule, a comma is used to separate clauses of the CNF formula, and C and l respectively denote a clause and a literal. $M \models F$ means that $(\wedge_{l_i \in M} l_i) \models F$ in propositional logic. $M \models_T F$ means $(\wedge_{l_i \in M} l_i) \models F$ in first order logic modulo the background theory T. For example, the unit

propagation rule UnitPropagate says that if there is clause $C \vee l$ that $M \models \neg C$, then l must be *true* and thus M is expanded to $M\ l$ while F is not changed. A transition rule of the form $M_1 \parallel F_1 \Longrightarrow M_2 \parallel F_2$ maintains the invariant that F_1 and F_2 are equi-satisfiable.

Given an initial state of the form $\emptyset \parallel F_0$, where $\emptyset$ denotes the empty sequence of literals, the goal is to derive a *final* state by applying the transition rules. A final state is either *Fail* or a state $M \parallel F$ such that the set of literals in M is T-satisfiable and M$\models F$. If a final state is not *Fail*, then the original formula F_0 is T-satisfiable.

When a SMT solver is represented by a transition system, properties of the solver, like soundness and completeness, can be proved. The basic idea is to define some order on the states and then proceed to show that all the transition rules will transform a state to another with regard to the order. For example, an SMT solver using any subset of the rules shown in Figure 2.3 is sound and complete, but may not terminate due to the T-Forget rule and Restart rule. A complete description of the framework is in [56].

UnitPropagate :

$$M \parallel F, C \vee l \implies M\ l \parallel F, C \vee l \quad \textbf{if} \begin{cases} M \models \neg C \\ l \text{ is undefined in } M \end{cases}$$

Decide :

$$M \parallel F \implies M\ l^{\mathsf{d}} \parallel F \quad \textbf{if} \begin{cases} l \text{ or } \neg l \text{ occurs in a clause of } F \\ l \text{ is undefined in } M \end{cases}$$

Fail :

$$M \parallel F, C \implies \mathit{Fail} \quad \textbf{if} \begin{cases} M \models \neg C \\ M \text{ contains no decision literals} \end{cases}$$

Restart :

$$M \parallel F \implies \emptyset \parallel F$$

T-Propagate :

$$M \parallel F \implies M\ l \parallel F \quad \textbf{if} \begin{cases} M \models_T l \\ l \text{ or } \neg l \text{ occurs in } F \\ l \text{ is undefined in } M \end{cases}$$

T-Learn :

$$M \parallel F \implies M \parallel F, C \quad \textbf{if} \begin{cases} \text{each atom of } C \text{ occurs in } F \text{ or in } M \\ F \models_T C \end{cases}$$

T-Forget :

$$M \parallel F, C \implies M \parallel F \quad \textbf{if} \begin{cases} F \models_T C \end{cases}$$

T-Backjump :

$$M\ l^{\mathsf{d}}\ N \parallel F, C \implies M\ l' \parallel F, C \quad \textbf{if} \begin{cases} M\ l^{\mathsf{d}}\ N \models \neg C, \text{ and there is} \\ \text{some clause } C' \vee l' \text{ such that:} \\ \quad F, C \models_T C' \vee l' \text{ and } M \models \neg C', \\ \quad l' \text{ is undefined in } M, \text{ and} \\ \quad l' \text{ or } \neg l' \text{ occurs in } F \text{ or in } M\ l^{\mathsf{d}}\ N \end{cases}$$

Figure 2.3: Transition rules of Abstract DPLL Modulo Theories

Chapter 3

Heuristic instantiation

As discussed earlier, the practical approach to general quantifier reasoning in SMT is heuristic-based instantiation, pioneered by the prover Simplify [24]. Simplify has been successfully applied in a variety of software verification projects including ESC/Java [30]. For many years Simplify has represented the state-of-the-art of quantifier reasoning in SMT. The basic idea is as follows: Suppose F is an universally quantified formula and F^g is an instantiation of F. Then F^g is a ground formula and can be easily checked by a traditional SMT solver. If F^g is unsatisfiable, then F is unsatisfiable. The number of instantiations of a quantified formula can be infinite, and the trick is how to pick up the right ground term for instantiation. Of course, when the formula

is satisfiable, no such ground terms exist.

This chapter proposes several improvements of the heuristic-based quantifier reasoning in SMT solvers based on the DPLL(T) architecture. Most of the results in this chapter have been reported in [34]. At the beginning of this chapter, the Abstract DPLL Modulo Theories framework is extended with rules for quantifiers. Next, the main heuristics employed by Simplify are introduced as strategies within the Abstract DPLL Modulo Theories framework. A number of improvements to Simplify's strategies are proposed, and most novel among them is a heuristic called *instantiation level* that effectively prioritizes the candidate terms for instantiation. These new heuristics have been implemented in the SMT solver CVC3, and the details of the implementations are discussed. This chapter concludes with a set of experimental results that demonstrate the effectiveness of the new heuristics in CVC3.

3.1 Modeling quantifier instantiation

The Abstract DPLL Modulo Theories framework can be extended to include rules for quantifier instantiation. The key idea is to allow also quantified formulas wherever atomic formulas are allowed. An *abstract atomic formula*

is either an atomic formula or a sentence of the form $\forall\overline{x}.\varphi$ or $\exists\overline{x}.\varphi$.

An *abstract literal* is either an abstract atomic formula or the negation of one. An *abstract clause* is a disjunction of abstract literals. From now on in this chapter, literals in the Abstract DPLL Modulo Theories can be abstract literals and clauses can be abstract clauses. For instance, in the state $M \parallel F$, M is a sequence of abstract literals and F is a conjunction of abstract clauses.

With this slight modification, the two rules in Figure 3.1 are added to Abstract DPLL Modulo Theories to model quantifier instantiation. Without loss of generality, it is assumed that a quantified formula in M only appears positively. If a quantified formula is negated, the negation can be pushed inside the quantifier.

The $\forall$-Inst rule may be applied if a literal of the form $\forall\overline{x}.\varphi$ appears in M. When applied, the $\forall$-Inst rule adds a new clause $\neg\forall\overline{x}.\varphi \vee \varphi[\overline{x}/\overline{s}]$ to F, where $\overline{s}$ are ground terms. The new clause is an implication $\forall\overline{x}.\varphi \rightarrow \varphi[\overline{x}/\overline{s}]$, which means that if $\forall\overline{x}.\varphi$ holds then so does $\varphi[\overline{x}/\overline{s}]$.

The $\exists$-Inst rule can be applied if a literal of the form $\exists\overline{x}.\varphi$ appears in M. The $\exists$-Inst rule adds a clause $\neg\exists\overline{x}.\varphi \vee \varphi[\overline{x}/\overline{c}]$ to F, where $\overline{c}$ are fresh constants.

Example 3.1.0.1

Suppose a and b are constant symbols and f is an uninterpreted function

$\exists$-Inst :

$$M \parallel F \quad \Longrightarrow \quad M \parallel F,\, \neg\exists\overline{x}.\varphi \vee \varphi[\overline{x}/\overline{c}] \quad \textbf{if} \begin{cases} \exists\overline{x}.\varphi \text{ is in } M \\ \overline{c} \text{ are fresh constants} \end{cases}$$

$\forall$-Inst :

$$M \parallel F \quad \Longrightarrow \quad M \parallel F,\, \neg\forall\overline{x}.\varphi \vee \varphi[\overline{x}/\overline{s}] \quad \textbf{if} \begin{cases} \forall\overline{x}.\varphi \text{ is in } M \\ \overline{s} \text{ are ground terms} \end{cases}$$

Figure 3.1: Transition rules for quantifier reasoning.

symbol. The task is to prove the validity of the formula $(0 \leq b \wedge (\forall\, x.\, x \geq 0 \rightarrow f(x) = a)) \rightarrow f(b) = a$ where the background theory is arithmetic. First, the formula is negated and put into abstract CNF:

$$0 \leq b \quad \wedge \quad \forall\, x.\, (\neg(x \geq 0) \vee f(x) = a) \quad \wedge \quad \neg(f(b) = a)\ .$$

Let $l_1, l_2, \neg l_3$ denote the three abstract literals respectively in the above clauses. Let l_4 denote the abstract literal $b \geq 0$. Then the following is a derivation in

the extended Abstract DPLL Modulo Theories framework:

$$
\begin{array}{lrcll}
 & \emptyset & \| & l_1,\, l_2,\, \neg l_3 & \text{(initial state)} \\
\Longrightarrow^* & l_1\; l_2\; \neg l_3 & \| & l_1,\, l_2,\, \neg l_3 & \text{(by UnitPropagate)} \\
\Longrightarrow & l_1\; l_2\; \neg l_3 & \| & l_1,\, l_2,\, \neg l_3,\, \neg l_2 \vee \neg l_4 \vee l_3 & \text{(by } \forall\text{-Inst)} \\
\Longrightarrow & l_1\; l_2\; \neg l_3\; l_4 & \| & l_1,\, l_2,\, \neg l_3,\, \neg l_2 \vee \neg l_4 \vee l_3 & \text{(by } T\text{-Propagate)} \\
\Longrightarrow & \mathit{Fail} & & & \text{(by Fail)}
\end{array}
$$

The initial state is $\emptyset \parallel l_1,\, l_2,\, \neg l_3$ and F consists of three clauses l_1, l_2 and $\neg l_3$. Since all three clauses are unit clauses, after three steps of UnitPropagate, the state becomes $l_1\; l_2\; \neg l_3 \parallel l_1,\, l_2,\, \neg l_3$. Next, l_2 is picked for the $\forall$-Inst rule and the variable x in l_2 is instantiated with constant b. The resultant clause $\neg l_2 \vee \neg(b \geq 0) \vee f(b) = a$ is added into F. Since l_1 implies $b \geq 0$ and thus $l_1\; l_2\; \neg l_3 \models_T b \geq 0$, the T-Propagate rule derives the state $l_1\; l_2\; \neg l_3\; (b \geq 0) \parallel l_1,\, l_2,\, \neg l_3,\, \neg l_2 \vee \neg(b \geq 0) \vee f(b) = a$. The last transition is possible because M falsifies the last clause in F and contains no decisions (case-splits). As a result, it is concluded that the original clause set is T-unsatisfiable, which implies that the original formula is valid in T.

By using an analysis similar to that in [56], it is easy to prove that the $\exists$-Inst and $\forall$-Inst rules preserve the satisfiability of F and therefore the soundness of the transition system. To guarantee termination, the number of applications

of the two rules must be limited. It is obvious that there is no benefit from applying the $\exists$-Inst rule more than once to a existentially quantified formula in M. On the other hand, a universally quantified formula may need to be instantiated with different ground terms to prove that F is unsatisfiable. The key point for heuristic-based instantiation is to effectively control the number of applications of the $\forall$-Inst rule. From now on, only the $\forall$-Inst rule will be discussed.

3.2 Strategies for instantiation

3.2.1 Instantiation via matching

A naive strategy for applying the $\forall$-Inst rule is as follows: Once $\forall$-Inst is selected for an abstract literal $\forall \overline{x}.\varphi$, instantiate $\overline{x}$ with all possible ground terms in a set G. A natural choice for G is the set of ground terms that occur in M. We call this approach *naive instantiation*. In CVC3, each variable has a type, and only ground terms with the same type of $\overline{x}$ are used in naive instantiation.

Naive instantiation is often sufficient for cases that contain a small number of ground terms. When the formula being checked contains a large number of ground terms, naive instantiation will not work because too many instantia-

tions need to be checked.

The Simplify prover uses a better heuristic in which a smaller set of ground *relevant* terms are selected for instantiation. Within the Abstract DPLL Modulo Theories framework, the concept of relevant terms can be explained as follows: given an abstract literal $\forall \overline{x}.\varphi$ that appears in M, try to find a term t, a ground term g and ground terms $\overline{s}$ such that t is a non-variable sub-term of $\forall \overline{x}.\varphi$ containing the variables $\overline{x}$, g appears in a literal in M and $t[\overline{x}/\overline{s}]$ is *equivalent*[1] to g. In this case, it is expected that instantiating $\overline{x}$ with $\overline{s}$ is more likely to be helpful than instantiating with other ground terms. The terms $\overline{s}$ are the relevant terms.

Example 3.2.1.1

Consider again the formula in Example 3.1.0.1. When $\forall$-Inst is applied, M consists of the following sequence of literals:

$$0 \leq b, \quad \forall\, x.\, (\neg(x \geq 0) \vee f(x) = a), \quad \neg(f(b) = a)\ .$$

There are four ground terms appearing in M: 0, a, b, and $f(b)$. Thus, naive instantiation would apply $\forall$-Inst four times, once for each ground term. On the other hand, notice that $f(x)$ is a non-variable sub-term of the quantified

[1] the definition of equivalence will be discussed later in the chapter.

formula. If $f(x)$ is instantiated with x bound to b, then $f(x)[x/b]$ is $f(b)$ and $f(b)$ appears in a literal in M. Therefore, following Simplify's heuristic, b is used to instantiate $\forall\, x.\, (\neg(x \geq 0) \vee f(x) = a)$. The resultant instantiation, in this case, is actually the only instantiation needed.

The question is then how to efficiently find the terms t, g and $\overline{s}$. One approach works as follows: Given a non-variable sub-term t containing $\overline{x}$, instantiate t with all available ground terms in M and then proceed to check if there is a term among the results of instantiation that is equivalent to a term appearing in a literal in M. For some cases, this approach results in too many instantiations. Suppose a term contains n variables and m ground terms are available, then n^m instantiations are needed. It is also expensive to check if an instantiation of t is equivalent to a term in M.

Simplify uses E-matching to find t, g and $\overline{s}$. Given an abstract literal $\forall\overline{x}.\psi$ in M, first select one or more non-variable sub-terms or predicates, each of which contains $\overline{x}$. Suppose t is such a selected term or predicate and g is ground term in M, Simplify then checks if there are ground terms $\overline{s}$ such that $t[\overline{x}/\overline{s}]$ is equivalent to g. If there are such $\overline{s}$, then $\psi[\overline{x}/\overline{s}]$ is sure to contain some ground terms that are equivalent to some terms in M. The term t is called a *trigger* in Simplify. Not every term or predicate that contain $\overline{x}$ can

be selected as a trigger. Sometimes no single term or predicate contains all variables in $\overline{x}$. The heuristics for selecting triggers will be discussed later in the chapter.

Ideally, equivalence modulo background theories, i.e. $M \models_T t[\overline{x}/\overline{s}] = g$, should be checked. However, this is too expensive or impossible to check for some theories of interest. In Simplify, the equivalence is checked modulo a set of equalities E implied by $M \cup T$. In other words, $E \models_= t[\overline{x}/\overline{s}] = g$ is checked, where $\models_=$ means satisfiability modulo the theory of equality.

In the unification theory for general theorem proving, the problem of checking whether $E \models_= t[\overline{x}/\overline{s}] = g$ holds is called *E-matching*, and is often denoted by $t =^? g$. Ideally, one would like E to be the set $E_{M \cup T}$ of all the equalities entailed by $M \cup T$. In general, this is not feasible because of theoretical or practical limitations depending on the specific background theory T. In Simplify, E consists of the congruence closure of the ground equalities appearing in M. CVC3 adopts the same method as well, and uses an E-matching algorithm similar to the one used by Simplify.

For some theories [4], such as theories of an associative, commutative and idempotent symbol or the theory of Abelian groups, there are efficient algorithms to compute the E-matching problem.

3.2.2 Eager instantiation versus lazy instantiation

So far, the question is *how* to apply the rule $\forall$-Inst to a given quantified formula. An orthogonal question is *when* to apply it. One strategy, called *lazy instantiation*, is to apply $\forall$-Inst only when it is the only applicable rule. At the opposite end of the spectrum, the *eager instantiation* is to apply $\forall$-Inst as soon as a universal quantified formula is added into M.

A *round* of instantiation is a series of consecutive applications of the $\forall$-Inst rule.

In Simplify, propositional search and quantifier instantiation are interleaved. When Simplify has a choice between instantiation and splitting, it will generally favor instantiation. Thus, Simplify can be seen as employing a form of eager instantiation. Some solvers [29] use the lazy approach. As will be shown in later sections, under certain situations eager instantiation is preferred, while lazy instantiation is better for some other cases. Therefore, some heuristics that can take advantage of both approaches are needed. The instantiation level heuristics discussed in Section 3.3.8 can be seen as a heuristic that balances eager and lazy instantiation.

3.3 Improving instantiation strategies

This section describes a number of improvements to the basic strategies discussed above. These strategies have been implemented in CVC3, and an evaluation of these improvements is given in Section 3.4.

3.3.1 Triggers

Consider a generic quantified formula $\forall \overline{x}.\varphi$. The first step in the heuristic-based instantiation described above is to find triggers within φ. CVC3 improves on Simplify's automated trigger generation methods in several ways.

Simplify requires that a trigger contain no more variables other than those in $\overline{x}$. For example, in the formula $\forall x.(f(x) \rightarrow \forall y.g(x,y) < 0)$, the term $g(x,y)$ will not be selected by Simplify because it contains y which is not bound by the outermost quantifier. CVC3 relaxes this restriction, and every sub-term of φ that contains all the variables in $\overline{x}$ and at least one function or predicate symbol is considered a viable trigger. In the above example, CVC3 will select $g(x,y)$ as a trigger. In our experimental evaluation, some benchmarks can be proved only when terms of this kind are used as triggers.

CVC3 treats an interpreted symbol as an uninterpreted one in selecting

triggers except for a few cases. A term is *useful* for matching in CVC3 if is can be selected as a trigger. The notation of 'useful' is defined by the following recursive rules:

- A variable is not useful.
- An equality is not useful.
- Terms of arithmetic operations of $*$, $+$ and $-$ are not useful.
- A term of the form $s < t$ or $s \leq t$ is useful if s or t is useful[2].
- All other terms are useful.

An equality is never selected as a trigger because allowing an equality as a trigger causes too many useless instantiations. For the same reason, terms of arithmetic operations of $*$, $+$ and $-$ are not selected. For terms of the form $s < t$ or $s \leq t$, if both s and t are not useful, they will not be selected as triggers. For example $0 \leq x$,$f(x) = y$ will not be selected as triggers in CVC3, while $f(x) < g(t)$ will be selected.

[2]In CVC3, $>$ and $\geq$ are converted into $<$ and $\leq$ as a pre-processing step.

3.3.2 Avoiding instantiation loops

Adding an instantiation to F may introduce new ground terms into M and the new terms may bring new matching and instantiation opportunities. Sometimes, this situation can cause an instantiation loop which may lead the solver into an infinite loop and prevent other needed instantiations from being carried out. Simplify uses a simple syntactic check to prevent this type of loop. When a potential trigger t and some syntactical instances of t both occur in the formula, t will be discarded as a trigger. For example, in $\forall x.P(f(x), f(g(x)))$, the term $f(x)$ will not be selected as a trigger because an instance of $f(x)$, namely $f(g(x))$, occurs in the formula. Suppose $f(x)$ is selected as a trigger and suppose ground term $f(a)$ is used in matching with $f(x)$, then a new ground term $f(g(a))$ appears in the result of instantiation and $f(g(a))$ can be matched with f(x) again resulting in $f(g(f(a)))$. This process can continue, resulting in an infinite chain of matching and instantiation.

Instantiation loops are particularly harmful for solvers that employ the eager instantiation method, because a round of instantiation in the eager approach will finish only when no more matching and instantiations are possible.

While simple and inexpensive, Simplify's static loop detection is insufficient to detect more subtle forms of loops, as shown in the following example.

Example 3.3.2.1

Consider a state $M \parallel F$ with M containing the abstract literal $\psi \equiv \forall x.\,(x > 0 \rightarrow \exists y.\, f(x) = f(y) + 1)$ where f is uninterpreted. The only trigger for ψ is $f(x)$ and Simplify has no reason to reject this trigger.

Now, suppose the set of ground terms contains $f(3)$, then an application of $\forall$-Inst may add the abstract clause $\neg\psi \vee \exists y.\, f(3) = f(y) + 1$ to F. Then, after an application of UnitPropagate and of $\exists$-Inst, the literal $f(3) = f(c_1) + 1$ can be added to M, where c_1 is fresh constant. The introduction of the ground term $f(c_1)$ can give rise to a similar application of the rules UnitPropagate and $\exists$-Inst resulting in a new term $f(c_2)$, and so on.

To prevent instantiation loops like those in the example above, in addition to Simplify's static loop detection method, CVC3 also implements a general method that dynamically recognizes loops (including loops caused by groups of formulas together) and then disables the triggers that cause the loops. The basic idea is to keep a record of any successful matching and resultant instantiations. Suppose t is successfully matched with g and an instantiation I added, then any *new* ground term appearing in I is linked to a tuple $\langle t, g, I\rangle$. A new ground term is one that has not appeared in any literal in M so far. Note that a new ground term n in I can later be used for matching and may appear

in a tuple $\langle t_2, n, I_2 \rangle$. Now suppose a ground term is successful matched with trigger t_i and one variable x is bound to ground term g_i. The g_i's linked tuple, if there is one, $\langle t_j, g_j, I_j \rangle$ is checked. If t_j is the same trigger as t_i, then there is a cycle and trigger t_i will be disabled. If t_j is not the same trigger as t_i, then g_j's linked tuple $\langle t_k, g_k, I_k \rangle$ is checked to see if t_k is the same as t_i, and so on.

If the theory solvers of a SMT solver will not add new terms into F, as in the standard Abstract DPLL Modulo Theories, the dynamic loop detection will catch any instantiation loops. However, CVC3 implemented an extension of the standard DPLL modulo theories for better performance, and new terms can be added into F. Suppose after a round of matching and instantiation, the SMT solver calls its theory solvers that can add some new ground terms. The links between new terms and tuples are then lost for the new terms introduced by theory solvers.

In the dynamic loop detection, suppose a trigger for $\forall x.\psi$ is matched a ground term where x is bound to g and a loop is detected. One can either abandon g or add $\neg(\forall x.\psi) \vee \psi[x/g]$ into the set of clauses F before disabling the trigger. The experiments with CVC3 showed that for benchmarks in SMT-LIB it is usually better to add the formula $\neg(\forall x.\psi) \vee \psi[x/g]$. This somewhat surprising result suggests that loops may not always be bad. For example, the

following case cannot be proved unsatisfiable by matching and instantiation if loops are not allowed: $\{f(0) = 1, f(20) < 0, \forall x.\neg(x- > 0) \vee f(x) = f(x-1) * (-1000)\}$.

Both loop detection methods are eventually abandoned in CVC3 because the instantiation level heuristic described in Section 3.3.8 below is much more effective.

3.3.3 Multi-trigger generation

When there are no terms or predicates that contain all the variables in $\overline{x}$, Simplify generates a *multi-trigger* consisting of a set of terms in φ that together contain all (and exactly) the free variables in $\overline{x}$.

CVC3 uses essentially the same method to generate a multi-trigger. When there are several potential sets of terms, CVC3 uses a heuristic based on *polarity* to select one set of terms for the following abstract CNF formula, which essentially formalize predicate P is transitive.

$$\forall x, y, z.(\neg P(x, y) \vee \neg Q(y, z) \vee R(x, z)) \ ,$$

Given an abstract CNF formula F, an atomic formula in F has positive polarity if the it appears only positively in F and has negative polarity if it

occurs only negatively. If an atomic formula appears both positively and negatively, it has both positive and negative polarity. CVC3 will choose the set $\{P(x,y), Q(y,z)\}$ as a multi-trigger because $P(x,y)$ and $Q(y,z)$ have the same polarity (negative) and together contain all bound variables. On the other hand, the set $\{P(x,y), R(y,z)\}$ will not be chosen because $R(y,z)$ has positive polarity. CVC3 also identifies formulas that axiomatize antisymmetry. For formulas of the form $\forall x, y. \neg P(x,y) \vee \neg P(y,x) \vee x = y$, CVC3 generates a multi-trigger $\{P(x,y), P(y,x)\}$.

3.3.4 Matching algorithm

As mentioned in the previous section, when E-matching triggers, CVC3 chooses a rather restricted subset of the equalities entailed by $M \cup T$, where T is the background theory and M is the current set of assumed abstract literals. As M is modified, CVC3 computes and stores the congruence closure of the ground equations in M.When the $\forall$-Inst rule is used for an abstract literal $\forall \overline{x}.\varphi$, CVC3 E-matches the triggers of $\forall \overline{x}.\varphi$ against all ground terms in M.

CVC3 implements a sound and terminating E-matching algorithm based on the standard syntactic unification algorithm. Suppose the trigger is t of the form $f(t_1, \ldots, t_n)$ where f is an uninterpreted symbol. For each of the

ground terms in M of the form $f(s_1, \ldots, s_n)$, the algorithm proceed to solve the (simultaneous) unification problem $\{t_1 =^? s_1, \ldots, t_n =^? s_n\}$. The standard unification algorithm fails when it encounters the case $g(\overline{t}) =^? g'(\overline{s})$ where g and g' are distinct symbols. CVC3, however, tries to do more. If $g(\overline{t})$ is ground and $g(\overline{t}) =_E g'(\overline{s})$, then the case $g(\overline{t}) =_E g'(\overline{s})$ is solved and removed. If g is an uninterpreted symbol and there is a term of the form $g(\overline{u})$ in M such that $g'(\overline{s}) =_E g(\overline{u})$, then CVC3 proceeds to solve $\overline{t} =^? \overline{u}$.

As a simple example, consider matching a trigger like $f(h(x))$ with a ground term $f(a)$ where f, h, a are uninterpreted symbols and x is a variable. Suppose that $a = h(s) \in E$ for some s. Then the procedure above can match $f(h(x))$ and $f(a)$ with x bound to s.

3.3.5 Implementation

Figure 3.2 shows the pseudo-code of CVC3's E-matching algorithm.

The argument *binding* is a partial function that maps variables to ground terms. If v is a variable, the result of applying *binding* to v is denoted by *binding*[v] and v is *bound* in *binding* if *binding*[v] exists. $s =_E t$ means terms s and t are equal modulo the congruence closure of equalities. CVC3 implements an efficient date structure to maintain the congruence closure of equalities and

it is almost constant time to check $s =_E t$. *gterm*[i] means the i-th child of *gterm*.

The main matching function is `recMultMatch`. It takes three parameters: a ground term *gterm*, a trigger *vterm*, and a binding *binding*. The goal of the function is to match *gterm* with *vterm* in a way that is consistent with the given *binding*. It returns a set of bindings.

If *vterm* is a variable and *vterm* is bound in *binding*, then matching can only succeed if *binding*[*vterm*] and *gterm* are equivalent modulo equality. Otherwise, if *vterm* is a variable and *vterm* is not bound in *binding*, then *binding* is extended to map *vterm* to *gterm*.

If *vterm* is ground, then the matching can be succeed if *vterm* is equivalent to *gterm*.

If *vterm* is neither a variable nor ground, then it must be a function application. The function `equivalent(`*gterm*`,` *vterm*`)` returns all terms that are equivalent to *gterm* modulo the set of equalities and begin with the same symbol as *vterm*. For example, if the set of equalities contains $a = f(b)$ and $a = g(c)$, then `equivalent(`a`,` $g(d)$`)` includes $g(c)$. For each term in `equivalent(`*gterm*`,` *vterm*`)`, the algorithm proceeds to match the children.

The children are matched using the function `multMatchChild`, which takes

```
FUNCTION recMultMatch(gterm, vterm, binding)
 IF (vterm is a variable)
 THEN IF (vterm is bound in binding)
      THEN IF (gterm =_E binding[vterm])
           THEN RETURN { binding };
           ELSE RETURN  ∅;
      ELSE
           binding[vterm] := gterm;
           RETURN { binding };
 ELSE IF (vterm is ground)
      THEN IF (gterm =_E vterm)
           THEN RETURN { binding };
           ELSE RETURN  ∅;
      ELSE
           allGterms := equivalent(gterm, vterm);
           newBindings := ∅;
           FOR_EACH (g in allGterms)
               newBindings := newBindings
                                ∪ multMatchChild(g, vterm, binding);
           RETURN newBindings;


FUNCTION multMatchChild(gterm, vterm, binding)
 newBindings := { binding };
 FOR (i := 1 TO gterm.arity())
     nextBindings := ∅;
     FOR_EACH (binding in newBindings)
         nextBindings := nextBindings ∪
                           recMultMatch(gterm[i], vterm[i], binding);
         newBindings := nextBindings;
 RETURN newBindings;
```

Figure 3.2: Matching algorithm

a ground term *gterm*, trigger *vterm*, and binding *binding*. *gterm* and *vterm* must begin with the same function symbol. This function iterates through each child and builds up a set of bindings. The bindings returned from the result of matching the i-th children are considered as candidate bindings for matching the $i+1$-th children.

3.3.6 Heuristics and optimizations

When a term t of the form $t_1 < t_2$ or $t_1 \leq t_2$ is matched with ground term p in M of the form $s_1 < s_2$ or $s_1 \leq s_2$, CVC3 generates the E-matching problem $\{t_1 =^? s_2, t_2 =^? s_1\}$ if t has positive polarity and literal p appears in M, or t has negative polarity and literal $\neg p$ occurs in M; otherwise it generates the problem $\{t_1 =^? s_1, t_2 =^? s_2\}$. The motivation of this heuristic is best explained in the following example. Suppose M contains the following abstract literals:

1. $\forall x, y.\, (\neg x < y \vee f(x) < f(y))$
2. $a < b$
3. $f(b) < f(a)$

If $f(b) < f(a)$ is directly matched with $f(x) < f(y)$, the result will be $b < a \rightarrow f(b) < f(a)$, and no further matching is possible and no contradiction

can be deduced. If, however, $f(a) < f(b)$ is matched with $f(x) < f(y)$, the resultant instantiation will be $a < b \rightarrow f(a) < f(b)$, and then $(a < b)$, $(a < b \rightarrow f(a) < f(b))$, and $f(b) < f(a)$ constitutes a contradiction. Intuitively, matching a trigger with a ground term with the opposite polarity may help more for proving unsatisfiability.

When used within the DPLL(T) architecture, the matching algorithm will be invoked a huge number of times. Given n ground terms and m triggers, a naive approach is to do matching for all mn pairs of ground terms and triggers. CVC3 improves on this approach as follows. At all times, a map is maintained that maps each function symbol to the list of triggers beginning with that symbol. When a ground term a with top symbol f is to be matched, each trigger in the list with top symbol f is matched with a. More sophisticated techniques for matching multiple triggers at the same time are described in [21].

A trigger is *simple* if all its proper sub-terms are variables. For example, $f(x, y)$ is simple, where x and y are variables. When matching a ground term with a simple trigger, say $f(x, y)$, as long as the ground term's top symbol is f, the matching is always successful. CVC3 keeps track of simple triggers and avoids calling the matching algorithm on them.

Two triggers are *α-equivalent* if they can be rewritten into each other by renaming of the variables. For example, $f(x, a)$ and $f(y, a)$ are α-equivalent, where a is a ground term and x and y are variables. CVC3 detects triggers that are α-equivalent and only matches against a single representative trigger for each such set of triggers.[3]

Equalities can cause redundant matchings. For instance, if $a = g(b)$ holds, then matching trigger $f(g(x))$ with the ground terms $f(a)$ and $f(g(b))$ will produce the same result. CVC3 maintains a unique equivalence class representative for each equivalence class induced by the set of equalities in M. The *signature* of a term is the result of replacing each of its children with its equivalence class representative. During matching, CVC3 uses only one ground term from a set of terms with the same signature.

Because CVC3 employs eager instantiation, after a round of instantiation, there may be more splitting and instantiation. CVC3 keeps track of which terms and triggers have been matched along the current branch of the search tree so that these matches are not repeated. However, a newly asserted equality may result in more opportunities for matching a previously attempted trigger

[3]This is equivalent to using term indexing for bound variables, as is common in many general first order provers.

and ground term pair. For example, suppose trigger $f(a, x)$ is matched with $f(g(b), c)$. If a is not known to be equivalent to $g(b)$, the matching algorithm yields nothing. However, suppose that later along the same branch of the DPLL search tree, $a = g(b)$ is asserted. Then $f(a)$ and $f(g(b))$ can be matched.

One approach for handling this problem is to use the *inverted path tree* data structure introduced in [21]. CVC3 employs a simple alternative solution. The basic idea is to periodically retry pairs of ground terms and triggers that did not match before. Since this operation is expensive, CVC3 does it lazily, that is, only when all other heuristics fail to produce a contradiction. In addition, CVC3 only considers pairs involving a ground term for which one or more of its proper sub-terms appears in an equivalence class that has changed. This achieves a similar effect as Simplify's *mod-time* heuristic [24]. Notice that simple triggers never need to be re-tried.

3.3.7 Special heuristics

In addition to E-matching, CVC3 also employs some specialized instantiation heuristics that have proven useful on formulas that appear in SMT-LIB.

One heuristic will set up a special multi-trigger for formulas of the form $\forall x, y : P(x, y) \wedge P(y, x) \rightarrow (x = y)$. According the rules so far, a normal trigger

$P(x, y)$ will be set up. However, This formula basically says that predicate P is anti-symmetric, and a multi-trigger $\{P(x, y), P(y, z)\}$ is more efficient.

Another heuristic is for formulas that involve array operations. When array operations are present, say array read *read* or array write *write* appear in M, in addition to the usual matching and instantiation, all ground terms appearing in the position of an array index will be picked out and used for instantiating variables that appear as array index. For example, suppose the formula $\forall x : \ldots write(a, x, v) \ldots$ and term $read(a, i, v)$ appear in M, where i is a ground term. After normal matching and instantiation, i will be used to instantiate the variable x.

3.3.8 Trigger matching by instantiation levels

For a large class of quantified formulas from verification applications, the formulas are of the form $\Gamma \wedge \neg\varphi$ where φ is a verification condition and Γ is a large and more or less fixed T-satisfiable collection of (quantified) formulas. Γ usually formalizes the relations and functions that are relevant to the verification application. As a result, a large number of the formulas in Γ typically have no bearing on whether φ is T-satisfiable together with Γ. An SMT solver can easily spend too many resources instantiating these unrelated formulas.

The basic idea to deal with this problem is to give all quantified formulas a fair chance for matching. Simplify uses a *matching depth* heuristic to address this problem. Every clause is assigned a value, the *depth*, that is initially 0. The *current depth* is the highest depth among all clauses on the current branch in the DPLL search tree. When a new instantiation is generated, it is assigned a value that is one greater than the current depth in the previous round of instantiation. Later, when splitting is needed in DPLL, a literal from clauses with a lower matching depth will be favored. A limit on matching depth is also used to determine when to give up and terminate.

To acheive these same goals, CVC3 uses a different approach, better suited to the DPLL(T) structure. Modern SAT solvers employ sophisticated heuristics to choose the literal for splitting. Though not impossible, it would be inconvenient and could likely cause performance loss to customize a heuristic that incorporates the idea of matching depth in the SAT solver used by an SMT solver. Also, theory reasoning can introduce splitting [7] which further complicates the situation. Instead of assigning a value to clauses, CVC3 assigns an *instantiation level* to every ground term. Intuitively, a ground term has an instantiation level n if it is the result of n instantiations. All ground terms in the original formula are given an instantiation level of 0. If a formula

$\forall x.\, \varphi$ is instantiated with the ground term t with an instantiation level of n, then all the new terms in $\varphi[x/t]$ (as well as any new terms derived from them via theory reasoning) are given the instantiation level $n + 1$. When matching, CVC3 matches triggers only against ground terms with instantiation level less than or equal to an upper bound b. The initial value of the upper bound is 0. When CVC3 reaches a non-fail state after all rules have been tried, i.e. a state from which no contradiction can be deduced, the upper bound b is increased by one to allow more ground terms for matching.

The instantiation level heuristic has proved very effective in the experiments, discussed in the next section. It also neutralizes the possible harmful effects of instantiation loops in the eager instantiation strategy. The reason is simply that a new ground term resulting from an instantiation will not be considered until the upper bound b is increased. The value of b will only be increased after all the terms with instantiation level less or equal to b have been considered for matching and instantiation. Therefore, checking for instantiation loops, either statically or dynamically, is completely unnecessary. Moreover, the instantiation level heuristic allows some triggers that otherwise would be disabled by static or dynamic loop detection. As discussed before, such triggers are actually necessary to prove many examples.

The instantiation level also achieves a balance between eager and lazy instantiation. Quantified formulas can be instantiated eagerly as soon they appear in M. However, further matching and instantiation with the new ground terms will be delayed until the upper bound is increased, a somewhat lazy method. As will be shown in Section 3.4, some benchmarks favor lazy instantiation and some favor eager instantiation, and the best result is obtained when the instantiation level heuristic is used.

For some applications, an SMT solver is called frequently and thus a fast result from the SMT solver is preferred. When heuristic-based instantiation is employed, the solver can spend a lot of time on satisfiable cases that cannot be proved by the heuristic instantiation method. Therefore, some termination criterion is needed, and the upper bound used in the instantiation level heuristic provides a natural choice for this purpose. The SMT solver can choose to abandon the search after the upper limit reaches a pre-determined value. For certain classes of problems, this pre-determined valued can be obtained from experiments.

Some theorem provers based on instantiation-based first-order calculi [57, 47, 9] also use fair instantiation strategies based on assigning values to terms, which is needed to guarantee refutational completeness. Many of these provers

instantiate variables with terms whose *depth* is below a progressively larger bound, where a term's depth is measured as the depth of the term's abstract syntax tree, or in some other equivalent way. While simpler to implement than our instantiation level strategy, an instantiation strategy based on term depth is not suitable in our case because it does not guarantee fairness in a SMT solver. The main problem is that SMT solvers employ theory solvers that may simplify a term. For example, $(a-1)-1$ may be simplified to $a-2$. Therefore, the syntax-based depth cannot guarantee fairness at all. Another reason is that in CVC3, an unbounded number of Skolem constants may be generated as a result of applying the $\exists$-Inst rule to new formulas generated by the $\forall$-Inst rule, which means an unbounded number of ground terms of the same depth.

3.3.9 Implementation details

Although the idea of instantiation level looks rather simple to implement, it is not easy for SMT solvers like CVC3 that combine several theory solvers. The major obstacle is that new ground terms can be created by theory solvers and it is difficult to assign an instantiation level to such new terms. One possible way out is to modify the decision procedures in theory solvers and require that

an instantiation level is assigned whenever a new term is created. However, this is not always feasible. For example, when a new term is created because of the existence of some other terms, then the term can be assigned the highest instantiation level of these other terms. Unfortunately, some new terms are created without involving any other terms

CVC3 depends on its proof system to assign instantiation levels to terms. In CVC3 a *theorem* consists of a *conclusion* and a set of *assumptions*. An assumption can be the conclusion of another theorems. The *proof* of a theorem in CVC3 can be seen as a tree with the conclusion as the root. It is required that all formulas added into F must be conclusions of theorems. A nice feature of CVC3 is that all theorems created by decision procedures are valid ones independent of M. In other words, if the instantiation of a quantified formula results in a new theorem created by a decision procedure, the instantiation will appear in the proof tree of the new theorem. More details of the proof system will be discussed in Chapter 5.

CVC3 assigns an instantiation level to every theorem. CVC3 keeps a map from terms to theorems. When a new term n is introduced because it appears in the conclusion of a newly added theorem T, an entry from n to T is inserted into the map and the instantiation level of n is set to be the instantiation level

of T. In the beginning, all theorems have instantiation level of 0. When a new theorem T is created, T's assumptions are checked. Because assumptions are just terms, the instantiation levels of T's assumptions are compared and the highest one is set as the instantiation level of T. As an exception, if a theorem creates an new instantiation and the highest instantiation level of its assumptions is h, then the instantiation level of the theorem is set to be $h+1$.

3.4 Experimental results

The section discusses a set of experiments that shows the improvement in CVC3. These results have been reported in [34].

The experiments were run for CVC3 version 1.1. The performance of these heuristics are evaluated both within CVC3 and in comparison with other theorem provers and SMT solvers. Two leading automated theorem provers (ATPs) for first-order logic compared are Vampire 8.1 [58] and SPASS 2.2 [67]. Three SMT solvers supporting quantified formulas compared are Simplify, Yices 1.0, and Fx7[4].

These results in the section represent the state-of-the-art as of May 2007.

[4]The version of Fx7 was the one available at `http://nemerle.org/~malekith/smt/en.html` as of February 2007.

Some results comparing more recent versions of some SMT solvers can be found on the SMT competition website: `http://www.smtcomp.org`. All tests were run under Linux on AMD Opteron-based (64 bit) systems. The timeout is 5 minutes (unless otherwise stated) and the memory limit is 1 GB.

3.4.1 Benchmarks

The benchmarks used are from the SMT-LIB benchmark library. The test set consists of 29,004 benchmarks from three different SMT-LIB *logics*: AUFLIA, AUFLIRA and AUFNIRA. Cases in AUFLIA have a background theory consisting of arrays, equalities and linear integer arithmetic [5]. AUFLIRA cases have a background theory consisting of arrays, equalities, and mixed linear integer and real arithmetic. The AUFNIRA cases have a background theory consisting of arrays, uninterpreted functions, and mixed non-linear integer and real arithmetic.

These benchmarks are further divided into families. In AUFLIA, there are five families: *Burns*, *misc* (in which we include a single benchmark originally in the *check* family), *piVC*, *RicartAgrawala*, and *simplify*. AUFLIRA consists

[5]It should be remarked that most of the benchmarks in AUFLIA make little or no use of the array theory.

of two families: *misc* and *nasa*. And AUFNIRA has a single family: *nasa*. For more information on the other benchmarks and on the SMT-LIB library, please refer to the SMT-LIB website: `http://www.smtlib.org`.

The *nasa* families make up the vast majority of the benchmarks with a total of 28,065 benchmarks in two families. These cases are *safety obligations* automatically generated from annotated programs at NASA. Following their introduction in [23], these benchmarks were made publicly available in TPTP format [62], a format for pure first-order logic.

These *nasa* benchmarks in the TPTP format were translated into the SMT format as follows. First, some assumptions, which were valid with regard to the background theory, were removed. These assumptions formalize the background theories for ATP systems and involve axioms for the theories of arrays and arithmetic. Since SMT solvers have built-in theory solvers, these assumptions are not needed. For example, an assumption asserts $succ(two) = three$, which means the successor of 2 is 3. If translated, it would be $2 + 1 = 3$, a formula any modern SMT solver can prove valid. Second, sorts are inferred for every symbol. The rules for sort-inference are

1. The index of an array is of integer sort;

2. The return sort of the functions *cos*, *sin*, *log*, *sqrt* is real;

3. The terms on both sides of infix predicates =, <=, >=, < and >, must have the same sort;

4. If the sort of a term cannot be deduced by the above rules, it is assumed to be real.

According to [23], of the 28,065 cases, only 14 are supposed to be satisfiable and the rest are unsatisfiable. However, after experimentation and careful examination of the benchmarks in their present form in the TPTP library, the best guess is that somewhere around 150 of the cases are actually satisfiable (both in the SMT-LIB format and in the original TPTP format). It is difficult to figure out which ones are indeed satisfiable for sure.

The other major family is the *simplify* family, which was translated (by others) from a set of over 2,200 benchmarks introduced in [24]. Only a selection of the original benchmarks were translated. According to the translator, benchmarks that were too easy or involved non-linear arithmetic [20] were excluded. There are 833 benchmarks in this family and all are unsatisfiable.

Lazy strategy		(i) BTBM		(ii) BTSM		(iii) STSM		(iv) IL	
Category	#cases	#unsat	time	#unsat	time	#unsat	time	#unsat	time
AUFLIA/Burns	12	**12**	0.013	12	0.013	12	0.014	12	0.020
AUFLIA/misc	14	10	0.010	14	0.022	**14**	0.021	14	0.023
AUFLIA/piVC	29	25	0.109	25	0.109	29	0.119	**29**	0.117
AUFLIA/RicAgla	14	14	0.052	**14**	0.050	14	0.050	14	0.050
AUFLIA/simplify	769	471	1.751	749	3.846	**762**	0.664	759	0.941
AUFLIRA/nasa	4619	**4113**	1.533	4113	1.533	4113	1.551	4113	1.533
AUFNIRA/nasa	142	46	0.044	**46**	0.043	46	0.043	46	0.044
Total	5599	4691	1.521	4973	1.849	**4990**	1.402	4987	1.409
Eager strategy		(i) BTBM		(ii) BTSM		(iii) STSM		(iv) IL	
Category	#cases	#unsat	time	#unsat	time	#unsat	time	#unsat	time
AUFLIA/Burns	12	**12**	0.012	12	0.020	12	0.019	12	0.019
AUFLIA/misc	14	10	0.008	12	0.013	12	0.013	**14**	0.047
AUFLIA/piVC	29	25	0.107	25	0.108	29	0.127	**29**	0.106
AUFLIA/RicAgla	14	14	0.056	14	0.058	14	0.056	**14**	0.041
AUFLIA/simplify	769	25	18.24	24	39.52	497	30.98	**768**	0.739
AUFLIRA/nasa	4619	4527	0.072	**4527**	0.071	4527	0.074	4526	0.014
AUFNIRA/nasa	142	**72**	0.010	72	0.010	72	0.011	72	0.012
Total	5599	4685	0.168	4686	0.273	5163	3.047	**5435**	0.117

Table 3.1: Lazy vs. eager instantiation strategy in CVC3.

3.4.2 Evaluating the heuristics

The first experiment examined naive instantiation (both the lazy and eager strategies) on all SMT-LIB benchmarks. Of 29,004 benchmarks, 23,389 can be solved in negligible time by both the eager and the lazy naive strategies. In fact, almost all of these can be solved without any quantifier reasoning at all. Obviously, these are not good benchmarks for testing instantiation strategies and they are excluded from the tables below.

For the remaining 5,599 benchmarks, the following instantiation strategies are tried:

- BTBM: basic trigger/matching algorithm with none of the heuristics described in Section 3.3. (i.e. no multi-triggers, syntactic matching only);
- BTSM: basic triggers with the smarter matching described in Section 3.3.4;
- STSM: smart triggers including multi-triggers as described in Section 3.3.1.
- IL: all the heuristics above plus the instantiation level heuristic.

The results are shown in Table 3.1. Each table lists the number of cases by family, the number of cases successfully proved unsatisfiable and the *average* time spent on these successful cases.

As can be seen, the basic matching strategy is quite effective on about 4/5 of the benchmarks. There are about 1,000 cases that cannot be solved without employing more sophisticated techniques.

Another observation is that the eager strategy generally outperforms the lazy strategy, both on average time and on number of cases proved, especially for the *nasa* cases. The exception is the *simplify* family, where the lazy strategy

performs much better except the last column. This is due to the fact that eager instantiation can easily fall into loops for the *simplify* cases. Since the lazy strategy is better at dealing with loops, it is much better than the eager strategy for the *simplify* cases.

3.4.3 Comparison with ATP systems

Table 3.2 compares CVC3 with Vampire, SPASS, and Simplify on the *nasa* benchmarks. For these tests, the timeout was 1 minute. Vampire was chosen because it is one of the best ATP system and has won several categories in the CASC competitions [61] in recent years. SPASS was chosen because it was the best solver tried in [23]. For easier comparison to the results shown in [23], the benchmarks are divided as in that paper into seven categories: $T_{\emptyset}$, $T_{\forall,\rightarrow}$, T_{prop}, T_{eval}, T_{array}, T_{policy}, T_{array*}. The first category $T_{\emptyset}$ contains the most difficult verification conditions. The other categories were obtained by applying various simplifications to $T_{\emptyset}$. For a detailed description of the categories and how they were generated, please refer to [23]. The 14 known satisfiable cases are excluded in this breakdown (as was also done in [23]), so there are 28051 benchmarks in total.

All solvers can prove most of the benchmarks, as most of them are easy.

		Vampire		SPASS		Simplify		CVC3	
Category	#cases	#unsat	time	#unsat	time	#unsat	time	#unsat	time
$T_{\emptyset}$	365	266	9.2768	302	1.7645	207	0.0679	**343**	0.0174
$T_{\forall,\rightarrow}$	6198	6080	2.1535	6063	0.6732	5957	0.0172	**6174**	0.0042
T_{prop}	1468	1349	4.3218	1343	1.0656	1370	0.0339	**1444**	0.0058
T_{eval}	1076	959	5.6028	948	0.7601	979	0.0423	**1052**	0.0077
T_{array}	2026	2005	1.4438	2000	0.2702	1943	0.0105	**2005**	0.0048
T_{array*}	14931	14903	0.6946	14892	0.2323	14699	0.0101	**14905**	0.0035
T_{policy}	1987	1979	1.4943	1974	0.2716	1917	0.0101	**1979**	0.0050
Total	28051	27541	1.5601	27522	0.4107	27072	0.0145	**27902**	0.0043

Table 3.2: ATP vs SMT

The ATP systems can solve more cases than Simplify, while Simplify is generally much faster. CVC3 outperforms the other systems in both time and number of cases solved. There are only 149 cases that CVC3 cannot solve (as mentioned earlier most of these are suspected to be actually satisfiable). For the most challenging category $T_{\emptyset}$, CVC3 was able to solve 343 out of 365 cases, significantly more than the ATP systems. At the time these tests were done, this was the best result ever achieved on these benchmarks.

3.4.4 Comparison with other SMT systems

At the time of the experiments, only two other known SMT systems included support for both quantifiers and the SMT-LIB format: yices [26] and Fx7 [53]. Yices was the winner of SMT-COMP 2006, dominating every category. Fx7 was a new system recently developed by Michal Moskal. Fx7 uses quantifier

		Fx7		yices		CVC3	
Category	#cases	#unsat	time	#unsat	time	#unsat	time
AUFLIA/Burns	12	12	0.4292	**12**	0.0108	12	0.0192
AUFLIA/misc	14	12	0.6817	**14**	0.0500	14	0.0479
AUFLIA/piVC	29	15	0.5167	**29**	0.0300	29	0.1055
AUFLIA/RicAgla	14	14	0.6400	**14**	0.0257	14	0.0407
AUFLIA/simplify	769	760	3.2184	740	1.4244	**768**	0.7386
AUFLIRA/nasa	4619	4187	0.4524	4520	0.0824	**4526**	0.0138
AUFNIRA/nasa	142	48	0.4102	N/A	N/A	**72**	0.0118
Total	5599	5048	0.8696	5329	0.2681	**5435**	0.1168

Table 3.3: Comparison of SMT systems

instantiation techniques that are similar to those used in Simplify, with some extensions [55].

Table 3.3 compares Fx7, yices, and CVC3. While yices is sometimes faster than CVC3, CVC3 can prove as many or more cases in every category. In total, CVC3 can prove 34 more cases than yices (the AUFNIRA cases, which Yices does not support, are not counted). Also, CVC3 is significantly faster on the *simplify* and *nasa* benchmarks.

It is natural to compare CVC3 to Simplify on the *simplify* benchmarks. Not surprisingly, Simplify can solve all of these benchmarks very fast, and it can solve all 2,251 benchmarks in its suite in 469.05 seconds, much faster than both yices and CVC3. Simplify achieves these impressive results by relying on special annotations, *manual triggers*, that instruct Simplify on which triggers to use. If the manual triggers are removed, Simplify can only prove 444 of the

original 2251 benchmarks. Of course, it is a bit unfair to compare Simplify with the manual triggers removed because both Simplify and the benchmarks were crafted under the assumption that manual triggers would be used. On the other hand, the results on the *nasa* benchmarks show that the heuristics used by CVC3 can effectively solve verification conditions without manual triggers, and the ability to prove these benchmarks *automatically* and without annotations represents a significant step forward for SMT solvers.

Ideally, the experiments would include Simplify's results on all of the SMT-LIB benchmarks. Unfortunately, Simplify does not read the SMT-LIB format and the translation from SMT-LIB to Simplify's language is non-trivial as it involves moving from a sorted to an unsorted language.

Chapter 4

Complete instantiation

Although heuristic instantiation is relatively effective for some software verification applications [5, 30], it suffers from several problems. The major problem is incompleteness. Additionally, some heuristics are sensitive to the syntax, which require the users to have a deep understanding of the solver.

For some fragments of first order logic with background theories, it is possible to have complete decision procedures based on instantiation. An example is the *array property fragment* proposed by Bradley, et al. [10]. A quantified formula F in the array property fragment can be shown to be equi-satisfiable to a conjunction of a finite number of instantiations of F, which in turn can be easily decided by an SMT solver.

The instantiation-based approaches are attractive because they can be easily integrated into existing SMT solvers that employ efficient decision procedures for ground formulas in many useful theories.

This chapter proposes a series of decidable fragments of first-order logic with background theories, and proposes an instantiation-based complete decision procedure for them. Most of the results in this chapter has been reported in [35].

To prove a quantified formula is equi-satisfiable to a ground formula, the key point is to show how to construct a model for the quantified formula when the corresponding ground formula is satisfiable.

The first fragment discussed is the essentially uninterpreted fragment, in which quantified variables can only appear as arguments of uninterpreted functions and predicates. The terms used for instantiation come from the least solution of a system of constraints over sets of ground terms. A projection function is defined and used to show how to construct the models for quantified formulas. Later, the essentially uninterpreted fragment is extended to allow variables to appear in more places.

Some notation is as follows. For a term t, $t[x_1, \ldots, x_n]$ denotes that t may contain the variables x_1,x_2,..., x_n, and $t[r_1, \ldots, r_n]$ denotes the result of

simultaneously substituting r_i for x_i $(1 \leq i \leq n)$ in t. If S_i $(1 \leq i \leq n)$ are sets, $t[S_1, \ldots, S_n]$ denotes the set $\{t[r_1, \ldots, r_n] \,|\, r_1 \in S_1, \ldots, r_n \in S_n\}$. For a clause C, $C[r_1, \ldots, r_n]$ and $C[S_1, \ldots, S_n]$ are defined in the obvious way, where r_i are terms and S_i are sets of terms. In this chapter, without loss of generality, it is assumed that the formula F being checked is represented as a set of CNF clauses where C_k denotes the k-th clause. Variables in each clause are universally quantified. The i-th variable in a clause is denoted by x_i. If M is a structure, then $M(S)$ denotes $\{t^M \,|\, t \in S\}$, where S is a set of ground terms.

4.1 Herbrand theorem

Given a quantified formula F in pure first-order logic, the Herbrand universe [28] in the standard Herbrand Theorem is defined as the set of terms that can be constructed by using the function symbols appearing in F. For example, suppose a formula contains a binary function symbol g, an unary function symbol f, and a constant a, then the Herbrand universe is

$\{a, f(a), g(a,a), f(f(a)), f(g(a,a)), g(a,f(a)), g(a,g(a,a)) \ldots\}$.

The standard Herbrand Theorem states that a formula is satisfiable if and

only if it is satisfiable in a structure whose domain is a Herbrand Universe for that formula. In other words, formula $\forall \overline{x}.\psi$ is satisfiable if and only if the set of formula $\{\psi[\overline{x}/\overline{t}] \,|\, \overline{t} \in H^n\}$ is, where H is the set of terms in the Herbrand Universe, n is number of variables in $\overline{x}$ and H^n is the Cartesian product over H. If the Herbrand Universe is finite, then obviously the quantified formula is decidable.

In particular, if there are no function symbols, then the Herbrand Universe is finite, which means the formula is decidable. This is the Bernays-Schönfinkel-Ramsey class. If there is at least one function symbol, then the Herbrand Universe will be infinite.

This chapter will show that, under certain conditions, it is sufficient to check the satisfiability of $\{\psi[\overline{x}/\overline{t}] \,|\, \overline{t} \in S^n\}$ for a finite subset S of H. One such sufficient condition is the stratified condition, which will be studied later in this chapter.

4.2 Essentially uninterpreted formulas

Suppose we are going to check the satisfiability of formula F that contains both interpreted and uninterpreted functions. A formula F is *essentially un-*

interpreted if any variable in F appears only as an argument of uninterpreted functions or predicates.

Example 4.2.0.1 essentially uninterpreted clause

$$f(g(x_1) + a) \leq h(x_1) \vee p(f(x_1) + b, x_2)$$

4.2.1 Ground terms for instantiation

The ground terms used for instantiation are obtained from the least solution of a system of constraints over sets of terms. These constraints over sets are defined as follows.

For each variable x_i in every clause C_k, let $S_{k,i}$ be a set of ground terms. For each n-ary uninterpreted function or predicate symbol f, let $A_{f,1}$, $A_{f,2}$, ..., $A_{f,n}$ be sets of ground terms. $S_{k,i}$ and $A_{f,n}$ are obtained as the least solution to a system of constraints. Intuitively, $S_{k,i}$ contains the ground terms for instantiating variable x_i in clause C_k, and $A_{f,n}$ contains ground terms that can appear as the n-th argument of f. For simplicity and without lose of generality, only functions will be discussed in this chapter, except at a few places.

Given a formula F, the system of constraints Δ_F is defined to contain the following constraints for each t that is the j-th argument of f in C_k.

- $t \in A_{f,j}$ if t is a ground term.
- $t[S_{k,1}, \ldots, S_{k,n}] \subseteq A_{f,j}$ if t is of the form $t[x_1, \ldots, x_n]$.
- $S_{k,i} \equiv A_{f,j}$ if t is the i-th variable x_i.

The first rule says that if the j-th argument of f in C_k is a ground term, then it should be in $A_{f,j}$. The second rule says that if $t[x_1, \ldots, x_n]$ appears as the j-th argument of f, then $A_{f,j}$ should contain all terms in the set $t[S_{k,1}, \ldots, S_{k,n}]$. The last rule says that if the j-th argument is a variable x_i, then $S_{k,i}$ should be equal to $A_{f,j}$.

It is required that each set $S_{k,i}$ and $A_{f,n}$ contains at least one ground term. This can always be done by adding a fresh constant to each set.

Notice that in an essentially uninterpreted formula, for each $S_{k,i}$ there will always be an equation $S_{k,i} \equiv A_{f,j}$ in Δ_F.

To illustrate the construction of Δ_F, consider the following example.

Example 4.2.1.1 Δ_F construction

Let F be the following four clauses.

$g(x_1, x_2) = 0 \vee h(x_2) = 0,$

$g(f(x_1), b) + 1 \leq f(x_1),$

$h(b) = 1,$

$f(a) = 0$

Δ_F is:

$S_{1,1} \equiv A_{g,1}, \quad S_{1,2} \equiv A_{g,2}, \quad S_{1,2} \equiv A_{h,1}$

$S_{2,1} \equiv A_{f,1}, \quad b \in A_{g,2}, \quad f(S_{2,1}) \subseteq A_{g,1}$

$b \in A_{h,1},$

$a \in A_{f,1}$

The least solution of Δ_F is $S_{1,1} \equiv A_{g,1} \equiv \{f(a)\}$, $S_{1,2} \equiv A_{g,2} \equiv A_{h,1} \equiv \{b\}$, $S_{2,1} \equiv A_{f,1} \equiv \{a\}$.

Define F^* as the set of ground clauses $\{C_k[S_{k,1}, \ldots, S_{k,m}] \,|\, C_k \text{ in } F\}$. That is, F^* is obtained by instantiating clauses with ground terms from $S_{k,i}$. For the above example, F^* is

$g(f(a), b) = 0 \vee h(b) = 0,$

$g(f(a), b) + 1 \leq f(a),$

$h(b) = 1,$

$f(a) = 0$

We claim that F is T-satisfiable if and only if F^* is T-satisfiable. Before proving this, we need some additional definitions and lemmas.

4.2.2 From M to M^π

Suppose F^* is T-satisfiable and has a model M. M is usually not a model for F. However, a model M^π of F can be constructed based on M. In other words, if F^* is T-satisfiable, so is F.

To construct M^π, some projection functions are needed. For each $A_{f,j}$, a projection function $\pi_{f,j} : |M| \rightarrow |M|$ is a function such that for every $e \in |M|$, $\pi_{f,j}(e) \in M(A_{f,j})$ and $\pi_{f,j}(e) \equiv e$ when $e \in M(A_{f,j})$. For convenience, when f is an unary function, π_f denotes $\pi_{f,1}$ and A_f denotes $A_{f,1}$.

Similarly a projection function $\pi_{k,i}$ is defined for each $S_{k,i}$. If $S_{k,i} \equiv A_{f,j}$ then the function $\pi_{k,i}$ is equals to to function $\pi_{f,j}$.

The functions $\pi_{f,j}$ and $\pi_{k,i}$ are well-defined because $A_{f,j}$ and $S_{k,i}$ are not

empty.

Let $\pi_k(\overline{a})$ denote the tuple $\langle \pi_{k,1}(a_1), \ldots, \pi_{k,m}(a_m) \rangle$.

M^π is defined as a model satisfying the following conditions:

- $|M^\pi| \equiv |M|$
- $c^{M^\pi} \equiv c^M$ for every constant c
- $f^{M^\pi} \equiv f^M$ for every interpreted function f
- $P^{M^\pi} \equiv P^M$ for every interpreted predicate P
- $f^{M^\pi}(e_1, \ldots, e_n) \equiv f^M(\pi_{f,1}(e_1), \ldots, \pi_{f,n}(e_n))$ for every uninterpreted function f
- $P^{M^\pi}(e_1, \ldots, e_n) \equiv P^M(\pi_{p,1}(e_1), \ldots, \pi_{p,n}(e_n))$ for every uninterpreted predicate P

As mentioned earlier in the chapter, only functions will be discussed except when necessary.

M^π is called a π-*extension* of M. M^π differs from M only in the way that uninterpreted functions are interpreted. Clearly M^π is also a model of T.

4.2.3 Interpretations of ground terms in M and M^π

Proposition 4.2.3.1

For every ground term $f(\ldots, t_j, \ldots)$ in F^*, where t_j is the j-th argument and f is uninterpreted, t_j is in $A_{f,j}$.

Proof. If t_j appears in F, then it is in $A_{f,j}$ because the rules for $A_{f,j}$ enforce it.

If t_j does not appear in F, then it is must be result of instantiation. By construction of F^*, it must be in $A_{f,j}$. □

The next lemma shows that M and M^π give the same interpretation for ground terms appearing in F^*.

Lemma 4.2.3.1

For every ground term t that appears in F^*, $t^M \equiv t^{M^\pi}$.

Proof. By induction on the structure of t.

- Suppose t is a constant, then $t^{M^\pi} \equiv t^M$ by the definition of M^π.
- Suppose t is a function application. Without loss of generality, assume t is of the form $f(s)$. By the induction hypothesis, $f^{M^\pi}(s^{M^\pi}) \equiv f^{M^\pi}(s^M)$. If f is interpreted, then $f^{M^\pi} \equiv f^M$ and thus $f^{M^\pi}(s^{M^\pi}) \equiv f^M(s^M)$. If f

is uninterpreted, then by definition of M^π, $f^{M^\pi}(s^M) \equiv f^M(\pi_f(s^M))$. By the previous proposition, $s \in A_f$ and $s^M \in M(A_f)$. Therefore $\pi_f(s^M) \equiv s^M$. It follows that $f^M(\pi_f(s^M)) \equiv f^M(s^M)$.

□

Lemma 4.2.3.1 implies that M^π is also a model of F^*.

4.2.4 Interpretations of terms in M and M^π

Proposition 4.2.4.1

For a ground term t, $t^{M^\pi\{\overline{x}\mapsto\overline{e}\}} \equiv t^{M\{\overline{x}\mapsto\pi_k(\overline{e})\}} \equiv t^M$.

Proof. By lemma 4.2.3.1, $t^{M^\pi} \equiv t^M$. Since t is a ground term and does not contain any variables, $t^{M^\pi\{\overline{x}\mapsto\overline{e}\}} \equiv t^{M^\pi}$ and $t^M \equiv t^{M\{\overline{x}\mapsto\pi_k(\overline{e})\}}$. Therefore, the proposition holds. □

Lemma 4.2.4.1

For any non-variable term $t[\overline{x}]$ in clause C_k and any tuple $\overline{e} \in |M|^n$, $t[\overline{x}]^{M^\pi\{\overline{x}\mapsto\overline{e}\}} \equiv t[\overline{x}]^{M\{\overline{x}\mapsto\pi_k(\overline{e})\}}$.

Proof. By induction on the structure of t.

If t is a ground term, then the lemma holds by proposition 4.2.4.1.

Suppose $t[\overline{x}]$ is a function application. Without loss of generality, assume it is of the form $f(s[\overline{x}])$. The lemma becomes $f^{M^\pi}(s[\overline{x}]^{M^\pi\{\overline{x}\mapsto\overline{e}\}}) \equiv f^M(s[\overline{x}]^{M\{\overline{x}\mapsto\pi_k(\overline{e})\}})$. Note that the interpretation of a function f does not depend on the interpretation of variables.

If f is interpreted, then by the definition of essentially uninterpreted formulas, $s[\overline{x}]$ is not a variable. Therefore, by the induction hypothesis $s[\overline{x}]^{M^\pi\{\overline{x}\mapsto\overline{e}\}} \equiv s[\overline{x}]^{M\{\overline{x}\mapsto\pi_k(\overline{e})\}}$. As the definition of M^π, $f^{M^\pi}(x) \equiv f^M(x)$ when f is an interpreted function. Thus $f^{M^\pi}(s[\overline{x}]^{M^\pi\{\overline{x}\mapsto\overline{e}\}}) \equiv f^M(s[\overline{x}]^{M\{\overline{x}\mapsto\pi_k(\overline{e})\}})$, and the lemma holds.

If f is uninterpreted, then by definition of M^π, we have $f^{M^\pi}(s[\overline{x}]^{M^\pi\{\overline{x}\mapsto\overline{e}\}}) \equiv f^M(\pi_f(s[\overline{x}]^{M^\pi\{\overline{x}\mapsto\overline{e}\}}))$.

Now, it suffices to show that $\pi_f(s[\overline{x}]^{M^\pi\{\overline{x}\mapsto\overline{e}\}}) \equiv s[\overline{x}]^{M\{\overline{x}\mapsto\pi_k(\overline{e})\}}$. There are three cases:

1. If $s[\overline{x}]$ is ground, then by proposition 4.2.3.1, $s[\overline{x}]^{M^\pi\{\overline{x}\mapsto\overline{e}\}} \equiv s[\overline{x}]^{M\{\overline{x}\mapsto\pi_k(\overline{e})\}}$. Because $s[\overline{x}]$ is in $A_{f,1}$, $\pi_f(s[\overline{x}]^{M^\pi\{\overline{x}\mapsto\overline{e}\}}) \equiv s[\overline{x}]^{M^\pi\{\overline{x}\mapsto\overline{e}\}}$. Therefore, the lemma holds.

2. $s[\overline{x}]$ is a variable x_i and x_i is interpreted as the e_i in the tuple $\overline{e}$. By the definition of π_f, $\pi_f(x_i^{M^\pi\{\overline{x}\mapsto\overline{e}\}}) \equiv \pi_f(e_i)$. By the construction of sets A_f

and $S_{k,i}$, because x_i is an argument of an f, Δ_F contains the constraint $A_f \equiv S_{k,i}$. Consequently $\pi_f = \pi_{k,i}$. Therefore, $\pi_f(e_i) \equiv \pi_{k,i}(e_i) \equiv x_i^{M\{\overline{x} \mapsto \pi_k(\overline{e})\}}$.

3. $s[\overline{x}]$ is a non-ground term. By the induction hypothesis, $\pi_f(s[\overline{x}]^{M^\pi\{\overline{x} \mapsto \overline{e}\}}) \equiv \pi_f(s[\overline{x}]^{M\{\overline{x} \mapsto \pi_k(\overline{e})\}})$. Suppose $\overline{x}$ is the tuple $\langle x_1, \ldots, x_m \rangle$, and $\overline{x}$ is interpreted as $\overline{e}$. Let $\pi_k(\overline{e})$ be $\langle \pi_{k,1}(e_1), \ldots, \pi_{k,m}(e_m) \rangle$. In $M\{\overline{x} \mapsto \pi_k(\overline{e})\}$ every variable x_i is interpreted as $\pi_{k,i}(e_i) \equiv M(r_i)$, for some ground term r_i in $S_{k,i}$. Thus, $\pi_f(s[\overline{x}]^{M\{\overline{x} \mapsto \pi_k(\overline{e})\}}) \equiv \pi_f(s[\overline{r}]^M)$. The ground term $s[\overline{r}]$ must be in A_f because Δ_F contains the constraint $s[S_{k,1}, \ldots, S_{k,m}] \subseteq A_f$. Therefore, $\pi_f(s[\overline{r}]^M) \equiv s[\overline{r}]^M \equiv s[\overline{x}]^{M\{\overline{x} \mapsto \pi_k(\overline{e})\}}$.

□

Theorem 4.2.4.1

F and F^* are equi-satisfiable in T.

Proof. If F^* is unsatisfiable, then so is F, since F^* is a set of ground instances of F.

Suppose that F^* is T-satisfiable, but F is not. Let M be a T-model for F^* and M^π is the π-extension of M. Since F is unsatisfiable, there is a clause $C_k[\overline{x}]$ in F such that for some $\overline{e}$, $M^\pi\{\overline{x} \mapsto \overline{e}\} \not\models C_k[\overline{x}]$. By Lemma 4.2.4.1,

$M\{\overline{x} \mapsto \pi_k(\overline{e})\} \not\models C_k[\overline{x}]$. Let $\overline{e}$ be the tuple $\langle e_1, \ldots, e_m \rangle$, then for every $\pi_{k,i}(e_i)$ in $\pi_k(\overline{e})$, there is some ground term r_j in $S_{k,j}$ such that $r_j^M \equiv \pi_{k,j}(e_j)$. Thus, $M\{\overline{x} \mapsto \pi_k(\overline{e})\} \models C_k[\overline{x}]$ if and only if $M \models C_k[\overline{r}]$, and consequently $M \not\models C_k[\overline{r}]$, contradicting the assumption that M satisfies F^* since $C_k[\overline{r}]$ is in F^*. □

4.2.5 Finite essentially uninterpreted formulas

A formula F is in the *finite essentially uninterpreted fragment* (FEU) if every $S_{k,i}$ is finite in the least solution of Δ_F. A formula in the finite essentially uninterpreted fragment is obviously decidable. The next proposition describes a sufficient and necessary condition for a essentially uninterpreted formula to be in the finite essentially uninterpreted fragment.

The Δ_F is *stratified* if there is a function *level* that maps sets to integers and *level* satisfies the following condition:

- *level*$(S_{k,j}) \equiv$ *level*$(A_{f,i})$, if constraint $S_{k,j} \equiv A_{f,i}$ appears in Δ_F.
- *level*$(A_{f,j}) <$ *level*$(S_{k,i})$,$(1 \leq i \leq n)$, if constraint $t[S_{k,1}, \ldots, S_{k,n}] \subseteq A_{f,j}$ appears in Δ_F.

Proposition 4.2.5.1

The least solution of Δ_F is finite if and only if Δ_F is stratified.

Proof. Suppose there is such a *level* function. Note that F is finite and the range of the *level* function is finite. Suppose there are n numbers $m_1, m_2, ..., m_n$ in the range of the *level* function. A solution of Δ_F can be constructed as follows. First, let all sets with *level* of m_1 be a set containing all ground terms appearing in F. It is obvious that all sets with *level* of m_1 are finite. Next, let each set with *level* of m_2 contain all ground terms appearing in sets with *level* of m_1. If there is a constraint $t[S_{k,1}, \ldots, S_{k,n}] \subseteq A_{f,j}$ in Δ_F and the *level* of $S_{k,1}, \ldots, S_{k,n}$ is less than m_2 and the *level* of $A_{f,j}$ is m_2, then let each set with *level* of m_2 contain $t[S_{k,1}, \ldots, S_{k,n}]$. The sets with *level* m_3 can be constructed in a similar way. The procedure can continue until sets with *level* m_n are constructed. It is easy to see that the constructed solution satisfies all constraints in Δ_F, and the solution is finite. Therefore, a least solution must be finite.

Suppose the least solution of Δ_F is finite. Define a relation R overs the sets in Δ_F such that $R(S_1, S_2)$ holds if there is a constraint $t[S_{k,1}, \ldots, S_{k,n}] \subseteq A_{f,j}$ in Δ_F and S_1 appears on the left side of $\subseteq$ and S_2 appears on the right side. Let the transitive closure of R be R^*. An observation is that in R^*, $R^*(S, S)$ never holds. (If $R^*(S, S)$ holds, then the least solution of Δ_F must be infinite.)

A graph can be constructed based on the R relation. The nodes are sets, and there is a directed edge from S_1 to S_2 if $R(S_1, S_2)$ holds. It is easy to see that the graph is directed and acyclic. A topological sort of the sets based on the graph will easily assign a *level* to each set.

□

Theorem 4.2.4.1 suggests a simple decision procedure for the formulas in the FEU fragment: Just generate F^* and check its T-satisfiability using a solver for theory T.

4.2.6 Compactness and completeness

The least solution of Δ_F is infinite if some $S_{k,i}$ in the least solution of Δ_F is infinite. If Δ_F is infinite, then F^* is an infinite set of ground clauses. When F^* is infinite, it is possible to obtain a refutation complete procedure by using the standard compactness theorem of pure first order logic. A procedure is *refutation complete* if it will always terminate and report unsatisfiable for an unsatisfiable formula. The compactness theorem states that a set of first order formula SF is unsatisfiable if and only if a finite subset of SF is unsatisfiable. Suppose a formula F is being checked modulo some background theory T. Then F is T-satisfiable if and only if $\{F\} \cup T$ is satisfiable in pure first-

order logic. By applying the compactness theorem, F is unsatisfiable modulo background theory T if and only if a finite subset of $\{F\} \cup T$ is. Assume T is consistent, and since F^* is equi-satisfiable with F in T, F is T-unsatisfiable if and only if a finite subset of F^* is T-unsatisfiable. Therefore, a refutation complete procedure can be obtained by using a fair enumeration of clauses in F^*[1]. A fair enumeration of F^* can be obtained by a fair enumeration of the least solution of Δ_F.

One possible enumeration is as follows. First, eliminate all equalities $S \equiv S'$ in Δ_F, by substituting S with S' everywhere. Let Δ'_F denote the resultant system. Next, convert Δ'_F to a system of recursive equations U_F as follows. For each set S in Δ'_F, add the following recursive equation to U_F:

$$S = S \cup R_1 \cup \ldots \cup R_m$$

where m is the number of constraints in Δ'_F that contain S. If the i-th constraint that contains S is of the form $a \in S$, then R_i is $\{a\}$. If the i-th constraint is of the form $t[S_1, \ldots S_k] \subseteq S$, then R_i is $t[S_1, \ldots S_k]$. The least fixed point of this system of equations is the least solution for Δ'_F. To illustrate the construction of U_F, consider the following example:

[1] A fair enumeration means a sequence of sets of clauses F^i such that $F^1 \subseteq F^2 \subseteq \ldots F^i \subseteq \ldots \subseteq F^*$ and, for any clause C in F^* there is an n such that C is in F^n

Example 4.2.6.1 Infinite Δ_F

Let F be the following two clauses.

$$f(x_1, x_2) = 0 \vee f(g(x_2), g(x_1)) = 1,$$

$$f(a, b) = 1$$

This formula induces the following system of set constraints Δ_F.

$$S_{1,1} = A_{f,1}, \quad S_{1,2} = A_{f,2}, \quad g(S_{1,2}) \subseteq A_{f,1}, \quad g(S_{1,1}) \subseteq A_{f,2}$$

$$a \in A_{f,1}, \quad b \in A_{f,2}$$

The least solution is:

$S_{1,1} = A_{f,1} = \{a, g(b), g(g(a)), g(g(g(b))), \ldots\}$

$S_{1,2} = A_{f,2} = \{b, g(a), g(g(b)), g(g(g(a))), \ldots\}$.

After eliminating the equations in Δ_F, we obtain the following system of constraints Δ'_F.

$$g(S_{1,2}) \subseteq S_{1,1}, \quad g(S_{1,1}) \subseteq S_{1,2}$$

$$a \in S_{1,1}, \quad b \in S_{1,2}$$

Then, the following system of equations U_F is generated:

$$S_{1,1} = S_{1,1} \cup g(S_{1,2}) \cup \{a\}$$

$$S_{1,2} = S_{1,2} \cup g(S_{1,1}) \cup \{b\}$$

For each equation $S = S \cup R_1 \cup \ldots \cup R_m$ in U_F, let $S^0 = \emptyset$, and $S^{n+1} = S^n \cup R_1^n \cup \ldots \cup R_m^n$. Then, we can enumerate clauses in F^* by instantiating clauses using ground terms in $S_{k,i}^n$.

Returning to example 4.2.6.1, we have:

Example 4.2.6.2 fair enumeration

For the formula F in example 4.2.6.1, we have in the first step:

$$f(a,b) = 0 \vee f(g(b), g(a)) = 1,$$

$$f(a,b) = 1$$

Then we have in the next step:

$$f(g(b), g(a)) = 0 \vee f(g(g(a)), g(g(b))) = 1,$$

$$f(a,b) = 0 \vee f(g(b), g(a)) = 1,$$

$$f(a,b) = 1$$

And so on.

Notice that the above procedure can have counter-intuitive results because of non-standard models. Consider the following example with the background theory of integer arithmetic.

Example 4.2.6.3 Non-standard models of arithmetic

$$f(x_1) < f(f(x_1)),\ f(x_2) < a,\ 1 < f(0)$$

By Theorem 4.2.4.1, these three clauses are equisatisfiable to the set of ground clauses $F^* = \{f(0) < f^2(0), f^2(0) < f^3(0), \ldots, f(0) < a, f^2(0) < a, \ldots, 1 < f(0)\}$ modulo the background theory of integer arithmetic.

Because every finite subset of F^* is satisfiable, F^* is satisfiable by the compactness theorem. However, F^* is not satisfiable in any extension of the standard model of integer arithmetic. The reason is as follows: The clause $f(x_1) < f(f(x_1))$ says that the range of f contains a strict increasing sequence. The clause $f(x_1) < a$ says there is a value a greater than any value in the range of f, which is impossible in the standard model of integer arithmetic.

The problem here is that the first order theory of integer arithmetic has non-standard models. F^* is satisfiable in a non-standard model.

Thus, if we are interested, not in T-satisfiability, but in satisfiability in an extension of the standard model, Theorem 4.2.4.1 no longer guarantees a refutation complete procedure.

4.3 Almost uninterpreted formulas

In an essentially uninterpreted formula, a variable x can only appear as the argument of uninterpreted functions and predicates. This section discusses several extensions of the essentially uninterpreted fragment.

The methodology is the same. A formula F is shown to be equi-satisfiable to a set of instantiations of F. The ground terms used for instantiation are

from the least solution of a system of constraints derived from F.

A trivial extension is to use *destructive equality resolution* as a pre-processing step. The clause $\neg(x = t) \vee C[x]$ can be simplified to $C[t]$, where x does not occur in t.

If a formula is of the form $g(\overline{x}) = t[\overline{x}]$ and g does not appear in $t[\overline{x}]$, then this formula can be seen as a *macro definition*, because g can be replaced by t. For example, the formula $g(x_1) = x_1 + c$ is a macro definition. The simplest way to handle a macro definition $g(\overline{x}) = t[\overline{x}]$ is to remove it from the set of clauses, and replace every term of the form $g(\overline{s})$ with $t[\overline{s}]$.

4.3.1 Arithmetic literals and almost uninterpreted formulas

Literals of the form $\neg(x_i \leq x_j)$, $\neg(x_i \leq t)$, $\neg(t \leq x_i)$, where t is a ground term, are called *arithmetic literals*. Obviously, the background theory must include arithmetic and $\leq$ is interpreted as the standard less-or-equal-to relation over reals or integers. If x_i ranges over integers, positive literals of the form $x_i \leq t$ can be rewritten into $\neg(t + 1 \leq x_i)$. If variable x_i ranges over integers, then literal of the form $x_i = t$ can be rewritten into $\neg(x_i \leq t - 1) \vee \neg(t + 1 \leq x_i)$. A

formula F is *almost uninterpreted* if variables in F appear only as arguments of arithmetic literals, or as arguments of uninterpreted functions or predicates.

4.3.2 Rules for Δ_F

The following rules are used to generate the Δ_F for a formula in the almost uninterpreted fragment. Note that the first three rules are exactly the rules for formulas in the essentially uninterpreted fragment as in Section 4.2.

1. $t \in A_{f,j}$ if t is a ground term.

2. $t[S_{k,1}, \ldots, S_{k,n}] \subseteq A_{f,j}$ if t is of the form $t[x_1, \ldots, x_n]$.

3. $S_{k,i} \equiv A_{f,j}$ if t is the i-th variable x_i.

4. $S_{k,i} \equiv S_{k,j}$ if literal $\neg(x_i \leq x_j)$ appears.

5. $t \in S_{k,i}$ if literal $\neg(x_i \leq t)$ or $\neg(t \leq x_i)$ appears.

As in the essentially uninterpreted fragment, given an almost uninterpreted formula F, F^* is defined as the set of ground clauses obtained by instantiating clauses in F using the ground terms from the least solution of Δ_F.

4.3.3 From M to M^π

As formulas in the essentially uninterpreted fragment, suppose F^* has T-model M, then a T-model M^π of F can be constructed in the same way. However, the projection functions are different. The construction of M^π is pretty much the same as that for formulas in the essentially uninterpreted fragment. The rules are as follows. Note that most rules are the same as the rules for the essentially uninterpreted fragment. The rule for interpreted predicate $\leq$ is special in that the projection functions are applied to the arguments of $\leq$. This is necessary because the arguments of $\leq$ can be a variable, while in the essentially uninterpreted fragment, an argument of a predicate can only be an uninterpreted function.

1. $|M^\pi| \equiv |M|$

2. $c^{M^\pi} \equiv c^M$ for every constant c

3. $f^{M^\pi} \equiv f^M$ for every interpreted function f

4. For interpreted predicate $e_1 \leq e_2$ that appears in clause S_k, $(e_1 \leq e2)^{M^\pi} \equiv (\pi_{k,i}(e_1) \leq \pi_{k,i}(e_1))^M$

5. $P^{M^\pi} \equiv P^M$ for every interpreted predicate P other than $\leq$

6. $f^{M^\pi}(e_1, \ldots, e_n) \equiv f^M(\pi_{f,1}(e_1), \ldots, \pi_{f,n}(e_n))$ for every uninterpreted function f

7. $P^{M^\pi}(e_1, \ldots, e_n) \equiv P^M(\pi_{p,1}(e_1), \ldots, \pi_{p,n}(e_n))$ for every uninterpreted predicate P

The new projection $\pi_{k,i}$ for formulas in the almost interpreted fragment is defined as follows: give an element e in $|M|$, let $\pi_{k,i}(e) \equiv e_1$ such that $e_1 \in M(S_{k,i})$ and one of the following conditions holds.

- $e_1 \leq e$ and for all $e_2 \in M(S_{k,i})$, either $e_2 \leq e_1$ or $e_2 > e$;
- $e_1 > e$ and for all $e_2 \in M(S_{k,i})$, $e_1 \leq e_2$.

Here, for elements e_1 and e_2 in M, $e_1 \leq e_2$ means $\langle e_1, e_2 \rangle \in \leq^M$, and $e_1 > e_2$ means $\langle e_1, e_2 \rangle \notin \leq^M$.

Given an element e in $|M|$ and a function f that appears in the formula, there are two cases. In the first case, if there is an element in $M(S_{k,i})$ that is less than e, then let $f^M(e^M)$ has the same value as $f^M(e_1^M)$, where e_1 is the greatest elements among elements in $M(S_{k,i})$ that are less than e. In the second case, e is less than any element in $M(S_{k,i})$. Then let $f^M(v^M)$ has the same value of $f^M(e_2^M)$, where e_2 is the least element in $M(S_{k,i})$.

As before, $\pi_{k,i} \equiv \pi_{f,j}$ if $S_{k,i} \equiv A_{f,j}$. Note that the range of $\pi_{k,i}$ is equal to $M(S_{k,i})$, and $\pi_{k,i}(e) = e$ for any $e \in M(S_{k,i})$.

For a better understanding of the intuition behind the new rules for Δ_F and the projection function, imagine variable x_i ranges over the x-axis in the Cartesian coordinate system. As described, x_i will be instantiated with all terms in $S_{k,i}$. A literal of the form $\neg(x_i \leq t)$ can be seen as an operation that cuts the x-axis into two segments, one contains elements less or equal to t and the other segment with elements greater than t. The segment containing all elements less or equal to t is *interesting*, and the other segment is not interesting because the clause k is trivial if x_i is interpreted as an element in the uninteresting segment. A literal of the form $x_i = t$ cuts the x-axis into two interesting segments, those less than t and those greater than t. The construction of Δ_F ensures that for each interesting segment s there is at least one element e in s such that e is the interpretation of some ground term in $S_{k,i}$. In other words, each interesting segment has a *representative* in $S_{k,i}$. Therefore, in some sense the instantiations obtained by instantiating these representatives capture the property of the quantified formula.

If F^* has a model, suppose r is a representative of an interesting segment. For any element e in the interesting segment that r represents, make $f^M(e)$

have the same value as $f^M(r^M)$. In this way, a model of the quantified formula is constructed, which shows the quantified formula is indeed satisfiable.

The rule for literals of the form $\neg(x_i \leq x_j)$ is a bit tricky. Intuitively, this literal will not give any more interesting segments at all because it contains no ground terms that could be used to cut the x-axis.

As in essentially uninterpreted fragment, we have a similar theorem.

Theorem 4.3.3.1

For a formula F in the almost uninterpreted fragment, F and F^* are equi-satisfiable in T.

The formal proof is pretty much as the proof of Theorem 4.2.4.1 in the essentially uninterpreted fragment. The lemma and propositions are as follows.

Lemma 4.3.3.1

For every ground term t that appears in F^*, $t^M \equiv t^{M^\pi}$.

The proof is the same as the similar Lemma 4.2.3.1 for the essentially uninterpreted fragment. Note that only terms are concerned.

Proposition 4.3.3.1

For a ground term t, $t^{M^\pi\{\overline{x}\mapsto\overline{e}\}} \equiv t^{M\{\overline{x}\mapsto\pi_k(\overline{e})\}} \equiv t^M$.

The proof is the same as the similar Proposition 4.2.4.1 for the essentially uninterpreted fragment. Note that a projection function for the almost uninterpreted fragment is also a projection function for the essentially uninterpreted fragment.

To prove the Theorem 4.3.3.2, the following proposition is needed.

A projection function $\pi_{k,j}$ is *monotonic* if for all e_1 and e_2 in $|M|$, $e_1 \leq e_2$ implies $\pi_{k,j}(e_1) \leq \pi_{k,j}(e_2)$.

Proposition 4.3.3.2

The projection functions defined in this section are monotonic.

Proof. Given a projection function $\pi_{k,j}$ and elements e_1 and e_2 in $|M|$. Suppose $e_1 \leq e_2$. If $e_1 \equiv e_2$, then of course $\pi_{k,j}(e_1) \equiv \pi_{k,j}(e_2)$ and thus $\pi_{k,j}(e_1) \leq \pi_{k,j}(e_2)$

If e_1 is not equal to e_2, there are two cases:

1. There is an element e in $M(S_{k,i})$ such that $e \leq e_1$.

   If there is an element in $M(S_{k,i})$ that is between e_1 and e_2, then by the definition of $\pi_{k,j}$, $\pi_{k,j}(e_1) \leq \pi_{k,j}(e_2)$.

   If there is no element in $M(S_{k,i})$ that is between e_1 and e_2, then by the definition of $\pi_{k,j}$, $\pi_{k,j}(e_1) \equiv \pi_{k,j}(e_2)$. Again $\pi_{k,j}$, $\pi_{k,j}(e_1) \leq \pi_{k,j}(e_2)$.

2. There is no element e in $M(S_{k,i})$ such that $e \leq e_1$. Then $\pi_{k,j}(e_1)$ is the least element in $M(S_{k,i})$ by the definition. Thus $\pi_{k,j}(e_1) \leq \pi_{k,j}(e_2)$ holds.

□

Proposition 4.3.3.2 says that for arithmetic literal of the form $\neg(t_1 \leq t_2)$, where t_1 and t_2 are terms, if $\neg(t_1 \leq t_2)$ holds in M, then it holds in M^π. Note that if $\neg(t_1 \leq t_2)$ holds in M^π, then usually we do not have that $\neg(t_1 \leq t_2)$ holds in M.

Theorem 4.3.3.2

For a formula F in the almost uninterpreted fragment, F and F^* are equi-satisfiable in T.

Proof. The proof is similar to the proof for Theorem 4.2.4.1.

If F^* is unsatisfiable, then so is F.

Suppose that F^* is T-satisfiable, but F is not. Let M be a T-model for F^* and M^π is the π-extension of M. Since F is unsatisfiable, there is a clause $C_k[\overline{x}]$ in F such that for some $\overline{e}$, $M^\pi\{\overline{x} \mapsto \overline{e}\} \not\models C_k[\overline{x}]$. Suppose l is a literal appearing in $C_k[\overline{x}]$. If l is does not contain $\leq$, then by Lemma 4.3.3.1, $M\{\overline{x} \mapsto \pi_k(\overline{e})\} \not\models l$. If l contains $\leq$, then by Proposition 4.3.3.2, $M\{\overline{x} \mapsto \pi_k(\overline{e})\} \not\models l$. Therefore, $M\{\overline{x} \mapsto \pi_k(\overline{e})\} \not\models C_k[\overline{x}]$. Let $\overline{e}$ be the tuple $\langle e_1, \ldots, e_m \rangle$, then for every

$\pi_{k,i}(e_i)$ in $\pi_k(\overline{e})$, there is some ground term r_j in $S_{k,j}$ such that $r_j^M \equiv \pi_{k,j}(e_j)$. Thus, $M\{\overline{x} \mapsto \pi_k(\overline{e})\} \models C_k[\overline{x}]$ if and only if $M \models C_k[\overline{r}]$, and consequently $M \not\models C_k[\overline{r}]$, contradicting the assumption that M satisfies F^* since $C_k[\overline{r}]$ is in F^*.

□

If function applications are not nested, i.e. terms of the form $(f(g(a))$ are not allowed, the almost uninterpreted fragment becomes the array property fragment in [10].

Example 4.3.3.1 Stratified Arrays

The following set of clauses is satisfiable. In this example, f can be seen as an array that maps an integer to pointers. h can be seen as a heap from pointers to values, and h' is the heap h after an update at position a with value b.

1. $\neg(0 \leq x_1) \vee \neg(x_1 \leq x_2) \vee \neg(x_2 \leq n) \vee h(f(x_1)) \leq h(f(x_2))$
2. $\neg(0 \leq x_1) \vee \neg(x_1 \leq n) \vee f(x_1) \neq c$
3. $\neg(x_1 = a) \vee h'(x_1) = h(x_1)$
4. $h'(a) = b$
5. $0 \leq i$

6. $i \leq j$

7. $j \leq n$

8. $h'(f(i)) > h'(f(j))$

The first clause says that the array f is sorted ascending within range $[1, n]$(in terms of the values in h pointed to by the elements in f). The second clause says that within range $[1, n]$, f is not equal to c. The last clause say that there exists i and j such that $0 \leq i \leq j \leq n$ and the values pointed by $f(j)$ in h' is greater than the values pointed by $f(j)$ in h'. This is satisfiable if a is equal to some $f(i)$, $(0 \leq i \leq n)$. Note that if the second clause is $\neg(0 \leq x_1) \vee \neg(x_1 \leq n) \vee f(x_1) \neq a$, then the examples is unsatisfiable.

4.4 Equalities in many-sorted logic

This section deals with equalities over uninterpreted sorts in many-sorted logic. The methodology is the same as before. A formula F is show to be equi-satisfiable to a conjunction of ground instantiations of F, and the ground terms are from the least solution of a set of constraints Δ_F.

Sorts naturally arise in SMT applications and in some cases sort information significantly simplifies the problem. SMT solvers such as CVC3 and

Z3 [21] have support for sorts. A sort σ is uninterpreted if it is not in the signature of the background theory. Otherwise, σ is interpreted. Let $=_\sigma$ denote the equality predicate for elements of sort σ.

Given a formula F in many-sorted logic that contains $=_\sigma$ for uninterpreted sort, a standard technique in general first order solvers is to provide a set of axioms for equality $=_\sigma$, and then treat $=_\sigma$ as an uninterpreted predicate symbol. The basic idea is to add the clauses EQ_σ that assert $=_\sigma$ is reflexive, symmetric, transitive, and congruent.

For SMT solvers, there is no need to add the clauses EQ_σ, because any SMT solver has built-in support for equality. Based on the framework in this chapter, it is sufficient to add to Δ_F any constraints induced by EQ_σ. The idea is to introduce a new set S_σ for each uninterpreted sort σ. Intuitively, S_σ contains the ground terms of sort σ. Let $dom_{f,j}$ denote the sort of the j-th argument of f. The following rules are needed to generate Δ_F.

1. $S_{k,i} = S_\sigma$, if x_i appears as an argument of $=_\sigma$.

2. $t[S_{k,1}, \ldots, S_{k,n}] \subseteq S_\sigma$, if $t[x_1, \ldots, x_n]$ appears an argument of $=_\sigma$.

3. $A_{f,j} \equiv S_\sigma$, if $dom_{f,j}$ is σ.

For example, the following formula can be handled now:

$$\neg subtype(x_1, x_2) \vee \neg subtype(x_2, x_1) \vee x_1 =_\sigma x_2$$

This formula is used to axiomatize the anti-symmetry property for the subtype relation in ESC/Java [30]:

A formula in many-sorted logic is *stratified* if there is a function *level* from sorts into integers such that for each function symbol $f\colon \sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma$, $level(\sigma) < level(\sigma_i)$ for all $i = 1, \ldots, n$. It is known that a many-sorted formula F over a stratified signature can be decided by instantiation [2]. This fact is a trivial consequence of Theorem 4.3.3.2 for the many-sorted case.

4.5 Modular equalities

Let $a =_m b$ denote that $a - b \equiv m * c$ for some value c. In some applications, modular equalities are useful. For example, suppose an object of some date structure occupies n bytes of memory. An array of such objects is stored in memory from location l_1 to l_2. The following formula says that all elements in the array have the same value. $star(p)$ intuitively means the value stored in the memory pointed to by p.

$$\forall p.(l_1 \leq p \leq l_2 \wedge p =_n l_1 \implies star(p) = e)$$

Define a literal of the form $\neg(x_i =_c d)$ a modular equality literal. Here, c is a concrete natural number and x_i is a variable.

In the rest of this section, it is assumed that if a modular equality appears in a clause C_k, then no other interpreted predicates appears in C_k. Given a formula F, let Δ'_F be the set of constraints obtained by using rules described so far. It is assumed that the least solution of Δ'_F is finite.

The following rule is needed for modular equalities of the form $\neg(x_i =_c d)$.

- Define $S_{k,i}$ to be the set $\{t, t+1, t-1, \ldots, t+c-1, t-c+1 \,|\, t \in S'_{k,i}\}$, where $S'_{k,i}$ is from the least solution of Δ'_F.

For example, suppose $S'_{k,i}$ is $\{a\}$ and $\neg(x_i =_4 g)$ appears in the formula, then $S_{k,i}$ is $\{a, a+1, a+2, a+3, a-1, a-2, a-3\}$.

Define F^* be the set of instantiations of F by using the ground terms from the $S_{k,i}$. As before, we have a theorem stating that F^* is equi-satisfiable with F in T. The formal proof is similar to the proof of Theorem 4.3.3.2. Suppose F^* has T-model M. The projection function is defined as follows.

For elements e_1 and e_2 in M, let $e_1 =_c e_2$ denote that $\langle e_1, e_2\rangle \in (=_c)^M$.

Define $\pi_{k,i}(e) \equiv e_1$ such that $e_1 \in M(S_{k,i})$, $e =_c e_1$, and one of the following conditions holds.

- $e_1 \leq e$ and for all e_2 such that $e_2 =_c e$ and $e_2 \in M(S_{k,i})$, either $e_2 \leq e_1$ or $e_2 > e$;

- $e_1 > e$ and for all e_2 such that $e_2 =_c e$ and $e_2 \in M(S_{k,i})$, $e_1 \leq e_2$.

As the projection for almost uninterpreted fragment, there are two cases. In the first case, there is an element in $M(S_{k,i})$ that is less than e, then let e_1 be the greatest element among elements in $M(S_{k,i})$ that are less than e and $e =_c e_1$. In the second case, there is no element in $M(S_{k,i})$ that is less then e, then let e_1 be the smallest element in $M(S_{k,i})$ such that $e =_c e_1$.

4.6 Related work and discussion

Some of the ideas described in this chapter have been implemented in the Z3 solver submitted to the SMT 2008 competition[2]. Z3 was the only theorem prover in the competition that could prove quantified formulas satisfiable.

Arrays are common in most programming languages and provide a natural model for memories. A decision procedure for array properties is of great interest. From the view point of logic, arrays can be treated as uninterpreted functions: array reads can be seen as function applications, and array writes

[2] `http://www.smtcomp.org`

can be encoded by array reads by using a common trick [50]. Therefore, the decision procedures discussed in this chapter can be directly applied to prove complex quantified array properties.

The fragment that contains arithmetic literals and the associated projection functions resemble much in spirit the array property fragment and its projection function proposed in [10], which is the original motivation for this chapter. The formulas in the array property fragment are of the form $I \vee V$, where I is a disjunction of "arithmetic literals" and V is an essentially uninterpreted formula without nested array reads. I is called *index guard* and V is called *value constraints*. It is obvious that the fragments in this chapter subsume the array property fragment. As proved in [10], nested array reads on indices will in general make the formula undecidable. However, this chapter shows that for certain cases even if nested array reads appear, a decision procedure is still possible as long as the set F^* is finite.

In [39] a logic called LIA is proposed, in which modulo equalities over variables, difference constraints, and non-nested array reads are allowed. The difference constraints in LIA are of the form $t - s \leq c$, where c is ground. It is also required in LIA that if t is a variable then s cannot be an array read containing a variable and vice versa. The decidability of LIA is proved by

employing a customized counter Büchi automata.

In a subsequent paper [38], alternating quantifiers over array indices are studied.

Ghilardi et al [36] proposed a logic for quantifier free array formulas that supports some interpreted predicates for specifying array properties.

In [31, 3, 2] procedures based on stratified vocabularies are presented. These procedures are in the context of many-sorted logic. A vocabulary is stratified if there is a function *level* from sorts to naturals, and for every function $f\colon \sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma$, $level(\sigma) < level(\sigma_i)$. Our method can decide a broader class of problems. For example, these methods fail if there is a single function $f\colon \sigma \rightarrow \sigma$, and cannot handle simple examples such as $f(x) = b \wedge f(a) = a$.

In [48] local theories and local extensions are studied; they propose a complete instantiation procedure for certain types of quantified formulas. A major difference is that the method in this chapter can provide models for satisfiable cases.

If an essentially uninterpreted formula only contains uninterpreted symbols, then Theorem 4.2.4.1 can be viewed as a frugal version of the standard Herbrand theorem, and the universe does not necessarily become infinite in the presence of functions.

This decision procedure in this chapter is also suitable for formulas coming from verification of parameterized systems [3].

Some customized theories also have decision procedures based on complete instantiation, and an example is shown in [52].

Chapter 5

Proof translation

While SMT solvers are much more efficient than a few years ago, they are also becoming more complicated. For example, CVC3 contains over 100 thousand lines of code now. Since SMT solvers are mostly used for verification applications where correctness is essential, a natural question is: are these SMT solvers correct? As shown in the competition SMT-COMP [1] and in [11] some SMT solvers did produce wrong results.

By employing the verification techniques available today, it would be extremely difficult, if not impossible, to verify that a modern SMT solver is correct. Even if such a verification were done, it would be very difficult to maintain the correctness since SMT solvers are constantly changing. A viable

alternative is to ask an SMT solver to produce a proof and then proceed to check the proof. For example, CVC3 can provide a formal proof for every unsatisfiable case it can prove. The rationale is that checking a proof should be easier than checking the prover itself.

This prove-and-check approach faces several challenges however. The first challenge is to design a suitable set of proof rules. Unlike SAT solvers, for which only one proof rule (propositional resolution) is sufficient for proof-checking, SMT solvers require a rich set of proof rules, depending on the background theories and the decision procedures employed. For proof-checking, a small set of simple rules is preferred. On the other hand, a larger set of complex rules is better for ease of implementation.

The next challenge is that of proof checking. Industrial cases are usually large. As a result the proofs are large, which makes them impossible for human to check. Therefore, a proof checker, another program, is needed. For simple proof rules, such as deriving $\neg true$ from $false$, a simple syntactic check is sufficient. However, for more complicated rules some sophisticated theory reasoning is required. For example, CVC3 encapsulates the normalization of a term of linear arithmetic in one proof rule, and a proof checker needs to check that the term before the normalization is equal to the term after, which is not

trivial at all. Even if such a proof checker can be obtained, the correctness of such a proof checker becomes questionable too because such a proof checker itself might be rather complicated.

This chapter proposes a proof translator instead of a proof checker. Most of the results in this chapter has been reported in [33]. The basic idea is to translate a proof into another trusted theorem prover. If the same theorem can be proved in the trusted prover, then it is assumed that the proof is indeed correct. In other words, the correctness is reduced to the trusted prover. A proof translator should be simpler than a proof checker. By using a proof translator the actual proof checking is done by the trusted prover. If the trusted prover is powerful enough, any proof rules, simple or complicated, can be handled. If some proof rules are changed inside the SMT solver, then only part of the translator needs to be updated.

Needless to say, for this proof-translating approach to work, it is crucial to select a good trusted prover. The trusted prover must be highly reliable. Since proofs from industrial cases are big, the trusted prover, as well as the proof translation procedure, must be highly efficient too. This chapter will show that there is a good choice of such a trusted prover, and the experimental results show that the this proof-translating method is feasible. For hard cases, the

average time spent on proof translation and checking is less than the average time used to find the proof. These results are especially significant considering that the proof translator and proof checker employed in the experiments are implemented in an interpreted language while the SMT solver is implemented in C++.

The chapter is organized as follows. Section 5.1 gives a brief introduction to CVC3's proof system. Section 5.2 describes the prover HOL Light and the reasons why it was chosen as the trusted prover. Section 5.3 discusses the translation procedure and several obstacles to efficiency. Section 5.4 discusses the experiments. Section 5.5 discusses the related work, and Section 5.6 concludes.

5.1 CVC3

CVC3 is the latest in a series of SMT solvers that originated with the CVC solver which was developed at Stanford University. High confidence has always been a primary goal of the provers in the CVC family. CVC pioneered a proof system within a state-of-the-art SMT solver [60]. CVC3's proof system continues in this tradition.

For performance, CVC3 employs state-of-the-art optimization strategies, and a large set of complex proof rules. At the time of this dissertation, CVC3 contained about 298 proof rules.

5.1.1 Proofs in CVC3

A *proof* can be seen as a tree in which each node represents a *theorem*. The theorem associated with the root of the proof tree represents the theorem of the proof tree. A *theorem* is a pair $\Gamma \vdash \phi$, where Γ is a set of formulas and ϕ is a formula. Γ is called the *assumptions* of the theorem and ϕ is called the *conclusion*.

A *proof rule* or *inference rule* is a function that takes some proofs and parameters and returns a new proof. Since a proof is a tree, a proof rule can be seen as a function that takes several trees as input and returns a new tree. On the other hand, because a theorem is always associated with a proof tree, a proof rule can be seen as a function that takes several theorems and returns a new theorem. A proof rule is *sound* if it returns a valid theorem when given valid theorems as input. If all proof rules are sound, then the theorem of a proof tree is always valid.

A proof rule is denoted by the following notation:

$$\frac{P_1 \quad \cdots \quad P_n}{C}$$

where the P_i's are proofs and C represents the theorem associated with the new proof tree.

The key point for proof translation is that every node in the proof tree represents a theorem, which is supposed to be valid. The proof translation is then just a bottom-up, i.e. from the leaves to the root, recursive traversal over the proof tree. During the recursive traversal, the theorem associated with each node is translated into HOL Light and checked by HOL Light.

5.1.2 Examples of proof rules

The most basic rule is the assumption rule, shown below:

Example 5.1.2.1

$$\frac{}{\phi \vdash \phi}\ \text{assume}$$

This assumption rules takes no other proof and just returns a theorem $\phi \vdash \phi$. Intuitively, this proof rule says that ϕ is true given the assumption that ϕ is true.

The next proof rules formalize the transitivity of the propositional connector $\leftrightarrow$. Given two theorems $\Gamma_1 \vdash \phi \leftrightarrow \psi$ and $\Gamma_2 \vdash \psi \leftrightarrow \theta$, the rule returns a new theorem $\Gamma_1 \cup \Gamma_2 \vdash \phi \leftrightarrow \theta$.

Example 5.1.2.2

$$\frac{\Gamma_1 \vdash \phi \leftrightarrow \psi \quad \Gamma_2 \vdash \psi \leftrightarrow \theta}{\Gamma_1 \cup \Gamma_2 \vdash \phi \leftrightarrow \theta}\,\text{iffTrans}$$

A complicated proof rule is shown below:

Example 5.1.2.3

$$\frac{\Gamma_0 \vdash \alpha_0 \quad \Gamma_1 \vdash \alpha_1 \quad \ldots \quad \Gamma_n \vdash \alpha_n}{\Gamma_0 \cup \Gamma_1, \ldots, \Gamma_n \vdash \phi \leftrightarrow \phi'}\,\text{simplify}$$

The `simplify` rule accepts a set of theorems $\{\Gamma_i \vdash \alpha_i \| \ 0 \leq i \leq n\}$ and a parameter ϕ, where ϕ is a formula. It returns a theorem $\Gamma_0 \cup \Gamma_1, \ldots, \Gamma_n \vdash \phi \leftrightarrow \phi'$, in which ϕ' is obtained by replacing all instances of α_i in ϕ by *true* and then applying simple propositional rewriting and simplification to ϕ.

5.1.3 Implementation

CVC3 uses a class `Theorem` to store theorems. Each instance of the `Theorem` contains the assumptions and the conclusion. The proof generation can be

turned off for performance. When proof generation is on, the proof for each theorem is recorded. It is worth mentioning that even when proof generation is turned off, the assumptions of a theorem are still stored because they are useful for conflict analysis (see [6]).

In CVC3, every formula that is asserted by theory reasoning is represented as a theorem, and a theorem can only be created by a proof rule. Usually the implementation of a proof rule includes some sanity checking to ensure that the proofs and parameters provided as input are appropriate to the rule.

5.2 HOL Light

The trusted prover is HOL Light [43]. HOL Light is a general-purpose interactive theorem prover based on higher order logic. As with other provers in the HOL family, HOL Light draws upon the work of Mike Gordon et al. [37] that dates back to the early 1980s.

HOL Light is powerful. The logic system used in HOL Light is capable of formalizing most mathematics. In principle, it can formalize any theories supported by current and future SMT solvers.

HOL Light is simple and reliable. Unlike other HOL-like provers, HOL

Light's logical core is very small and has been purified and simplified. The logic core is based on a variant of typed lambda calculus, and contains ten inference rules, mostly about equality, three axioms, and two functions for conservative definitional extension. HOL Light is implemented in Ocaml and the logical core is only about 430 lines of Ocaml code. There has even been work verifying that the logic core of HOL Light is indeed correct [45]. Except for equality, all other logical symbols are defined, even the propositional connectors like "$\wedge$" and "$\vee$". HOL Light's definitional extension mechanism guarantees that every new definition is sound and that any theorems proved are valid as long as the logical core is consistent. In other words, there is no way for the user to compromise the soundness of the system.

HOL Light can easily be extended. Complicated proof rules and even decision procedures can be implemented as Ocaml functions. Many such decision procedures have been implemented and are included in the HOL Light distribution already. For example, HOL Light contains a highly efficient decision procedure for real arithmetic, which can be invoked by the HOL Light command *REAL_ARITH*. HOL Light also includes decision procedures for propositional and first-order reasoning, which can be leveraged for proof checking.

As John Harrison, the author of HOL Light, has stated, HOL Light "sets a

very exacting standard of correctness" [42]. HOL Light has been chosen for a number of real world projects. For example, it has been employed for floating point algorithm verification at Intel [44] and in the Flyspeck project [40] that is attempting to formalize a proof of the Kepler Conjecture. For more details about HOL Light, please refer to the website [42].

5.3 Proof translation

The proof translator is mostly implemented in OCaml on top of HOL Light. CVC3 and the proof translator communicate through the C-API of CVC3 and the C interface of OCaml.

If CVC3 returns "unsatisfiable" for a given case, then a proof can be generated. This proof is the input to the proof translator. The translator traverses the proof tree and tries to prove the same theorem in HOL Light. The first step in the translation is to translate formulas in CVC3 into formulas in HOL Light.

5.3.1 Translation of formulas

The translation of formulas is mostly a syntactic one, and most of the time is fairly straightforward. There are, however, a few tricks in the translation.

SMT-LIB supports a built-in predicate of variable arity called `distinct`. `distinct`$(x_1, x_2, ..., x_n)$ means for all i and j, $1 \leq i, j \leq n$, $i \neq j \rightarrow x_i \neq x_j$) holds. The problem is that predicates in HOL Light must have a fixed arity. To deal with the `distinct` predicate, a new OCaml function is constructed in HOL Light. The OCaml function accepts a integer n and generates a new definition `distinct`$_n$ in HOL Light. Since it is quite expensive to create a new definition like `distinct`$_n$ in HOL Light, the definition is generated on the fly when needed.

The translation of variables and constants of real and integer types is a bit tricky. In CVC3, integers can appear wherever reals are allowed. In HOL Light, it is impossible to mix integers and reals because no operators exist that can accept a mix of reals and integers. It would be difficult to define such a set of new operators in HOL Light. Even if such a set of new operators were defined, all the decision procedures for arithmetic in HOL Light would have to be redone to incorporate the new operators, which is an even more difficult task. The proof translator adopts a simple solution, and the basic idea is to

lift integers into reals if both integers and reals appear in a formula. Though it sounds easy to implement, there are some difficulties for this approach too. One of the problems is that some proof rules are only valid for integers. For example, the theorem states that $x < c$ is equivalent to $x \leq c + 1$ is valid only if x is an integer. Therefore, x cannot be lifted into reals just because x appears in some operations involving reals. In other words, an integer should be lifted into reals only when necessary. Even this method does not always work. For example, suppose a proof rule accepts two theorems:

1. $\Gamma_1 \vdash c = d$

2. $\Gamma_2 \vdash P(c)$

The proof rule then returns a new theorem $\Gamma_1 \cup \Gamma_2 \vdash P(d)$. The proof of the new theorem in HOL Light depends on the fact that the c in theorem 1 and theorem 2 are the same c. However, it is possible that one c is an integer and the other c is lifted into a real. The solution adopted in the proof translator is to do some ad hoc work for such proof rules when needed.

CVC3 uses a special proof rule for skolemization. Given an existentially quantified formula $\exists x.P(x)$, CVC3 produces a theorem

$$P(c) \leftrightarrow \exists x.P(x) \vdash P(c) \leftrightarrow \exists x.P(x)$$

where c is a fresh constant. The c is called a *skolem* constant. The proof rule is valid, but it introduces a brand new assumption. In general $\vdash P(c) \leftrightarrow \exists x.P(x)$ is not a valid theorem. One solution is to leave the new assumption there, with the result that the resultant theorem in HOL Light is somewhat different due to these assumptions. The proof translator employs a different approach. As a special case, a skolem constant is translated as a *choice operator* applied to the body of the existentially quantified formula. The choice operator is denoted by @ in HOL Light. $@x.P$ means an element c that makes $P[x/c]$ true. For an existentially quantified formula $\exists x.P(x)$, $\vdash P[x/@x.P] \leftrightarrow \exists x.P(x)$ is now a valid theorem and no extra assumptions are needed.

5.3.2 Translation of proofs

As discussed earlier, the proof translator traverses the proof tree and translates the theorem represented by each node into HOL Light. After a formula is translated, the next step is to prove it in HOL Light.

A naive approach is just to call HOL Light's built in decision procedures and hope they will succeed. For instance, for arithmetic theorems, `REAL_ARITH` can be used. Most of the time, this approach is rather slow, especially for complicated rules and for rules that are frequently used.

A much better method is to exploit HOL Light's higher order reasoning capability, and prove a *meta-theorem* for each proof rule, which can be done before the translation starts. For example, a CVC3 proof rule returns the following theorem where A,B and C are arbitrary formulas:

$$\frac{}{\vdash (A \vee B) \vee (A \vee C) \leftrightarrow (A \vee (B \vee C))}$$

After the formula is translated, the naive approach would call some HOL Light decision procedure, say *TAUT*. We can do better by proving the following meta-theorem in HOL Light at first:

```
!a b c. ((a \/ b) \/ (a \/ c)) <=> (a \/ ( b \/ c))
```

Then after A, B, C are translated, we just instantiate a, b, c with A, B, C in the above meta-theorem. Please note that the meta-theorem is a higher order theorem because a,b and c range over propositional formulas.

Instantiation of such meta-theorems is highly efficient in HOL Light and can be used whenever a proof rule directly corresponds with a HOL Light theorem.

Sometimes it is difficult to represent a proof rule by a unique meta-theorem. For example, CVC3's `or_distributivity` rule generates a theorem $(A \wedge B_1) \vee (A \wedge B_2) \vee ... \vee (A \wedge B_n) \leftrightarrow A \wedge (B_1 \vee B_1 \vee ... \vee B_n)$, where n is not fixed and can

be any concrete number. For such proof rules, a customized meta-theorem is generated on-the-fly and then instantiated. For the `or_distributivity` rule, a formula, which represents the meta-theorem, is obtained by replacing A and each B_i with fresh propositional variables, and then the formula is proved by HOL Light's decision procedure yielding the meta-theorem. Instantiation of such meta-theorems is typically faster than proving the desired theorem directly. In the above case, the A and B_i in a particular instance of the proof rule could be arbitrarily complex, and HOL Light's decision procedure could spend a lot of resources by going deep into A and B_i.

Sometimes CVC3 and HOL Light have similar proof rules. For example, the `subst_op` rule in CVC3 and the `SUB_CONV` procedure in HOL Light are both for substitutions. However, the `subst_op` has a variant in which only some of the children are substituted. For this rule, a variant of `SUB_CONV` obtained from modifying the existing code of `SUB_CONV` is used.

Finally, some proof rules are too complicated for any approaches discussed so far. An example is the `rewrite_and` rule. This rule flattens nested conjunctions, removes conjunctions containing *true*, and performs several other rewrites. For such proof rules, ad hoc translation methods are used. For the `rewrite_and` rule, the translation is optimized by exploiting a HOL Light's

procedure for rewriting conjunctions.

5.3.3 Translation of propositional reasoning

Most modern SMT solvers use an off-the-shelf SAT solver for propositional reasoning and CVC3 is no exception. Modern SAT solvers like zChaff and Minisat can provide a resolution proof for unsatisfiable formulas. Only one rule - the propositional resolution rule - is needed for SAT solvers, and the resolution rule can be described as follows, where A, B and C can be any arbitrary propositional formulas.

$$\frac{\Gamma_1 \vdash A \vee B \quad \Gamma_2 \vdash \neg A \vee C}{\Gamma_1 \cup \Gamma_2 \vdash B \vee C}\ \text{bool\_resolution}$$

The proof rule is very simple, but proving such theorems in HOL Light turns out not so easy because A and B can be any propositional formulas, sometimes very large and complicated ones.

One method we experimented with is as follows: Suppose A is to be resolved, the first step is to reorder the disjunctions to move A and $\neg A$ to the front of clauses, removing any duplicate occurrences at the same time. Next, instantiate a meta-theorem like this: $(A \vee B) \wedge (\neg A \vee C) \leftrightarrow (B \vee C)$. Experiments show that re-ordering and removing duplicated terms is very expensive,

hole5	ReOrd1	ReOrd2	Seq	OrdList
Time	255s	155s	37s	2.8s

Table 5.1: Translation of Propositional Resolution

especially when A or B are complex.

The approach used in the proof translator adopts the idea from [66]. The key point is to represent the CNF clauses by the so-called *Sequent Representation*, in which the literals of a clause are put into the assumptions of a theorem. For instance, $\Gamma \vdash A \vee B$ is represented as $\Gamma, \neg A, \neg B \vdash False$. In HOL Light the assumptions of a theorem are treated as a set. Therefore, no re-ordering is needed and duplicates are removed automatically. In the latest version of HOL Light, the assumption list of a theorem is stored as an ordered list to further speed up propositional resolution. The following table shows some experimental results that indicate the time used by different methods of propositional resolution for a test case *hole5*.

For the *hole5* case, the first two columns show the time taken for two different versions of the translator doing explicit reordering and duplicate removal. They use 255 seconds and 155 seconds respectively. When the sequent representation is used, the time is reduced to 37 seconds. Using HOL Light with

the ordered list, the time is further reduced to 2.8 seconds.

Another related problem is due to the conversion of formulas into CNF before the SAT solver is called. CVC3 employs the standard Tseitin-style [64] conversion algorithm for CNF conversion, and additional variables are introduced during the conversion. For example, formula $A \leftrightarrow B$ will be converted into four clauses: $P \vee A \vee B$, $P \vee \neg A \vee \neg B$, $\neg P \vee \neg A \vee B$, $\neg P \vee A \vee \neg B$, where P is a new propositional variable. These clauses are not valid theorems. However, P is actually a placeholder. If we replace P by $A \leftrightarrow B$, then all the four clauses become tautologies. When translating these clauses from CNF conversion, placeholders like P should be replaced by the formulas they represent, rather than directly translating the placeholders into HOL Light. The new CNF clauses can be obtained by instantiating the corresponding tautologies.

5.3.4 Final check

If the proof translation succeeds, the result is a theorem in HOL Light. When the theorem is big, it is difficult to determine whether the theorem in HOL Light is actually the original theorem proved by CVC3. If there is a bug in the proof translator, it is possible that HOL Light proves a different theorem. To address this issue and eliminate the need to trust the proof translator, after

a proof is translated, a check is done to compare the translated theorem in HOL to the original theorem. The idea is to translate the assumptions and conclusion of the original theorem into terms in HOL Light, and then compare these terms in syntax to the assumptions and conclusion of the theorem proved in HOL Light.

5.3.5 Using the proof system to debug CVC3

The proof system plays an important role in maintaining consistency of CVC3. Once a time after some changes to the arithmetic module, CVC3 reported unsatisfiable for a known satisfiable benchmark. Manual examination could not find any problems. The bug was finally caught by running the proof translator. It was discovered that one step in the proof cannot be validated by HOL Light during the proof translation. A careful analysis of the proof rule in question showed that the proof rule was not sound. The proof rule had been written to produce a theorem with the conclusion in the form $(D \vee G) \wedge (\neg D \vee \neg G)$. D and G are some arithmetic terms. However, the intended conclusion of the theorem was $(D \vee G)$. This bug had been in CVC3 for years and it would have been extremely difficult to find without the proof system.

5.4 Experimental results

The experimental results are based on evaluation on benchmarks from the SMT-LIB library. These benchmarks are used for comparison in many papers as well as in the annual SMT-COMP competition. Every benchmark in SMT-LIB contains a status field that indicates whether the case is satisfiable, unsatisfiable, or the status is unknown. Though most benchmarks' status fields are correctly labeled, two cases were found that were incorrectly labeled during the experiments.

The experiments evaluated the proof translator on a subset of the AUFLIA benchmarks from SMT-LIB. All tests were run on AMD Opteron-based (64 bit) systems running Linux. The memory limit was 2GB and the time limit was one minute for CVC3 and 10 minutes for the proof translator.

Table 5.2 summarizes the results. Each row shows, from left to right, the name of the family of benchmarks, the total number of unsatisfiable cases in the family, the total number of cases that CVC3 reported unsatisfiable, the total and average time spent by CVC3, the number of cases for which the proof translation was successful, and the total and average time for the translation.

As seen from the table, the proof translator successfully translated most

family		CVC3			HOL Translation		
	Cases	Proved	Total	Ave	Proved	Total	Ave
simplify	833	833	814.30	0.98	833	16249.33	19.51
simplify2	2329	2306	2408.95	1.11	2164	19153.34	8.85
Burns	14	14	0.30	0.02	14	19.37	1.38
Ricart	14	13	0.89	0.07	13	228.80	17.60
piVC	41	41	4.92	0.12	41	59.40	1.45

Table 5.2: Results on a selection of AUFLIA benchmarks

of the cases proved by CVC3 within a reasonable amount of time. The proof translation failed for several cases in the simplify2 family. Some of these cases failed due to time out, and others contain proof rules not yet fully supported by the translator. The average time spent on the proof translation was about 10 to 100 times the time spent by CVC3. However, for hard cases, the proof translation performs better. The following Table 5.3 shows the results on hard cases in the Simplify1 family. Hard cases are those cases for which CVC3 spent more than 20 seconds to find the proofs. There are two rows in the table. The first row shows the results when pre-processing is enabled in CVC3, and the second shows the results when pre-processing is disabled. Each row shows the

	CVC3		HOL Translation	
Prep	5	47.25	5	41.49
No prep	4	48.91	4	64.27

Table 5.3: Hard cases in proof translation

number of cases proved by CVC3, the average time spent by CVC3 to find the proof, the number of cases for which the proof translation was successful, and the average time for the translation.

As seen, the proof translation for hard cases is quite efficient. The proof translator is run under the interpreted interactive OCaml toplevel. It can be expected that a compiled proof translator would spend significantly less time on these cases.

5.5 Related work

Moskal [54] proposed a rewriting system for proof-checking of SMT solvers. His implementation focuses on fast proof-checking. However, a rewriting system is not sufficient for some complicated proof rules used in CVC3[1] and

[1] An example of such complicated rules in CVC3 is the one that normalize an arithmetic term.

would restrict the proof rules can be used in SMT solvers. The correctness of the implementation of the rewriting system was not addressed in [54]. The method proposed in this chapter ultimately provides a very strong guarantee of correctness, and essentially neither the SMT solver nor the proof translator need be trusted.

In previous work [51], initial efforts to combine HOL Light and CVC Lite was reported. That work emphasized using CVC Lite as an external decision procedure for HOL Light. This chapter emphasizes efficient translation and the use of HOL Light as a proof-checker for CVC3.

5.6 Conclusion

This chapter discussed a proof-translating method to improve the correctness of SMT solvers. Experiments showed that proof-translating is feasible. Future work includes proving more cases in the SMT-LIB, and improving the efficiency of the translator.

Chapter 6

Conclusion and future work

This dissertation proposes several novel techniques for solving quantified formulas in Satisfiability Modulo Theories.

For general heuristic-based quantifier reasoning, this dissertation proposes several new heuristics, and most novel among them is a heuristic called "instantiation level" that solves several challenges at the same time. The new heuristics have been implemented in CVC3. Experimental results show that a number of additional benchmark can be solved than could be solved by CVC3 before.

For complete instantiation, we propose a series of new fragments. We prove that a quantified formula in these new fragments are equi-satisfiable to

a ground formula that is constructed by instantiation. We discuss the condition under which the ground formulas is finite.

This dissertation also proposes a proof translator that translates proofs from CVC3 into HOL Light. Experimental results show that a proof translator is a feasible solution to improve the correctness of SMT solvers. When translating proofs, we found two faulty proof rules in CVC3 and two mislabeled benchmarks in the benchmark library SMT-LIB.

There are a lot of ideas to be explored in the future. For general quantifier reasoning, besides research on better heuristics and better data structures, one very interesting question is how to make heuristic instantiation as complete as possible. A natural extension would be to incorporate techniques from general quantifier reasoning and quantifier elimination into SMT solvers. As a first step, we could integrate a general first order logic solver into a SMT solver. Many more interesting problems would ensue. For example, when and how should the general first order solver be called? Under what circumstances is the procedure complete? Will this method be as efficient as other approaches? With so many questions unanswered, I believe that the combination of heuristic instantiation and complete methods will be a challenging and promising topic.

For complete instantiation, interesting future work include an empirical study of model-guided instantiation, searching for heuristics to prioritize the instantiation procedure, investigation of more decidable fragments that admit complete instantiation, etc. It will also be very interesting to re-examine the quantified benchmarks in SMT-LIB to see if some of them can be represented within the decidable fragments.

Bibliography

[1] The Satisfiability Modulo Theories Competition (SMT-COMP). `smtcomp.org`, 2008.

[2] Aharon Abadi, Alexander Rabinovich, and Mooly Sagiv. Decidable fragments of many-sorted logic. In Nachum Dershowitz and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, volume 4790 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 2007.

[3] Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Jiazhao Xu, and Lenore D. Zuck. Parameterized verification with automatically computed inductive assertions. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV*

2001, Paris, France, July 18-22, 2001, Proceedings, volume 2102 of *Lecture Notes in Computer Science*. Springer, 2001.

[4] F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.

[5] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*. Springer, 2006.

[6] Clark Barrett and Sergey Berezin. A proof-producing boolean search engine. In *Proceedings of the* 1^{st} *International Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '03)*, July 2003. Miami, Florida.

[7] Clark Barrett, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Splitting on demand in SAT Modulo Theories. In M. Hermann and

A. Voronkov, editors, *Proceedings of Logic for Programming, Artificial Intelligence and Reasoning (LPAR'06)*, volume 4246 of *Lecture Notes in Computer Science*, pages 512–526. Springer, 2006.

[8] Clark Barrett, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.

[9] Peter Baumgartner and Cesare Tinelli. The model evolution calculus as a first-order DPLL method. *Artificial Intelligence*, 172:591–632, 2008.

[10] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In E. Allen Emerson and Kedar S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, volume 3855 of *Lecture Notes in Computer Science*. Springer, 2006.

[11] Robert Brummayer and Armin Biere. Fuzzing and delta-debugging SMT solvers. In Bruno Dutertre and Ofer Strichman, editors, *Proceedings of the 7^{th} International Workshop on Satisfiability Modulo Theories (SMT '09)*, August 2009. Montreal, Canada.

[12] Robert N. Charette. Why software fails. *IEEE Spectrum*, 42(9):42–49, September 2005.

[13] Robert N. Charette. This car runs on code. *IEEE Spectrum*, 46(3):3, February 2009.

[14] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[15] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[16] Intel Corporation. 1994 annual report. `http://download.intel.com/museum/archives/pdf/1994_AR.pdf`, 1995.

[17] Electric Consumers Resource Council. The economic impacts of the August 2003 blackout. `http://www.elcon.org/Documents/EconomicImpactsOfAugust2003Blackout.pdf`, 2004.

[18] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approxima-

tion of fixpoints. In *Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[19] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[20] Leonardo de Moura. Private communication, 2006.

[21] Leonardo Mendonça de Moura and Nikolaj Bjørner. Efficient E-matching for SMT solvers. In Frank Pfenning, editor, *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2007.

[22] David Déharbe and Silvio Ranise. Satisfiability solving for software verification. *International Journal on Software Tools Technology Transfer*, 11(3):255–260, 2009.

[23] Ewen Denney, Bernd Fischer, and Johann Schumann. Using automated theorem provers to certify auto-generated aerospace software. In David A. Basin and Michaël Rusinowitch, editors, *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8,*

2004, Proceedings, volume 3097 of *Lecture Notes in Computer Science*, pages 198–212. Springer, 2004.

[24] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of ACM*, 52(3):365–473, 2005.

[25] Mark Dowson. The Ariane 5 software failure. *SIGSOFT Software Engineering Notes*, 22(2):84, 1997.

[26] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.

[27] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In J. W. de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherlands, July 14-18, 1980, Proceedings*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 1980.

[28] Herbert Enderton. *A Mathematical Introduction to Logic*. Academic Press, 2000.

[29] Cormac Flanagan, Rajeev Joshi, and James B. Saxe. An explicating theorem prover for quantified formulas. Technical Report HPL-2004-199, HP Intelligent Enterprise Technologies Laboratory, 2004.

[30] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, June 2002. ACM.

[31] Pascal Fontaine and E. Pascal Gribomont. Decidability of invariant validation for parameterized systems. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*. Springer, 2003.

[32] Food and Drug Administration. FDA statement on radiation overexpo-

sures in panama. http://www.fda.gov/Radiation-EmittingProducts/RadiationSafety/Alertsand%Notices/ucm116533.htm.

[33] Yeting Ge and Clark Barrett. Proof translation and smt-lib benchmark certification: A preliminary report.

[34] Yeting Ge, Clark Barrett, and Cesare Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In Frank Pfenning, editor, *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*. Springer, 2007.

[35] Yeting Ge and Leonardo Mendonça de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009.*, volume 5643 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2009.

[36] Silvio Ghilardi, Enrica Nicolini, Silvio Ranise, and Daniele Zucchelli. Deciding extensions of the theory of arrays by integrating decision procedures and instantiation strategies. In Michael Fisher, Wiebe van der

Hoek, Boris Konev, and Alexei Lisitsa, editors, *Logics in Artificial Intelligence, 10th European Conference, JELIA 2006, Liverpool, UK, September 13-15, 2006, Proceedings*, volume 4160 of *Lecture Notes in Computer Science*. Springer, 2006.

[37] Mike Gordon. From LCF to HOL: a short history. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 169–186. The MIT Press, 2000.

[38] Peter Habermehl, Radu Iosif, and Tomás Vojnar. A logic of singly indexed arrays. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, Doha, Qatar, November 22-27, 2008. Proceedings*, volume 5330 of *Lecture Notes in Computer Science*. Springer, 2008.

[39] Peter Habermehl, Radu Iosif, and Tomás Vojnar. What else is decidable about integer arrays? In Roberto M. Amadio, editor, *Foundations of Software Science and Computational Structures, 11th International Conference, FOSSACS 2008, Budapest, Hungary, March 29 - April 6, 2008.*, volume 4962 of *Lecture Notes in Computer Science*. Springer, 2008.

[40] Thomas C. Hales. http://sites.google.com/site/thalespitt/.

[41] Joseph Y. Halpern. Presburger arithmetic with unary predicates is π_1^1 complete. *Journal of Symbolic Logic*, 56(2):637–642, 1991.

[42] John Harrison. http://www.cl.cam.ac.uk/∼jrh13/hol-light/.

[43] John Harrison. HOL Light: A tutorial introduction. In Mandayam K. Srivas and Albert John Camilleri, editors, *Formal Methods in Computer-Aided Design, First International Conference, FMCAD '96, Palo Alto, California, USA, November 6-8, 1996, Proceedings*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer, 1996.

[44] John Harrison. Formal verification at Intel. In *18th IEEE Symposium on Logic in Computer Science (LICS 2003), 22-25 June 2003, Ottawa, Canada, Proceedings*, pages 45–. IEEE Computer Society, 2003.

[45] John Harrison. Towards self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2006.

[46] Mike Hinchey, Michael Jackson, Patrick Cousot, Byron Cook, Jonathan P. Bowen, and Tiziana Margaria. Software engineering and formal methods. *Communications of the ACM*, 51(9):54–59, 2008.

[47] J.N. Hooker, G. Rago, V. Chandru, and A. Shrivastava. Partial instantiation methods for inference in first order logic. *Journal of Automated Reasoning*, 28(4):371–396, 2002.

[48] Carsten Ihlemann, Swen Jacobs, and Viorica Sofronie-Stokkermans. On local reasoning in verification. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*. Springer, 2008.

[49] K. Rustan M. Leino. Extended static checking. In David Gries and Willem P. de Roever, editors, *Programming Concepts and Methods, IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods (PROCOMET'98) 8-12 June 1998, Shelter Island, New York,*

USA, volume 125 of *IFIP Conference Proceedings*, pages 1–2. Chapman & Hall, 1998.

[50] John McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

[51] Sean McLaughlin, Clark Barrett, and Yeting Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. In Alessandro Armando and Alessandro Cimatti, editors, *Proceedings of the* 3^{rd} *Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '05)*, volume 144(2) of *Electronic Notes in Theoretical Computer Science*, pages 43–51. Elsevier, January 2006. Edinburgh, Scotland.

[52] Scott McPeak and George C. Necula. Data structure specifications via local equality axioms. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 476–490. Springer, 2005.

[53] Michal Moskal. Fx7, 2006. `http://nemerle.org/ malekith/smt/en.html`.

[54] Michal Moskal. Rocket-fast proof checking for SMT solvers. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Budapest, Hungary, March 29-April 6, 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 486–500. Springer, 2008.

[55] Michal Moskal, Jakub Lopuszanski, and Joseph Kiniry. E-matching for fun and profit. In Sava Krstic and Albert Oliveras, editors, *Proceedings of the 5^{th} International Workshop on Satisfiability Modulo Theories (SMT '07)*, pages 25–35, July 2007. Berlin, Germany.

[56] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, November 2006.

[57] David A. Plaisted and Yunshan Zhu. Ordered semantic hyper linking. *Journal of Automated Reasoning*, 25(3):167–217, 2000.

[58] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15(2-3):91–110, 2002.

[59] Mark E. Stickel. Automated deduction by theory resolution. *Journal of Automated Reasoning*, 1(4):333–355, 1985.

[60] A. Stump and D. Dill. Faster proof checking in the Edinburgh logical framework. In A. Voronkov, editor, *Proceedings of the* 18^{th} *International Conference on Automated Deduction (CADE '02)*, volume 2392 of *Lecture Notes in Computer Science*, pages 185–222. Springer, 2002.

[61] G. Sutcliffe. The IJCAR-2004 Automated Theorem Proving Competition. *AI Communications*, 18(1):33–40, 2005.

[62] G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[63] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Standards and Technology, May 2002.

[64] Grigori Tseitin. On the complexity of derivation in propositional calculus. In A. O. Silenko, editor, *Structures in Constructive Mathematics and Mathematical Logic*, volume II, pages 115–1252. Consultants Bureau, New York, 1968.

[65] Chao Wang, Aarti Gupta, and Malay K. Ganai. Predicate learning and selective theory deduction for a difference logic solver. In Ellen Sentovich, editor, *Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006*, pages 235–240. ACM, 2006.

[66] Tjark Weber. Efficiently checking propositional resolution proofs in Isabelle/HOL. In C. Benzmueller, B. Fischer, and G. Sutcliffe, editors, *Proceedings of the 6th International Workshop on the Implementation of Logics*, volume 212 of *CEUR Workshop Proceedings*, 2006.

[67] Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobald, and Dalibor Topic. SPASS Version 2.0. In Andrei Voronkov, editor, *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings*, volume 2392 of *Lecture Notes in Computer Science*, pages 275–279. Springer, 2002.

[68] Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In Andrei Voronkov, editor, *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen,*

Denmark, July 27-30, 2002, Proceedings, volume 2392 of *Lecture Notes in Computer Science*, pages 295–313. Springer, 2002.