Satisfiability Modulo Software

Michał Jan Moskal
PhD Thesis

Supervisor: Prof. Leszek Pacholski

Institute of Computer Science
University of Wrocław
ul. Joliot-Curie 15
50-383 Wrocław, Poland

Wrocław 2009

Abstract

Formal verification is the act of proving correctness of a hardware or software system using formal methods of mathematics. In the last decade formal hardware verification has seen an increasing usage of Satisfiability Modulo Theories (SMT) solvers. SMT solvers check satisfiability of first-order formulas, where certain symbols are interpreted according to background theories like integer or bit-vector arithmetic. Since the formulas used to encode correctness of hardware design are mostly quantifier-free, SMT solvers are built as theory-aware extensions of propositional satisfiability solvers. As a consequence, SMT solvers do not "naturally" support quantified formulas, which are needed for verification of complex software systems. Thus, while SMT solvers are already an industrially viable tool for formal hardware verification, software applications are not as developed.

This thesis focuses on both the software verification specific problems in the construction of SMT solvers, as well as SMT-specific parts of a software verification system. On the SMT side, we present algorithms for efficient non-ground reasoning through quantifier instantiation and techniques for proof generation and proof checking for quantifier-rich software verification problems. On the verification tool side, we present methods for transforming programs into formulas in a solver-friendly way, with particular emphasis on design of annotations guiding the SMT solver through the non-ground part of the problem.

The theoretical developments presented here were experimentally validated in implementations of state-of-the-art tools: an SMT solver and a verifier for concurrent C programs.

Systemy SMT w formalnej weryfikacji oprogramowania

Michał Jan Moskal
Praca doktorska

Promotor: Prof. Leszek Pacholski

Instytut Informatyki
Uniwersytet Wrocławski
ul. Joliot-Curie 15
50-383 Wrocław

Wrocław 2009

Streszczenie

Formalna weryfikacja to proces dowodzenia poprawności oprogramowania lub projektu układu scalonego (sprzętu) z użyciem formalnych metod matematyki. W ostatnim dziesięcioleciu w weryfikacji sprzętu coraz częściej używane były systemy SMT (ang. Satisfiability Modulo Theories). Sprawdzają one spełnialność formuł pierwszego rzędu, gdzie pewne symbole są interpretowane zgodnie z teoriami, np. zgodnie z arytmetyką na liczbach całkowitych lub arytmetyką maszynową na ciągach bitów. Ponieważ formuły używane do kodowania poprawności sprzętu są przeważnie pozbawione kwantyfikatorów, systemy SMT budowane są na bazie systemów sprawdzających spełnialność formuł rachunku zdań, rozszerzając je o procedury decyzyjne dla teorii. Z tego powodu systemy SMT nie obsługują formuł z kwantyfikatorami w "naturalny" sposób, a formuły takie są niezbędne w weryfikacji skomplikowanego oprogramowania. Dlatego też, pomimo, że systemy SMT są szeroko używane w przemyśle elektronicznym, zastosowania w weryfikacji oprogramowania nie są tak rozwinięte.

Niniejsza rozprawa skupia się na problemach specyficznych dla weryfikacji w konstrukcji systemów SMT oraz na aspektach systemów weryfikujących oprogramowanie mających związek z SMT. Od strony SMT prezentowane są metody generowania i sprawdzania dowodów niespełnialności formuł kodujacych poprawność programów oraz efektywne algorytmy używane w procesie poszukiwania takiego dowodu z użyciem instancjonowania kwantyfikowanych formuł. Od strony weryfikacji opisywane są metody kodowania poprawności programów w taki sposób, by system SMT efektywnie je przetworzył, ze szczególnym uwzględnieniem instrukcji specyfikujących, jak instancjonować kwantyfikowane formuły.

Teoretyczne wyniki prezentowane w niniejszej rozprawie zostały eksperymentalnie potwierdzone w zaimplementowanym systemie SMT oraz weryfikatorze równoległych programów napisanych w języku C.

Acknowledgments

I would like to thank Nikolaj Bjørner, Radu Grigore, Mikoláš Janota, Joe Kiniry, Rustan Leino, Kuba Łopuszański, Leonardo de Moura, Thomas Santen, Wolfram Schulte, and Stephan Tobies for their comments and support during development of this thesis. Further, I would like to thank my advisor, prof. Leszek Pacholski, for his continuous encouragement and guidance.

My PhD work was partially supported by Polish Ministry of Science and Education grant 3 T11C 042 30.

Contents

Chapter 1

Introduction and Summary

> When we had no computers, we had no programming problem either. When we had a few computers, we had a mild programming problem. Confronted with machines a million times as powerful, we are faced with a gigantic programming problem.
>
> *Edgar W. Dijkstra*

Given the fact that software is nowadays used in most areas of human life, improving reliability of software is of crucial social importance. Moreover costs of software defects are measured in billions of dollars[1], which, given the size of the software development business (about half a trillion dollars in 2008), is not very surprising. Improving reliability of software is thus also of crucial economic importance.

The ultimate software reliability can, however, only come through formal verification. To quote Edgar W. Dijkstra again, this time from his Turing Award lecture (Dijkstra, 1972):

> Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

It thus seems very unfortunate that, after nearly forty years of academic developments in verification, most of the software in use, including critical aviation or medical systems, has not undergone formal verification.

Lack of industrial acceptance of software verification can be attributed to inadequate automation of verification tools: clearly, if detecting a bug was as easy as pushing some button, everyone would do it. On the other hand, if we look at the hardware world, we notice a success story for formal methods. Especially since the introduction of very efficient model checkers and propositional satisfiability solvers, formal verification has gained much traction for chip design. The key reason seems to be that model checking and SAT-solving are a push-button technologies: there is no need to give guidance to an interactive theorem prover.

[1] This thesis uses American English, in particular a billion is understood as 10^9.

In recent years, in hardware verification, the usage of Satisfiability Modulo Theories (SMT) solvers was increasing. SMT solvers check satisfiability of a first-order formula, where certain function and constant symbols are interpreted according to a set of background theories. These theories typically include integer or rational arithmetic, bit vectors, and arrays.

The formulas in hardware verification are usually quantifier-free, thus the SMT logics are often seen as extensions of propositional logic. In fact, it is very common to build SMT solvers on top of an existing propositional SAT solvers.

Given SMT logics, expressible enough to encode software verification problems, an SMT solver seems to be a good candidate for the underlaying engine of a software verification tool: SMT is push-button technology, with a significant development effort (both academic and industrial) behind it.

This thesis focuses on aspects of SMT solving that are specific to software verification, in particular quantifier instantiation. We approach the topic from two perspectives.

First, from the perspective of a user of an SMT solver: we describe how quantifier instantiation techniques are used to build a custom theory, encoding semantics of the C programming language for a software verifier called VCC. We also report on results of applying VCC in the context of a large operating-system verification project, in particular focusing on specific demands of software verification tools placed on the SMT solver.

Second, we talk about quantifier instantiation techniques from the perspective of an author of an SMT solver. We talk about term indexing techniques used to speed-up quantifier instantiation, as well as efficient proof checking of large, quantifier-rich proofs. Both the term indexing, as well as proof generation were implemented in an SMT solver called Fx7.

The author of this thesis is the principal author of Fx7 (Moskal, 2007) and VCC (Cohen et al., 2009a).

This chapter gives some short overview of SMT solvers (Section 1.1), sketches the quantifier instantiation techniques and their usage in software verification (Section 1.2), and finally gives an outline of the thesis (Section 1.5).

1.1 History of SMT Solvers

Historically SMT solvers date back to the late 1970s and early 1980s when Greg Nelson and Derek Oppen introduced a system called Simplify for use with Stanford Pascal Verifier (Luckham et al., 1979). Later, during 1990s, David Dill's group at Stanford developed the SVC prover (Barrett et al., 1996) and also some decision procedures have been incorporated into the higher order PVS prover (Owre et al., 1992).

Most notably however David Detlefs, Greg Nelson, and James Saxe, in the course of ESC/Modula-3 (Detlefs et al., 1998) and ESC/Java (Flanagan et al., 2002) projects, developed another system confusingly also called Sim-

plify (Detlefs et al., 2005)[2]. On software verification queries the Simplify system remained unbeaten for the following ten years.

Simultaneously, in the early 2000s new, orders of magnitude more efficient, propositional satisfiability solvers (hereafter referred to as SAT solvers) were introduced. The most prominent ones were Chaff (Malik et al., 2001) (for the breakthroughs in the solving technology) and later MiniSAT (Eén and Sörensson, 2003) (for being extremely efficient, yet having very small, readable code base; this resulted in many derivative solvers).

Following suit several SMT solvers has been developed. The usual method of engineering such systems was to add theory reasoning to an existing SAT solver. These included Verifun (Flanagan et al., 2004), MathSAT (Bozzano et al., 2005), UCLID (Bryant et al., 2002), CVC (Stump et al., 2002) and CVC Lite (Barrett and Berezin, 2004). Also somewhere around that time the term Satisfiability Modulo Theories was first used. Later a new wave of systems were developed using tighter integration between the SAT solvers and decision procedures, giving another boost to performance. These include Barcelogic tools (Ganzinger et al., 2004) and Yices (Dutertre and de Moura, 2006). The most recent systems include also Ergo (Conchon et al., 2007), Spear (Babić and Hutter, 2008), DPT, CVC3 (Barrett and Tinelli, 2007), Fx7 (Moskal, 2007), and Z3 (de Moura and Bjørner, 2008). Lately the hot topics in SMT are extending the solvers to handle not only ground queries, but also ones with quantifiers as well as supporting various bit vector theories.

1.1.1 SMT-LIB

During recent years the development of different SMT solvers was stimulated by both growing industrial and academic interest as well as the SMT-LIB initiative (Ranise and Tinelli, 2006). SMT-LIB is a library of publicly available SMT problems (benchmarks), resulting mostly from industrial applications in hardware and software verification. Each year authors can submit solvers to the worldwide competition, the SMT-COMP. Solvers then compete on problems drawn randomly from SMT-LIB. This is similar to the TPTP (Thousands of Problems for Theorem Provers; Sutcliffe and Suttner 1998) library and the CADE Automated Theorem Proving System Competition (CASC), organised for first-order automatic theorem provers (cf. Section 1.3 for an explanation of differences between ATP and SMT systems).

The SMT-LIB is organised in different divisions, based on the theories used in benchmarks. The ones that are of the most relevance to this thesis is AUFLIA (containing around 6500 problems), which stands for Arithmetic, Uninterpreted Functions and Linear Integer Arithmetic and UFNIA (Uninterpreted Functions and Non-Linear Integer Arithmetic; it contains around 2000 problems). They contain formulas with quantifiers (most other divisions

[2]From now on we will use the name Simplify to refer to this system, rather than to the one used with Stanford Pascal Verifier.

contain quantifier-free formulas) resulting from software verification[3], mostly using ESC/Java, Boogie, VCC, and HAVOC tools. All UFNIA and over a half of the AUFLIA benchmarks were translated to the SMT format and submitted to SMT-LIB by the author of this thesis.

1.2 It's All About Quantifiers

SMT problems stemming from software verification are quite different from the problems resulting from hardware verification. In particular, most hardware verification problems are ground, while in software verification quantified formulas are often used to encode the custom theory describing semantics of the programming language being verified. Efficient handling of quantified formulas is thus of crucial importance for software verification tools. It has however proven difficult to go from the realm of (usually) NP-complete ground SMT problems to the undecidable class of quantified SMT problems.

In fact it was only in 2007 when the 10-year old Simplify was decisively beaten by Z3 on AUFLIA division of SMT-LIB. Before only a few systems (to be precise Yices, Zap2, Fx7, CVC Lite and CVC3) supported quantified queries with linear integer arithmetic and while some of them were able to outperform Simplify on some of the benchmarks, they were failing on others. In 2007 there were only four systems competing in the AUFLIA division (ordered by results: Z3, Fx7, CVC3 and Yices), whereas ten competed in the quantifier-free divisions. 2008 has seen three solvers (Z3, CVC3 and Alt-Ergo) in AUFLIA and thirteen in quantifier-free divisions, and in 2009, with Z3 not participating, only CVC3 run in AUFLIA.

The usual method used in SMT solvers for dealing with non-ground problems is based on instantiation. The quantified formulas are instantiated, in a way that the instantiations share subterms with ground part of the problem. The instantiation process is guided by annotations attached to quantified formulas, so called "triggers", and the procedure, making use of triggers, is called E-matching. Chapter 2 of this thesis provides a running example for solving a SMT query using E-matching, while Chapter 4 contributes a formal treatment of the problem, as well as two novel E-matching algorithms.

The triggers can be either explicitly supplied to the SMT solver, or the solver can select them using a simple heuristic. The explicit triggering has proven to be a powerful, if somewhat arcane, tool for building custom SMT theories, like the one describing a particular verification methodology. The triggering annotations can be viewed as a logic programming language used to implement a theory to be executed by the SMT solver. Of course one could also implement the theory inside of an SMT solver, which would likely be much more efficient, but the implementation would be much harder. Given how fast such a theory evolves during development of a verification tool, it

[3]The term *software verification* will refer to formal, static software verification throughout this thesis.

seems counterproductive in most cases. Chapter 3 of this thesis surveys most commonly used encoding techniques and contributes a few novel ones.

No matter how complex the triggers are, the instantiation procedure is always logically sound. So is the initial skolemization and clausification that SMT problems are subject to, as well as the resolution rule used in the proof search. Still, to guard against bugs in the SMT solver and possibly provide ground for Proof-Carrying Code scenarios for verification with SMT solvers, it would be good to have the solver produce a proof of unsatisfiability and later independently check such a proof. However, modern SMT solvers can produce proofs of enormous size very quickly. Moreover, the initial skolemization, required since the software verification queries are quantifier-rich, requires non-local checking of the proof. Thus, such proofs require specialized proof checking technology. We cover one solution to this problem in Chapter 5.

1.3 SMT vs. ATP

There is certain overlap between what is referred to as SMT solvers and first-order Automated Theorem Proving (ATP) systems. Both check satisfiability (or equivalently validity) of first-order formulas. However, unlike in SMT, the logic considered in ATP systems is usually just plain first-order logic with or without equality, but always without any additional theories. On the other hand, SMT solvers usually accept only ground (quantifier-free) problems, whereas ATP systems use reasoning methods like resolution or superposition, which are complete also in presence of non-ground formulas. Still, there are SMT solvers supporting quantifiers, and there are ATP solvers handling some background theories. The crucial differences are thus in the application areas and search methods.

SMT solvers evolved from propositional SAT solvers to support more efficient encoding of specific (hardware or software) verification problems. ATP systems are built to move forward automated deduction, in particular of mathematical theorems. In fact the TPTP library contains a number of open mathematical problems[4]. Thus, as a rule of thumb, SMT solvers are good at huge, shallow problems (usually from decidable classes), whereas ATP systems are good at small, deep problems (where the general classes of problems are never decidable). As a result, the constraints on search methods are different.

ATP systems usually do some sort of saturation search, where new formulas are derived from the input. Such process is guaranteed to finish if the formula is unsatisfiable but not in the opposite case.

SMT solvers try to build a model for the input formula, step by step. This is done in hope that the model pointing a verification failure will likely be small, and if there is no model, we will be able to quickly move through such small models and discharge them. The user expects an answer either pointing to a candidate model, or the information that there is none. Both

[4] To be fair, there have been an influx of verification problem in the TPTP library in recent years. This is however nowhere near the SMT-LIB.

answers need to be given quickly. As a consequence, when SMT solvers need to deal with quantified formulas some heuristic is employed. This sacrifices completeness for the "common"-case performance. From automated deduction point of view, this does not seem very elegant. However, verification tools are very often incomplete for several other reasons, having to do with imprecise or restricted encoding of the verified system. Thus, one additional source of incompleteness does not look so bad anymore, especially when one can trade it for speeding up the interactive feedback loop.

1.4 Previous Publication of Presented Results

Chapters 3, 4, and 5 contain material previously published in the following papers:

Programming with Triggers. Michał Moskal. 7th International Workshop on Satisfiability Modulo Theories (SMT 2009), ACM International Conference Proceeding Series, to appear.

E-matching for Fun and Profit. Michał Moskal, Jakub Łopuszański, and Joseph R. Kiniry. 5th International Workshop on Satisfiability Modulo Theories (SMT 2007), and Electronic Notes in Theoretical Computer Science, vol. 198.2, pp. 19–35, Elsavier 2008.

Rocket-fast Proof Checking for SMT Solvers. Michał Moskal. 14th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008). Lecture Notes in Computer Science vol. 4963, pp. 486–500, Springer 2008.

1.5 Structure of This Thesis

Chapter 2: Deductive Verification with SMT introduces deductive verification and related concepts. We also cover the basics of an SMT solver operation on software verification input.

Chapter 3: Programming With Triggers describes various common techniques used for defining theory axiomatizations, in particular the theories describing semantics of programming languages.

Chapter 4: E-matching introduces two novel algorithms used for finding quantifier instantiations during SMT proof search, so an axiomatization like the one developed in Chapter 3 can be efficiently used.

Chapter 5: Proof Checking introduces a technique of proof checking using an small and efficient, yet extensible proof checker using first-order term rewriting. Thus, if the SMT solver produces a proof that a piece of software is correct (modulo axiomatization developed in Chapter 3, thus being large and

involving many quantified formulas), we can actually check the proof faster then the SMT solver could produce it.

Chapter 6: Conclusions concludes.

Chapter 2

Deductive Verification with SMT

This thesis focuses on areas of interaction between SMT solvers and software verification tools, in particular on parts of SMT solvers that are specific for software verification and parts of verification tools that are needed to make use of SMT solvers. On the other hand, the general architecture of SMT solvers or verification tools are not topics of this thesis. Nevertheless, to set up a context, this chapter briefly discusses relevant concepts of deductive verification, and provides a high-level overview of the search procedure in a modern SMT solver.

2.1 Deductive Verification

This section describes the deductive verification architecture commonly employed in software verification tools, which make use of SMT solvers. The basic idea is to encode correctness of a program in a logical formula, called the *verification condition* (VC), and use a theorem prover (either automatic, like an SMT solver, or interactive) to check validity of the formula.

The meaning of "correctness" depends on design goals of a particular tool, but usually includes absence of runtime errors (division by zero, null pointer dereference, accessing unmapped memory, overflows, etc.), and also adherence to some kind of specification, either supplied inline, in the program text, or externally. The correctness is then checked modulo some assumptions, regarding the semantics of the language being verified, which are not enforced by the verification tool. When all these assumptions are thought to be guaranteed by the compiler and runtime system of the programming language being verified, the verification is said to be *sound* (with respect to the properties checked). Otherwise the assumptions are thought to hold for some sort of "common" executions, and problems found for such paths indicate likely bugs in the program, and thus such activity is called *bug-finding*. The particular assumptions might have to do, e.g., with pointer aliasing, type-correct access to memory, or integer overflows.

There exist multiple tools, following the deductive verifier architecture using SMT solvers, for either sound verification or bug-finding. For example, Java developers have Krakatoa (Marché et al., 2004) and JACK (Barthe et al., 2006) for verification and ESC/Java (Flanagan et al., 2002) for bug-finding, C# programmers can make use of Spec# (Barnett et al., 2005) verifier, while C users can employ VCC (Cohen et al., 2009a) for verification, HAVOC (Lahiri and Qadeer, 2008) for bug-finding and Frama-C (Moy, 2009) for either. Spec#, VCC and HAVOC share an intermediate intermediate language called Boogie (DeLine and Leino, 2005; Leino, 2009), while Krakatoa and Frama-C share the intermediate language Why (Filliâtre, 2003).

The general architecture for all those tools, especially the ones sharing a common intermediate language, is very similar. The input program is translated into the intermediate language. The verification condition generator (e.g., the Boogie or Why tool) generates formulas, and sends to one or more theorem provers for checking. The theorem provers can be either automatic (like SMT solvers or ATP systems) or interactive (e.g., Böhme et al. 2009). In case the formulas are invalid, the SMT solvers return some description of a model where the negation of the formula is true. Such a model is treated as a counterexample, a description of a program run where the program goes wrong. In case the formula is valid, the program is assumed to be correct. In case of Proof-Carrying Code scenarios, a proof of correctness might be extracted and checked independently (see Chapter 5 for a description of a proof checking technology useful in such a case). Both kind of outcomes are reported to the user of the verification tool.

2.2 An Example

We shall now go through a simple C program, see how its correctness is encoded as an SMT formula, and how the SMT solver checks its satisfiability.

```
void absolute(int *x)
{
  if (*x < 0)
    *x = -(*x);
  assert(*x >= 0);
}
```

The program reads an integer from the memory location pointed to by `x`, and if this integer is negative, it writes its negation back at the same location. Then, the **`assert`**`(...)` statement expresses programmer belief, that after such operation the integer at `x` is non-negative. A condition in the **`assert`**`(...)` statement is meant to be checked by the verifier. That is, when the verifier proves correctness of the program, the programmer expects this condition to follow from the context and wants to see an error message if this is not the case. A condition under the **`assume`**`(...)` statement (used later in the text) would not be checked, but just added to the context. The operational meaning of both statements is the same: they are supposed to hold for any

execution of the program.

A verifier would usually generate a number of implicit assertions for a program. In C one would expect assertions talking about validity of memory location pointed to by `x`. For brevity we skip those, so by correctness of the program we mean that the explicit assertion never fails. The first step toward establishing correctness of this program is to make the heap encoding explicit. We do that by introducing a global variable representing the heap, and substituting heap accesses with applications of functions `rd(...)` and `wr(...)`, for reading and updating the heap respectively. An axiom, we are going to use to reason about a heap update is:

$$\forall H, p, v.\ \mathrm{rd}(\mathrm{wr}(H, p, v), p) = v$$

If one writes v at heap location p, then reading from the updated heap at p will yield v. There are more axioms describing the heap, these are described in detail further in Section 3.2.2. The program, after making the heap explicit, looks like:

```
heap H;

void absolute(int *x)
{
  if (rd(H, x) < 0) {
    H = wr(H, x, -rd(H, x));
  }
  assert(rd(H, x) >= 0);
}
```

Next assignments are removed from the program, by introducing fresh variables:

```
void absolute(int *x)
{
  if (rd(H0, x) < 0) {
    assume(H1 == wr(H0, x, -rd(H0, x)));
  } else {
    assume(H1 == H0);
  }
  assert(rd(H1, x) >= 0);
}
```

Further, the conditional statements can be replaced with non-deterministic choice:

```
void absolute(int *x)
{
  if (*) {
    assume(rd(H0, x) < 0);
    assume(H1 == wr(H0, x, -rd(H0, x)));
  } else {
    assume(!(rd(H0, x) < 0));
    assume(H1 == H0);
```

```
    }
    assert(rd(H1, x) >= 0);
}
```

The program is correct iff for any path through non-deterministic choices, for any model satisfying assumptions on that path, until a particular assertion, the model also satisfies the assertion. The conjunction of the following three formulas, is unsatisfiable iff the program is correct:

$$\forall H, p, v.\ \mathrm{rd}(\mathrm{wr}(H, p, v), p) = v \tag{2.1}$$

$$\begin{aligned}(\mathrm{rd}(H_0, x) < 0 \ \wedge H_1 = \mathrm{wr}(H_0, x, -\mathrm{rd}(H_0, x))) \vee \\ (\neg(\mathrm{rd}(H_0, x) < 0) \wedge H_1 = H_0)\end{aligned} \tag{2.2}$$

$$\neg(\mathrm{rd}(H_1, x) \geq 0) \tag{2.3}$$

Conjunct (2.1) is the axiom specifying behavior of the heap described earlier. Conjunct (2.2) encodes the semantics of the conditional statement: either the condition was true, and the new heap is constructed by updating the old heap at x, or the condition was not true, and the new heap equals to the old heap. Finally, conjunct (2.3) says that the assertion is violated. A model for the conjunction of the three formulas corresponds to a program execution, where the assertion is violated. If such a model does not exists (i.e., the formula is unsatisfiable), then the program is correct.

Let us now examine how the SMT solver establishes unsatisfiability of the conjunction. At the high level, the solver searches through different models that might satisfy the formula. If a model satisfies the formula, it is returned as a counterexample. Otherwise, a conflict clause, explicitly conflicting with the current model, is created and conjoined to the input formula. The conflict clause is a tautology modulo background theories, so adding it to the input formula does not change the satisfiability status of the input formula, but it does narrow down the search space, as the current model (and possibly other models) are propositionally excluded from further search. After adding a conflict clause a new model is considered, or if no model can be constructed, the formula is considered to be unsatisfiable. This procedure is further explained below.

The search for a model is first performed for a boolean abstraction of the formula. For example, the initial view of our input formula in the SMT solver is:

$$Q \wedge ((C \wedge A_1) \vee (\neg C \wedge A_2)) \wedge \neg A_3$$

where Q is an abstraction of the heap axiom, C is an abstraction of the condition and A_i are abstractions of assertions and assumptions. The SMT solver employs an embedded boolean SAT-solver to find a satisfying boolean assignment to such an abstraction. Let us assume, that the satisfying assignment found is:

$$Q = \mathit{true}, C = \mathit{false}, A_2 = \mathit{true}, A_3 = \mathit{false}$$

We can interpret this as the solver following the "else" branch. This assignment is translated back to a *monome*: a set of literals true in the current

boolean model. Our monome is:

$$\{\forall H, p, v.\ \mathrm{rd}(\mathrm{wr}(H, p, v), p) = v,\ \neg(\mathrm{rd}(H_0, x) < 0),\ H_1 = H_0,\\ \neg(\mathrm{rd}(H_1, x) \geq 0)\}$$

The monome is then communicated to the decision procedures, to see if a model can be built for it. The arithmetic decision procedure (DP) gets an abstraction of the part of the monome that is relevant for it, which in our case is: $\neg(a_0 < 0)$, $\neg(a_1 \geq 0)$. The structure of terms $\mathrm{rd}(H_0, x)$ and $\mathrm{rd}(H_1, x)$ is hidden behind variables a_0 and a_1 respectively. The uninterpreted function DP[1] gets the literal $H_1 = H_0$, and also is told that terms $\mathrm{rd}(H_0, x)$ and $\mathrm{rd}(H_1, x)$ might be of interest to other theories. By congruence it infers $\mathrm{rd}(H_0, x) = \mathrm{rd}(H_1, x)$, which is then communicated to the arithmetic DP (as $a_0 = a_1$). At this point the arithmetic DP signals a conflict: the part of monome communicated to it cannot have a model. The reasons for conflict are then analysed, and the solver finds that the three literals of the initial monome except for the heap axiom contributed to the conflict. Thus the conflict clause is:

$$(\mathrm{rd}(H_0, x) < 0) \vee \neg(H_1 = H_0) \vee (\mathrm{rd}(H_1, x) \geq 0)$$

Note that it is tautology under uninterpreted functions and arithmetic. The conflict, after abstracting it to $C \vee \neg A_2 \vee A_3$, is conjoined to the initial input formula abstraction and passed to the internal SAT solver which restarts the search for a boolean model.

The only remaining boolean model is $Q \wedge C \wedge A_1 \wedge \neg A_3$. The ground part of the monome:

$$\{\mathrm{rd}(H_0, x) < 0,\ H_1 = \mathrm{wr}(H_0, x, -\mathrm{rd}(H_0, x))),\ \neg(\mathrm{rd}(H_1, x) \geq 0)\}$$

has a model, therefore the ground DPs will not find a conflict. Thus, the quantified formula needs to be instantiated, by adding an *instantiation tautology* to the input formula. An instantiation tautology is a formula of the form:

$$(\forall x_1, \ldots, x_n.\ \psi(x_1, \ldots, x_n)) \Rightarrow \psi(t_1, \ldots, t_n)$$

for some $t_1, \ldots, t_n$. The selection of particular $t_1, \ldots, t_n$ (or equivalently a substitution mapping x_i to t_i) is performed heuristically, as described in the next section. The particular instantiation tautology we need for our problem is:

$$(\forall H, p, v.\ \mathrm{rd}(\mathrm{wr}(H, p, v), p) = v) \Rightarrow\\ \mathrm{rd}(\mathrm{wr}(H_0, x, -\mathrm{rd}(H_0, x)), x) = -\mathrm{rd}(H_0, x)$$

This tautology is abstracted to $Q \Rightarrow I_1$, and thus the input formula abstraction now looks like:

$$Q \wedge ((C \wedge A_1) \vee (\neg C \wedge A_2)) \wedge \neg A_3 \wedge\\ (C \vee \neg A_2 \vee A_3) \wedge (Q \Rightarrow I_1)$$

[1] Everything that is not pure equality and propositional connectives is treated as theory in SMT. This includes the uninterpreted function theory, which could be axiomatized with $\forall x_1, ..., x_n, y_1, ..., y_n . x_1 = y_1 \wedge ... \wedge x_n = y_n \Rightarrow f(x_1, ..., x_n) = f(y_1, ..., y_n)$ for every function symbol f with arity n.

The SAT solver looks for a model for the new formula, and finds: $Q \wedge C \wedge A_1 \wedge \neg A_3 \wedge I_1$. Using reasoning similar to the one before, we get a conflict clause of $\neg C \vee \neg A_1 \vee A_3 \vee \neg I_1$, which after conjoining to the input formula, makes it unsatisfiable at the boolean level, and thus the SMT solver concludes that the original input formula was also unsatisfiable.

This description of the proof search highlights the most important features of SMT solvers:

1. Handling of the propositional structure of the formula is very much like in the modern, extremely efficient, propositional SAT solvers.

2. Multiple DPs built into the SMT solvers need to cooperate, in our example the uninterpreted function DP propagated equality to the linear arithmetic DP. This is usually achieved using some variant of Nelson-Oppen combination (Nelson and Oppen, 1979).

3. The search space is narrowed by the conflict clauses that the SMT solver learns during the search.

4. The quantified formulas are handled by instantiation and therefore we need some heuristic to decide how to instantiate. The most commonly used heuristics is called *E-matching* and is described in the following section.

2.3 E-matching

The E-matching instantiation heuristics is based on the idea that the instantiation tautologies should have some terms in common with the current ground monome. Therefore, particular subterms[2] of the body of the quantified formulas are designated as *triggers*. The solver then looks for substitutions, which make the triggers equal to terms in the current ground monome, modulo equality relation imposed by the current ground monome.

More precisely, a term is *active* in a monome, if it occurs in one of the quantifier-free literals. For the purpose of matching, some predicates are also treated as terms (it usually applies to uninterpreted predicates, i.e., ones not coming from a theory built into an SMT solver, but details vary between implementations). The equality relation imposed by the monome is the least congruence relation, containing all equalities positively present in the monome.

A trigger is a set of non-ground terms. A trigger is said to *match* in the monome M with the substitution σ, if for each term t in the trigger, $\sigma(t)$ is equal, modulo the equality relation imposed by M, to some active term in M. A formal definitions of triggers and matching is presented in Chapter 4.

[2] Explicitly supplied triggers do not necessarily need to be subterms of the body of the formula. It is however always possible, and usually advisable, to design the axiomatization in a way that triggers indeed are subterms of the body. This possibility is also irrelevant for understanding of the intuition behind the triggers.

Going back to our example, let us assume the trigger for the heap axiom is a single-element set: $\{\mathrm{rd}(\mathrm{wr}(H,p,v),p)\}$, which, following Boogie, we will from now on write as:

$$\forall H,p,v.\ \{\mathrm{rd}(\mathrm{wr}(H,p,v),p)\}\ \mathrm{rd}(\mathrm{wr}(H,p,v),p) = v$$

In the ground monome:

$$\{\mathrm{rd}(H_0,x) < 0,\ H_1 = \mathrm{wr}(H_0,x,-\mathrm{rd}(H_0,x))),\ \neg(\mathrm{rd}(H_1,x) \geq 0)\}$$

the active, non-variable terms are:

$$\mathrm{rd}(H_0,x),\ \mathrm{wr}(H_0,x,-\mathrm{rd}(H_0,x))),\ -\mathrm{rd}(H_0,x),\ \mathrm{rd}(H_0,x),\ \mathrm{rd}(H_1,x)$$

If we take $\sigma = [H := H_0, p := x, v := -\mathrm{rd}(H_0,x)]$, then $\sigma(\mathrm{rd}(\mathrm{wr}(H,p,v),p)) = \mathrm{rd}(H_1,x)$ assuming that $H_1 = \mathrm{wr}(H_0,x,-\mathrm{rd}(H_0,x)))$.

A *multi-trigger* is a trigger with more than one element. As described above, the trigger is said to match is all its terms match. Multi-triggers are useful when there is no single term containing all the variables, for example, in the following formula stating transitivity of relation $P(...)$:

$$\forall x,y,z.\ \{P(x,y),P(y,z)\}\ P(x,y) \wedge P(y,z) \Rightarrow P(x,z)$$

A quantified formula can also have more than one trigger, in such case it is good enough if one of the triggers match. For example, if we want the above transitivity formula, to trigger also when one of the premises and the consequence is active we could say:

$$\begin{aligned}\forall x,y,z.\ &\{P(x,y),P(y,z)\}\ \{P(x,y),P(x,z)\}\ \{P(y,z),P(x,z)\}\\ &P(x,y) \wedge P(y,z) \Rightarrow P(x,z)\end{aligned}$$

Therefore, multiple terms in a trigger are akin to conjunction, while multiple triggers are akin to disjunction.

2.4 DPLL(T)

For simplicity, Section 2.2 above describes the search for boolean satisfying assignment as a separate step, not involving SMT theories. Most modern SMT solvers, however, interleave these two steps, using the DPLL(T) search algorithm. DPLL(T) (Ganzinger et al., 2004) is an extension of the propositional version of the Davis-Putnam-Logemann-Loveland procedure (Davis and Putnam, 1960; Davis et al., 1962). The procedure operates on the *assignment stack* S, which is a sequence of literals. The propositional version of the procedure, checking satisfiability of the formula ψ, consists of three main steps.

1. *Boolean constraint propagation* (BCP). In this step, the assignment stack is extended with literals propositionally implied by the conjunction of the input formula ψ and the literals on the assignment stack. To be precise, a literal l is added to S, iff $l \notin S$ and $(\bigwedge_{l' \in S} l') \wedge \psi \Rightarrow l$.

2. *Decision*, where an arbitrary literal l, such that (1) $l \notin S$ and $\neg l \notin S$ and (2) l or $\neg l$ occur in ψ, is pushed on the assignment stack. Such decision is the followed by another BCP phase, followed by another decision, and so on, until a conflict is reached.

3. *Conflict resolution*, where l results from BCP, but $\neg l$ is already found on the decision stack. Then the reasons for l (and possibly $\neg l$) are analysed and a conflict clause is created using propositional resolution. The conflict clause is $\neg l_1 \vee ... \vee \neg l_n$, such that $l_i \in S$ and $\psi \wedge l_1 \wedge ... \wedge l_n \Rightarrow l \wedge \neg l$. Thus, the conflict clause follows from the input formula. Such a conflict clause will have an effect on later BCP phases, where whenever $l_1, ..., l_m, l_{m+2}, ..., l_n$ are pushed on the assignment stack, $\neg l_{m+1}$ will be pushed, thus preventing this particular conflict on l. After conflict resolution, some literals from the assignment stack are popped, until the conflict clause can be satisfied. If the conflict clause cannot be satisfied, even after popping all decision literals from the stack, the problem is unsatisfiable.

The SMT version of this procedure allows theories to participate in BCP (i.e., theories can say that the literal $x > 7$ is implied by $x > y$ and $y > 10$, even though there might be no propositional connection between them in ψ). Additionally, the theories may perform some more expensive checks only from time to time. In general, they can signal a conflict at any time during the search. Such conflict is resolved very similarly to the propositional conflicts.

The non-ground extension of this procedure adds instantiation tautologies to ψ.

2.5 Modular Verification and Function Calls

Deductive verification systems very often use procedure-modular reasoning, that is each procedure is verified separately. This is due to limited capabilities of the theorem prover: one can hardly expect it to gracefully handle a VC for the entire program at once. The basic idea behind procedure-modular reasoning is to use function contracts, in form of pre- and postconditions, to desugar calls. As an example, let us consider the following C function:

```
int add_two(int x)
  requires(0 <= x && x < 100)
  ensures(2 <= result && result < 102)
{
  return x + 2;
}
```

The function `add_two(...)` requires its parameter to fit the [0..99] range and ensures that the result returned from it will be in the [2..101] range. The function `use_case(...)` then tries to establish that `add_two(y)` will be less than `7` provided that `y` was less than `5`:

```
void use_case(int y)
  requires(0 <= y && y < 5)
{
  int z;
  z = add_two(y);
  assert(z < 7);
}
```

This fails because the body of `use_case(...)` is verified only with respect to the specification of `add_two(...)` and not its body. This is done by asserting the precondition, assigning arbitrary values to locations modified by the function (this assignment is often referred to as the *havoc* operation), and then assuming the postcondition. The call-desugaring of `use_case(...)` is:

```
void use_case()
{
  int y, z;
  // havoc the parameter
  y = *;
  // assuming precondition of use_case
  assume(0 <= y && y < 5);
  // asserting precondition of called function
  assert(0 <= y && y < 100);
  // call "happens": havoc the local variable,
  // where the result of the call is written
  z = *;
  // assuming postcondition of add_two
  assume(2 <= z && z <= 102);
  // user-supplied assertion
  assert(z < 7);
}
```

The specification can thus hide details of implementation.

Chapter 3

Programming with Triggers

This chapter focuses on usage patterns of SMT solvers in software verification scenarios. In particular, we present a case study of applying VCC (Cohen et al., 2009a)[1] program verifier, powered by the Z3 (de Moura and Bjørner, 2008) SMT solver, in a large operating system verification project (more in Section 3.1.2). VCC is a deductive verifier for concurrent C programs. Therefore, following the model outlined in Section 2.1, VCC takes annotated C functions as input, and turns them, with the help of Boogie (Barnett et al., 2006), into verification conditions (VCs). Validity of a VC implies (partial, as we do not check for termination) correctness of a program. Therefore, if a model for a negation of a VC can be constructed, it points to a possible problem in the function, while unsatisfiability of negation of a VC implies correctness of the verified function.

3.1 E-matching for Theory Building

Each verification tool depends on a *verification methodology*, dictating the specification language and commonly used specification idioms, as well as the particular modelling of the programming language semantics to be used. Therefore, from a SMT point of view, verification conditions should be evaluated modulo a theory describing the verification methodology. Clearly, no SMT solver supports such arbitrary theory out of the box. Moreover, given the complexity of such a theory and the pace at which it tends to evolve during development of a verification tool, it seems highly impractical to implement such theory inside of an SMT solver. This is why usually (Detlefs et al., 1998; Flanagan et al., 2002; Barnett et al., 2005; Lahiri and Qadeer, 2008; Filliâtre, 2003) in deductive verification a first-order axiomatization is developed, using theories available in the SMT solver (like uninterpreted functions, integer arithmetic, bit-vector arithmetic, and arrays). The formulas presented as axioms to the SMT solver should be understood as theorems in the model of the

[1] VCC, including SMT-support tools described later in the chapter, is available for academic research, with source code, at `http://vcc.codeplex.com/`.

programming language semantics and verification methodology.

Such an axiomatization consists mostly of universally quantified formulas. The E-matching procedure (Section 2.3), controlled by triggers, is what SMT solvers usually (Detlefs et al. 2005; de Moura and Bjørner 2008; Ge et al. 2007; and also Fx7, as described in Chapter 4) use to deal with quantified formulas.

Quantifier instantiation with triggers is often viewed, especially in the SMT community, as an unreliable heuristic, with no completeness guarantees, developed for a legacy system (the SMT solver Simplify; Detlefs et al. 2005) for solving first-order problems. On the other hand, the deductive verification community is generally not concerned with general first-order problems, and instead wants a way of encoding the semantics of the verified programming language. The views of trigger/axiomatization engineering vary from "we need even more control" to "let us pick some triggers and hope the magical SMT solver will get it right". This chapter strongly supports the former camp: with unrestricted quantifier instantiation verification problems very quickly become intractable for Z3, and the experience with Z3's superposition calculi was similar.

3.1.1 Related Work and Contributions

With the exception of Spec#'s treatment of comprehensions (Leino and Monahan, 2009), there has been not much publications about particulars of triggering. On the other hand several tools, including ESC/Modula-3 (Detlefs et al., 1998), ESC/Java (Flanagan et al., 2002), Spec# (Barnett et al., 2005), Havoc (Lahiri and Qadeer, 2008), and Why (Filliâtre, 2003) use these kinds of patterns. Only the encoding of ESC/Java's logic is described in some more detail (Saxe and Leino, 1999). Overall it seems that there is not enough knowledge exchange between the SMT and deductive verification communities regarding these topics. We hope that this chapter will partially bridge that gap, and help develop alternatives to E-matching, by clarifying its present usage patterns.

We view the first-order logic, together with trigger annotations, as a logic programming language used to encode the semantics of the code being verified. This operational view is illustrated by a number of encoding patterns:

- frame clauses (Section 3.2.2) being source of a large fraction of quantified formulas in a typical VC
- versioning (Section 3.2.3), demonstrating the automatic trigger selection employed in Simplify and Z3 to be too restrictive
- stratified triggering (Section 3.2.4), showing the opposite situation, with novel existential activation used to be again more liberal
- and finally rather surprising behavior of a set theory axiom (Section 3.2.5).

We also describe typical use cases of SMT in verification (Section 3.3), including the particular timing and output requirements placed on the SMT solver.

The encoding patterns presented here are the most complex among ones we have used in VCC. We thus postulate them to be benchmark problems for a possible E-matching alternative.

Several axiomatization patterns we present are heavily influenced by the Spec# program verifier, due to similarities in treatment of ownership and framing. We note in the text when this is the case.

3.1.2 Background: The Hypervisor Verification and VCC

The Hypervisor verification project[2] aims at full functional verification of the kernel of Hyper-V, an industrial virtualization platform, currently shipped with Microsoft Windows Server 2008. It is essentially a small operating system, with memory management, a scheduler, and essential device drivers. It consists of about 100 000 lines of C code (excluding comments) and about 5 000 lines of assembly.

The ultimate goal of the project is a formal proof that Hyper-V simulates the virtualized hardware for each of the guest operating systems. There are however multiple intermediate goals, the first one being verification of memory safety in concurrent context. Even this first step relies on establishing, e.g., functional correctness of red-black trees and complex concurrency synchronization protocols.

The goal of the project is to verify the code that is shipped, not to change it just to facilitate verification. This requires handling C in its full "glory", a restriction to a "safe subset" is out of question. Moreover, the entire code-base should be verified, including concurrency control primitives (e.g., spin locks), which are usually taken for granted by verification methodologies. Finally, annotations are supposed to be maintained by the regular development team once the verification is complete. Since an average programmer is usually not an expert in interactive theorem proving, automatic methods should be used as much as practically possible. The project involves up to 20 people working, mostly on specification of the Hyper-V, for three years, making it one of the largest formal verification efforts ever attempted.

These conditions make for a fairly good case-study for verification in the "real world".

VCC (Cohen et al., 2009a) is a tool used for Hypervisor verification. It was developed with the needs of Hypervisor verification project in mind, but given the scope of that project we expect it to be usable on wide spectrum of C programs. In particular, the verification methodology (Cohen et al., 2009b), seems applicable to a wide class of various concurrent algorithms.

[2]It is part of the Verisoft XT verification project, supported by BMBF under grant 01IS07008. The Verisoft's Aviation subproject, focusing on PikeOS embedded operating system verification (Baumann et al., 2009) also uses VCC.

VCC extends the C language with contracts in style of JML (Leavens et al., 1999) and Spec# (Barnett et al., 2005). Functions are equipped with pre- and post-conditions while types (structures and unions) are equipped with two-state invariants, which describe valid states and possible changes of objects of those types. Contracts are specified in a variant of the C programming language consisting of side-effect free expressions, first-order quantification, and lambda expressions.

The annotated C programs are translated, with help of the Boogie verification condition generator (Barnett et al., 2006) to formulas understood by the Z3 (de Moura and Bjørner, 2008) SMT solver. Even though Boogie has multiple theorem prover back-ends (not even restricted to first-order logic, e.g., there exists a back-end (Böhme et al., 2009) for Isabelle/HOL), VCC currently focuses on the SMT back-end and Z3 in particular.

Verification in VCC is function- and thread-modular: each function is verified separately, as if executed by a single thread, where actions of other threads are simulated at certain points.

First Order Manifesto Verification of complex, functional properties of programs has been, to date, mostly done using interactive, higher-order provers. To leverage automation offered by modern SMT solvers, VCC restricts the specification language not to use any higher order or specialized logics. The specifications are expressed using first-order predicates, possibly operating on *ghost state*, i.e., fields and objects introduced only for the purpose of specification. Ghost fields are used, e.g., to store a map-abstraction representing all nodes of a red-black tree in the tree object, or to capture concurrent protocols.

We have been able to specify and verify multiple recursive data structures, as found in the Hyper-V code, some complex synchronization primitives (spin locks, reader-writer locks, rundowns, custom algorithms for message passing) and specify a good deal of data structure invariants. We currently do not face expressiveness problems with the first-order specification language.

Annotation Language Flexibility To facilitate the specification of complex functional properties, VCC supports manipulation of ghost data types, including maps (from pointers and integers into arbitrary types) as well as entire states of execution, which can be captured and used to evaluate expressions in them. Additionally, new user-defined ghost data types can be specified at the level of C, using function symbols and axioms.

Foremost, however, VCC supports explicit triggers in quantified formulas. We found this ability invaluable in specification of recursive data structures (Section 3.2.4), and helpful in a number of other situations. We intend to survey common triggering styles in specifications, toward the end of the project, to see if and how the trigger selection can be mechanized. Currently, however, we focus on a handful of "specification idioms", which are "recipes" describing how to specify a particular implementation artifact, including triggers.

3.2 Encoding Patterns

This section presents a few common patterns for encoding of verification methodology with a help of E-matching.

While triggers give precise restrictions when an instance can be generated, the exact time at which the instance *will* be generated is determined heuristically. Experience with Z3, VCC and Spec# suggests eager instantiation (i.e., the instance is generated as soon as the relevant terms appear in the monome, before any case-splitting) to be the most efficient.

3.2.1 The Simple: Tuples and Inverse Functions

This simple example shows how triggering can make the behavior of an SMT solver rather unpredictable. Let us consider a typical axiomatization of a pair constructor and selector functions:

$$\forall x, y.\ \{\text{pair}(x, y)\}\ \text{fst}(\text{pair}(x, y)) = x \wedge\ \text{snd}(\text{pair}(x, y)) = y$$

In other words, whenever the term $\text{pair}(t, s)$ becomes active, the axiom will also activate (and give value to) the selector functions. Thus, if an assumption like $\text{pair}(0, a) = \text{pair}(1, a)$ is present, the axiom will, through congruence closure, cause $0 = 1$ to be assumed. On the other hand, should we select $\{\text{fst}(\text{pair}(x, y))\}$ as the trigger, which would be natural if we thought of the axiom being the definition for the fst(...) function, an assumption like the above alone would not trigger the axiom. Only if the terms $\text{fst}(\text{pair}(0, a))$ and $\text{fst}(\text{pair}(1, a))$ would happen to be active, possibly because of some other proof obligations, would the axiom trigger and cause inconsistency to be detected. This would generally cause unpredictable behavior of the SMT solver: a proof of a particular assertion could be dependent on some unrelated previous proofs. Therefore, the author of the axiomatization needs to identify the cases where the existence of an "interface" function like fst(...) is also used to derive some properties of the objects it is applied to, in particular distinguishing between different instances of such objects.

Extensible Records

Consider the tuple example again, but one where we do not define the constructor function (or the definition axiom) at all. Instead, whenever we need to construct a tuple object, let us say $\langle 1, 2\rangle$, we would introduce a new constant c, and assume $\text{fst}(c) = 1 \wedge\ \text{snd}(c) = 2$. Since there is no mention of the constructor function, new fields can be added freely, assuming the cardinality of the type of c is big enough. For example, VCC background axiomatization defines several selector function on program states, including one for memory values ($\text{state}_{\text{mem}}$) and one for status (ownership etc., state_{st}). We then define helper functions to access different "dimensions" of state. Finally, we subdivide the information about ownership of a particular pointer even further,

using $\text{status}_{\text{closed}}$ and $\text{status}_{\text{owner}}$ selector functions:

$$\begin{array}{lcl} \text{memory}(S,p) & \equiv & \text{rd}(\text{state}_{\text{mem}}(S),p) \\ \text{status}(S,p) & \equiv & \text{rd}(\text{state}_{\text{st}}(S),p) \\ \text{owner}(S,p) & \equiv & \text{status}_{\text{owner}}(\text{status}(S,p)) \\ \text{closed}(S,p) & \equiv & \text{status}_{\text{closed}}(\text{status}(S,p)) \end{array}$$

The reason for such a two-stage encoding is performance. For example, ordinary memory write putting value v at pointer p is going to turn a state S_0 into S_1, where only $\text{state}_{\text{mem}}$ is updated, while state_{st} stays unchanged:

$$\text{state}_{\text{mem}}(S_1) = \text{wr}(\text{state}_{\text{mem}}(S_0),p,v) \wedge \text{state}_{\text{st}}(S_1) = \text{state}_{\text{st}}(S_0)$$

Subsequent reads from $\text{state}_{\text{st}}(S_1)$ do not need to go through any quantifier instantiation to be transformed into reads on $\text{state}_{\text{st}}(S_0)$. On the other hand, the ownership-related information tends to be updated all at once, and therefore there is no reason for separation of heaps. For example, closing an object p and setting its owner to o is done with the following assumption[3]:

$$\begin{array}{l} \text{state}_{\text{mem}}(S_1) = \text{state}_{\text{mem}}(S_0) \wedge \\ \quad (\exists s.\ \text{state}_{\text{st}}(S_1) = \text{wr}(\text{state}_{\text{st}}(S_0),p,s)) \\ \quad \wedge\ \text{owner}(S_1,p) = o \wedge\ \text{closed}(S_1,p) \wedge ... \end{array}$$

We postulate existence of a status of p such that the owner of o is p and p is closed. Alternatively, instead of the existential quantifier, one could say that the new state after update is what it is:

$$\begin{array}{l} \text{state}_{\text{mem}}(S_1) = \text{state}_{\text{mem}}(S_0) \\ \quad \wedge\ (\text{state}_{\text{st}}(S_1) = \text{wr}(\text{state}_{\text{st}}(S_0),p,\text{status}(S_1,p))) \\ \quad \wedge\ \text{owner}(S_1,p) = o \wedge\ \text{closed}(S_1,p) \wedge ... \end{array}$$

which might be trickier to understand, but is otherwise very similar. Another example is pointers to ghost state, which we can draw freely from the set of integers, and thus they can encode arbitrary amounts of information. In particular, pointers to certain objects encode versions of ownership domains.

3.2.2 The Common: Framing in the Heap

Basically any reasoning in deductive verification builds on top of heap updates and accesses. This suggests the heap encoding to be crucial for performance. In fact the time of reasoning about the heap is dominant in VCC problems. This section gives an overview of the heap encoding, as used in VCC and Spec#, with some references to other systems. The VCC heap[4] is axiomatized

[3] The ownership information also includes time stamps, reference counts, and so on, which tend to be updated all at once, even if closedness and ownership do not.

[4] The memory model designed for VCC (Cohen et al., 2009c) imposes a typed object model on top of C flat memory. Thus, the heap axiomatization is very similar to the one used for type-safe languages.

using standard select-of-store axioms:

$$\forall H, p, v.\ \mathrm{rd}(\mathrm{wr}(H, p, v), p) = v$$
$$\forall H, p, q, v.\ p \neq q \Rightarrow\ \mathrm{rd}(\mathrm{wr}(H, p, v), q) = \mathrm{rd}(H, q)$$

The function wr(...) is used when a single heap location is updated. On the other hand, upon procedure call several locations, let us say a and b, need to be updated. This is expressed by introducing a fresh variable H_1 and connecting it with the current heap, say H_0, using a *frame clause*, like:

$$\forall q.\ \mathrm{rd}(H_0, q) = \mathrm{rd}(H_1, q) \vee q = a \vee q = b$$

Spec# and VCC use *ownership* to organize objects in the heap, in particular with respect to framing. Each object has a distinguished field which stores the reference to the current owner of the object. The *ownership domain* of an object o is the set of objects from which o can be reached by following zero or more ownership links. If a procedure is allowed to write p, it can also write everything in the ownership domain of p. Since the reachability relation, used in the definition of the ownership domain, is not expressible in first-order logic, we over-approximate the set of written locations to include all objects not directly owned by the current thread (denoted me). Therefore, a frame clause for a procedure writing a and b in VCC would be:

$$\forall q, f.\, H_0[q, f] = H_1[q, f] \vee q = a \vee q = b \vee H_0[q, \mathrm{owner}] \neq \mathrm{me}$$

where $H[p, f] \equiv \mathrm{rd}(H, \mathrm{field}(p, f))$[5], and the function $\mathrm{field}(p, f)$ gives the address of a field f in the object pointed to by p. Consequentially, for almost every $H[p, f]$ access we will see, the term $H[p, \mathrm{owner}]$ being generated. Depending on the methodology, there might be more such artifacts, which together contribute a fair amount of complexity to heap reasoning.

Chaining VCC uses backward chaining on frame clauses, i.e., they trigger on $H_1[q, f]$. Any heap access at H_k will be back-propagated to H_{k-1}, H_{k-2} and so on. Alternatively triggering on $H_0[q, f]$ would lead to forward chaining: accesses at the beginning of the function will be propagated toward the end. VCC requires backward chaining. For example let us consider a simplified version, of a verification condition, saying that writing 7 to a field cnt of some object preserves the invariant that all cnt fields are positive. Let $I(H) \equiv (\forall q.\, H[q, \mathrm{cnt}] > 0)$.

$$(I(H_0) \wedge (\forall q, f.\, H_0[q, f] = H_1[q, f] \vee q = a) \wedge H_1[a, \mathrm{cnt}] = 7) \Rightarrow I(H_1)$$

To prove validity of that formula, the SMT solver will skolemize the universal quantifier from $I(H_1)$, generating an assumption $\neg(H_1[q_0, \mathrm{cnt}] > 0)$. $I(H_0)$ will only be applied on q_0, when the term $H_0[q_0, \mathrm{cnt}]$ is activated, which cannot happen, if the frame clause triggers only on $H_0[q, f]$.

[5] We use the symbol $\equiv$ to define a syntactic shortcut, which is expanded before the SMT solver sees it. This has triggering behavior different from introducing a function symbol and defining equivalence through an axiom. Boogie allows for easy switching between those two styles of function definitions on per-function basis.

Multiple Heaps Some tools use multiple logical constants to encode the heap. For example in ESC/Java the split is done per-field (Saxe and Leino, 1999), while in Frama-C (Moy, 2009), the heap is further split based on syntactic aliasing analysis. This is clearly beneficial for the SMT solver, as no reasoning is necessary to infer that updates on different heaps commute. However, in case of VCC or Spec#, the benefits would be minimized because the frame clause of a procedure potentially needs to simulate write effects on all the partial heaps, as one does not know where objects from ownership domains might be stored.

The Good Heap

The verification methodology usually involves some protocols on accessing the heap. For example, one might model heap locations holding machine integers as mathematical, unbounded integers, but make sure a value outside the appropriate machine integer range is never stored in such a location. However, an axiom like:

$$\forall H, p.\ 0 \leq \mathrm{rd}(H, p) \leq 2^{32} - 1$$

introduces an inconsistency, as in principle one could instantiate it with $[H := \mathrm{wr}(H_0, q, 2^{32}), p := q]$. However, the point is that we are never going to create a heap like that. Moreover, because of triggering, the SMT solver is never going to instantiate the axiom with such a heap, which makes the unsoundness of such an axiom hard to detect. On the other hand, we do not want to rely on triggering for soundness, and therefore heaps obeying the verification methodology protocols are distinguished from other heaps with help of a predicate, let us call it good_heap(H). It is assumed for every "approved" heap, i.e., one introduced by the verification tool, and used as a precondition in axioms like the range axiom above. We never supply a definition for such a predicate, only axiomatize its consequences.

One can use several layers of such predicates (and e.g., Spec# and VCC do), depending on which of the system invariants hold. For example, after a heap update we know about integer ranges, but we do not know that invariants of all objects are preserved, before we check the invariant of the object just being updated.

3.2.3 The Liberal: Versioning

This section gives an example where the usual automatic trigger selection heuristics (used in Simplify, Z3 and CVC3) are too restrictive, and we need to introduce explicit triggers to make it more liberal.

In VCC, in a particular heap, an object can be either *open* or *closed*. In closed objects, only fields marked with the volatile modifier can change. The set of objects owned by an object is stored in a non-volatile field[6], and

[6] This can be overridden by an explicit annotation, but for brevity we skip that possibility.

consequently the set of object in an ownership domain, as well as their non-volatile fields, cannot change, as long as the root object is closed. Therefore, if an object o is closed in each of a sequence of consecutive heaps, we can infer that a value of some field in the domain of o is the same in all of them. To avoid quantifying over sequences of states we use versioning: each object is equipped with a field carrying the version, and when the object is closed, this field is assigned a value encoding the values of all elements of the ownership domain. We do not specify the encoding explicitly, just axiomatize some of its properties.

Let $\text{domain}(H, r) \equiv \text{ver\_domain}(H[r, \text{version}])$[7]. There are no additional axioms attached to the ver_domain(...) function. Its mere existence guarantees that the domain is part of the version encoding (cf. Section 3.2.1). The following axiom states that the encoding of the version also includes all non-volatile fields of objects in the domain:

$$\begin{aligned}&\forall H, p, q, f.\ \{p \in \text{domain}(H, q), \text{rd}(H, \text{field}(p, f))\}\\ &\quad \neg\,\text{volatile}(f) \wedge p \in \text{domain}(H, q) \Rightarrow\\ &\qquad \text{rd}(H, \text{field}(p, f)) = \text{fetch}(H[q, \text{version}], \text{field}(p, f))\end{aligned}$$

Any value read from an object p known to reside in domain of q is considered to be a function of version of q (fetch(...) is similar to ver_domain(...) in that sense), and hence if the version of q does not change, the value also does not change[8]. The explicit triggers on the axiom above, will cause it to be applied whenever a field of an object, known to reside in a domain is accessed. However, should we allow Simplify or Z3 to automatically choose the trigger, it would go for $\text{fetch}(\text{rd}(H, q), \text{field}(p, f))$, as this is the only single-term trigger possible. This is however fairly useless since the only way to activate applications of fetch(...) is to apply this very axiom. This is thus an example where automatic trigger selection has not only performance implications, but also makes the axiom outright useless.

Filtering Trigger

The fetch(...) axiom above, with the explicit trigger, will be instantiated for volatile and non-volatile fields, and then the instances for volatile fields will be discarded per the implication precondition. To limit its applicability already at the instantiation level, we introduce a function symbol non_volatile(...), assume it for non-volatile fields and then add $\text{non\_volatile}(f)$ to the trigger. If we use the function symbol $\text{non\_volatile}(f)$ in the bodies of quantified formulas only when it is already placed in the trigger, no new instances of non_volatile(...) will be created. Thus, the formulas will trigger only for non-volatile fields,

[7] VCC stores the version of the object at the address of the object itself (i.e., we have $\text{field}(p, \text{version}) = p$; since pointers in VCC include type information, no real field is actually stored there), so listing an object in the frame clause really means listing its version. This plays well with encoding of frame clauses.

[8] Following Spec# we use a similar trick for frame axioms of pure methods (Darvas and Leino, 2007).

for which we explicitly assumed the predicate. A similar pattern is used to supply different definitions of certain functions for primitive and non-primitive pointers.

If for some reason we want to avoid multi-triggers (for example because Simplify does not handle them very efficiently), one can use "the *as* trick" (Saxe and Leino, 1999): in addition to $P(x) = \text{true}$, we would assume $\text{as\_}P(x) = x$, thus allowing triggering on $\{h(\text{as\_}P(x))\}$ instead of the multi-trigger $\{h(x), P(x)\}$. We have not used it in VCC in this particular place, but Section 3.2.4 uses similar trick in the definition of the $\in'$ predicate.

3.2.4 The Restrictive: Stratified Triggering

The following section demonstrates a case, where the automatic trigger selection will cause too many instantiations, i.e., we will need to restrict triggering.

Consider the following formula, being part of a simplified invariant of a doubly-linked list:

$$\begin{aligned}&\text{inv}(H,l) \;\Leftrightarrow\; ...\\ &\quad\wedge\; (\forall p.\; p \in \text{owns}(H,l) \quad\Rightarrow\quad H[p,\text{next}] \in \text{owns}(H,l) \;\wedge\\ &\qquad\qquad\qquad\qquad\quad H[p,\text{prev}] \in \text{owns}(H,l) \;\wedge\; H[p,\text{data}] \neq \text{null})\end{aligned}$$

The expression $\text{owns}(H,l)$ refers to the set of objects owned by l in the heap H. If the trigger would be $p \in \text{owns}(H,l)$ we would cause a *matching loop*: a term $p_0 \in \text{owns}(H,l)$ will possibly activate the terms $H[p_0,\text{prev}] \in \text{owns}(H,l)$ and $H[p_0,\text{next}] \in \text{owns}(H,l)$, each of which will in turn activate two more terms, and so on. Even if we limit instantiation depth to n, we still get 2^n quantifier instances, and severe restrictions on n are unrealistic (see Section 3.3). Instead we split the formula into two recursive parts, triggering on the consequent of the implication, and a non-recursive part describing properties of a single list node:

$$\begin{aligned}&\text{inv}(H,l) \;\Leftrightarrow\; ...\\ &\quad\wedge\; (\forall p.\; \{H[p,\text{next}] \in \text{owns}(H,l)\}\\ &\qquad\qquad p \in \text{owns}(H,l) \;\Rightarrow\; H[p,\text{next}] \in \text{owns}(H,l))\\ &\quad\wedge\; (\forall p.\; \{H[p,\text{prev}] \in \text{owns}(H,l)\}\\ &\qquad\qquad p \in \text{owns}(H,l) \;\Rightarrow\; H[p,\text{prev}] \in \text{owns}(H,l))\\ &\quad\wedge\; (\forall p.\; \{p \in \text{owns}(H,l)\}\\ &\qquad\qquad p \in \text{owns}(H,l) \;\Rightarrow\; \psi(H,l,p))\end{aligned}$$

where $\psi(H,l,p) \equiv (H[p,\text{data}] \neq \text{null})$. This way we have removed the matching loop, but another problem remains.

For real trees and lists ψ is more complicated, and therefore we want to avoid ψ being instantiated too often. However, terms of the form $p \in \text{owns}(H,l)$, occur commonly and may have nothing to do with lists, e.g., might become active due to instantiation of definition of set operations or frame clauses. To address this problem, we introduce a function as_node(...), along with an axiom: $\forall p.\; \{\text{as\_node}(p)\}\; \text{as\_node}(p) = p$, making it an identity and

define $p \in' S \equiv \text{as\_node}(p) \in S$. So, by wrapping as_node(...) around a term, we are essentially putting a special marker on it that can be later used in triggers. If we replace all occurrences of $\in$ with $\in'$ in the invariant[9], both in triggers and formula bodies, we end up with much more restricted triggering behavior. The formula will trigger only for pointers p for which as_node(p) was activated.

For example, the typical verification condition might look like $\text{inv}(H_0, l) \wedge \Delta(H_0, H_1) \Rightarrow \text{inv}(H_1, l)$, stating that the invariant of l is preserved by the state transition between heaps H_0 and H_1 (there might be intermediate heaps between them, but this is irrelevant here). When proving the last conjunct of the invariant, the SMT solver tests satisfiability of a formula $\text{inv}(H_0, l) \wedge \Delta(H_0, H_1) \wedge \neg(\forall p.\ p \in \text{owns}(H, l) \Rightarrow \psi(H, l, p))$. Assuming the bound variable p to be skolemized into p_0, the solver assumes $p_0 \in' \text{owns}(H_1, l)$ and $\neg\psi(H_1, l, p_0)$. In particular, the term as_node(p_0) is activated. Thus, when the solver infers $p_0 \in \text{owns}(H_0, l)$, then $\psi(H_0, l, p_0)$ follows, hopefully conflicting with $\neg\psi(H_1, l, p_0)$.

We have now limited the instantiations. In cases where the limitations are over-restrictive, e.g., the user needs $\psi(H_0, l, n)$ as a lemma for a specific list node n, then the user needs to introduce the marker as_node(n), usually by adding an explicit assertion of the form $n \in' \text{owns}(H_0, l)$.

Existential Activation In the previous example, it is possible that one needs to look one element forward in the list, to prove that the invariant of an arbitrary element is preserved, e.g., we might need $\psi(H_0, l, H_1[p_0, \text{next}])$ in addition to $\psi(H_0, l, p_0)$. However, since p_0 is a fresh constant, introduced by the SMT solver, the user cannot explicitly assert $H_1[p_0, \text{next}] \in' \text{owns}(H_0, l)$, which would be required to trigger it. Instead, the user can supply an annotation which states what terms should be activated when the formula undergoes skolemization:

$$(\forall p.\ \{p \in \text{owns}(H, l)\}\{\textbf{ex\_act}{:}H[p, \text{next}] \in' \text{owns}(H, l)\} \\ p \in \text{owns}(H, l) \Rightarrow \psi(H, l, p))$$

This pattern was crucial in verification of recursive data structures in VCC.

3.2.5 The Weird: Distributivity, Neutral Elements and Friends

This section talks about rather surprising behavior of a common set theory axiom. Similar axioms, causing similar problems, include pointer arithmetic normalization (`&(&p[i])[j] = &p[i+j]`) and various distributivity axioms (for integer arithmetic, bit-vector arithmetic or set theory). The set theory example we are going to use is the following axiom describing the relation between set union and difference:

$$\forall A, B, C.\ \{(A \setminus B) \setminus C\}\ (A \setminus B) \setminus C = A \setminus (B \cup C)$$

[9] VCC provides a definition of the $\in'$ predicate, so the user can choose to use $\in'$ in invariants, and live with the consequences.

Such an axiom may seem benign, but the number of applications resulting from a ground term $(...((c \setminus d_0) \setminus d_1)... \setminus d_n)$ is exponential with n (we will get all possible parenthezations of the expression $d_0 \cup ... \cup d_n$, and also some of its subexpressions). The number of instances can be reduced to quadratic by introducing another function symbol $\hat{\setminus}$:

$$\forall A, B.\ \{A \setminus B\}\ A \setminus B = A \hat{\setminus} B$$
$$\forall A, B, C.\ \{(A \hat{\setminus} B) \setminus C\}\ (A \hat{\setminus} B) \setminus C = A \hat{\setminus} (B \cup C)$$

Alternatively, we could trigger the original axiom on $A \setminus (B \cup C)$ However, should at some point the term $c \setminus (\emptyset \cup d)$ arise, a matching loop would occur, provided the SMT solver would know $d = \emptyset \cup d$, but not $c = c \setminus \emptyset$, for example, because of axiom instantiation order or a missing axiom. The matching loop would involve instantiations where $B = \emptyset, C = d$ and A is c, $c \setminus \emptyset$, $(c \setminus \emptyset) \setminus \emptyset$ and so on.

The morale here, is that one needs to be careful when suppling such axioms that can be recursively applied, and make sure they do not loop or trigger excessively often. Luckily, such cases can be usually easily spotted when profiling the axiomatization (i.e., examining SMT solver log files containing list of instances produced during solver's run on a particular problem).

3.3 Performance Requirements on the SMT Solver

An important aspect of verification tools that is often overlooked is that they fail most of the time. This is inherent in the process of developing specifications: one tries different version of annotations (and possibly code) until the program finally goes through the verifier. This usually involves running the verifier every minute or so, after small changes or additions in annotations. Thus, usually we have a large number of unsuccessful runs of the verifier, and one successful run at the end. Therefore, in terms of SMT, the time to find a (probable) model is much more important than the time to prove unsatisfiability. This is particularly interesting, as it seems SMT with quantifiers currently lacks good stop conditions, short of waiting for all the matching possibilities to be exhausted. This is more of practical, rather then theoretical, problem because even if we were able to express the axiomatization in a fragment of logic with finite model property the size of formulas involved would likely make the theoretically finite models gigantic.

From the interactive standpoint, it would be ideal to have responsiveness in the range of a regular compiler, i.e., a couple of seconds. Experience shows that response times of over a minute are discouraging (or worse), and response times of over an hour definitely stop the development of annotations. Incrementality could be possibly exploited: the verifier is run several times with only slightly different versions of the VC. VCC currently does it manually: the user specifies which assertion they are interested in proving, and once they have proven them one-by-one, they can run the full verification, possibly on a build server (or multiple build servers).

As for the general scale of problems, the VCC background axiomatization includes about 300 quantified formulas, almost all with explicit triggers, including 50 with multi-triggers. While Hyper-V is structured into layers, most of its types are visible in most of the functions. There are about 300 types with 1500 fields, which after translation yields about 13000 axioms, half of them ground, consuming about 5 megabytes of SMT-format (Ranise and Tinelli, 2006) file. Just the number of triggers involved exposed a couple of problems in Z3 E-matching indices. Since vast majority of Hyper-V functions are small, the background description of types and their invariants dwarfs the size of the translation of the function body itself. On the other hand, the function body is where the complexity lies: verification times vary between few seconds and hours, despite the fact that the background types are the same.

The number of quantifier instances when verifying a function is usually in the range of tens of thousands. Moreover, most of the axioms are never instantiated. However, interactions between the ones that are needed are quite complex. For example, we examined a proof of validity of a simple function inserting an element into a singly-linked list. While the example took less than 10 seconds to verify, the maximal required matching depth, i.e., the depth of the causal DAG for instances needed in the proof, was already 17. This function involved about 10 heap updates (most of which were ghost updates of the map abstracting the list and methodology bookkeeping). Thus, for every heap location accessed at the end of the function, one would need to apply 10 axioms to learn about the value of that location at the beginning of the function. 10 out of 17 instances in the chain were actually application of frame clauses. On the other hand, this chain also involved 6 different user defined formulas, coming from the invariant of the list. We conclude from this that any attempt at putting hard limits at instantiation depth are misguided.

3.4 Debugging and Profiling Axiomatizations

Analogously to ordinary programs, axiomatizations need to be debugged when the verification tool gives invalid answers and profiled when the tool takes too much time or memory. An invalid answer can be either due to unsoundness, when the tool proclaims a buggy program to be correct, or due to an incompleteness in the opposite case.

This section gives some insight on methods and tools implemented in VCC to aid in debugging and profiling axiomatizations.

3.4.1 Soundness

It is possible to develop axiomatization, where each formula presented to the SMT solver as an axiom is actually a theorem, which is valid in a theory modelling the programming language semantics. The theory should be a conservative extension built in a higher order logic proof assistant. In case of VCC there are however some practical reasons because of which such an ef-

fort was not undertaken. The two most important reasons are: a verification methodology in flux, developed alongside the axiomatization and the need to meet performance requirements on the behavior of Z3 with the axiomatization, which required frequent updates and a lot of experimentation. Analogous state of affairs persists in other, similar verification/bug-checking efforts (Detlefs et al., 1998; Barnett et al., 2005; Lahiri and Qadeer, 2008).

The quality assurance in VCC is thus largely based on testing. As with most compilers, the size of the test suite corpus by far exceeds the size of the source of the compiler. The tests are both positive (i.e., benchmarks that are expected to verify) and negative (where a specific error is expected). The problem with negative benchmarks is that they are usually synthetic: simple code snippets constructed to show a specific verification error. The positive benchmarks can be also synthetic, but additionally larger code snippets are usually available showing somewhat more complicated use cases, combining together several features.

To partially compensate for that, VCC uses a reachability analysis (Janota et al., 2007), which checks if the SMT solver can find any unreachable program points (i.e., statements in the program, which the SMT solver can prove will never be executed). An unreachable program point, for the SMT solver, amounts to proving false at that point. Therefore, the analysis essentially reduces to testing several versions of the program, with `assert(false)` statements placed just before joints of the control flow graph. If the SMT solver finds such assertion to be valid, the location is unreachable. Such a result might point to code that is indeed unreachable (for example, stemming from a defensive programming technique), however more often than not it points to an inconsistent function preconditions or an inconsistent background axiomatization. Such a test is definitely not a silver bullet[10], in particular it only detects immediate inconsistencies, not ones requiring non-trivial actions from the user to reproduce, or giving too strong guarantees about the programming language semantics. Still, we found this analysis very valuable during VCC development.

Soundness problems are thus usually discovered through a failing test case. After initial efforts to minimize such test case, if the cause of unsoundness is still unknown, it is useful to know a small subset of axioms that are needed for the problem to manifest. Such a subset can be extracted from a full proof, if one can be produced by the SMT solver, but also from the UNSAT core[11]. Additionally, if neither of those options is available, one can just run smaller and smaller subsets of axioms through the SMT solver, using some automated procedure. On the other hand, we have not found the exact proof to be very useful, mainly due to its size and complexity.

[10] In Boogie it is referred to as the *smoke test*, from turning a device on and seeing if the smoke goes out of it.

[11] If we treat an SMT problem as a big conjunction, then the UNSAT core is a (hopefully small) subset of the conjuncts, which are unsatisfiable.

3.4.2 Completeness

When analyzing a completeness problem (i.e., one where the verifier returns spurious errors), we try to pinpoint the cause of a failure by minimizing the test case, and possibly adding explicit assertions. If a problem is not apparent, we turn to a model in which the negation of a VC is satisfiable. The general technique is to evaluate the failed assertion in the model and try to trace the reasons why it fails. When doing so, we usually arrive at a point where our conceptual model of what should happen disagrees with the Z3 model. If the annotations are correct, the disagreement is because of a missing axiom or a wrong trigger. Such problems need to be solved by both the verification tool author, but also by the users, in case they use complex specifications.

The Model Viewers

As the models can get quite large, VCC includes two tools for inspecting them. The VCC specific model viewer offers a debugger-like view of the counterexample, interlinked with the source code. One can trace pointers and inspect values of fields of data structures in different states. It thus hides a lot of axiomatization detail and is meant for the user of the verification tool as an aid in debugging their specifications.

The other tool is generic and can be used with different axiomatizations, mainly by the authors of the axiomatization or the SMT solver. It displays all active terms from the model and allows for inspecting equalities between them. For every term one can also see its immediate sub- and super-terms. For example, when looking at term $\mathrm{rd}(H_7, p)$, one can see it is currently equal to 17, go to its subterm p, and then look at all other terms using p, e.g., $\mathrm{rd}(H_6, p)$, $\mathrm{rd}(H_8, p)$ and $\mathrm{wr}(H_8, p, 3)$, revealing value of pointer p in different states.

The two encoding patterns described below do not alter the logical value of the VC, and not even (significantly) the search tree of the SMT solver, but are used only to make the reason for failure more explicit in the model.

Instantiation Traces in Models

To make sure a quantified formula was triggered, one can use *marker functions*. First, we axiomatize the marker function to be always true:

$$\forall x, y.\ \{mark_1(x, y)\}\ mark_1(x, y)$$

Then, if we want to check if the following formula triggers:

$$\forall x, y.\ \psi(x, y)$$

we change it into the following (leaving trigger intact and preserving it truth value):

$$\forall x, y.\ \psi(x, y) \wedge mark_1(x, y)$$

This forces the model to include $mark_1(t, s)$ for every t, s, for which the quantified formula is instantiated. Note that $\forall x, y.\ \psi(x, y)$ might be used positively and negatively, for example as part of a type invariant, which is why we need to axiomatize the marker function to be always true.

Such marker functions can be used by the author of a verifier, but are also available for the end-user if the language allows for specification function definitions as it is the case for VCC.

Assignment Traces in Models

When looking at the model, it is often useful to inspect values of local variables at different points. As described in Section 2.2, each source level assignments to a variable introduces, during VC generation, a new *incarnation* of that variable. To help keep track of different incarnations through the tool chain (e.g., to protect from copy-propagation), after the assignment we introduce an assumption, using the uninterpreted function symbol local_is(...), for example the following source code:

```
void foo()
{
  int x;
  x = 10;
  x = x + 1;
  assert(x < 11);
}
```

will be transformed to:

```
void foo()
{
  assume(x1 == 10);
  assume(local_is(the_x, 4, 2, x1));
  assume(x2 == x1 + 1);
  assume(local_is(the_x, 5, 2, x2));
  assert(x2 < 11);
}
```

The parameters to local_is(...) are a unique symbolic constant, unused elsewhere, and thus immune from copy-propagation, the line and column where the assignment took place and the current value of the variable (i.e., a reference to the current incarnation). The model for this program will include $\text{local\_is}(\text{the\_x}, 4, 2, 10) = \mathit{true}$ and $\text{local\_is}(\text{the\_x}, 5, 2, 11) = \mathit{true}$, revealing the value of `x` at different program points, and allowing the user to spot why the assertion fails. We use a similar technique to tie different heaps to source locations.

3.4.3 Performance Problems

Usually there is a tension between expressiveness and ease of annotation on one side and performance on the other side. Generally, proper triggering

allows for mitigating some such tensions. A feature, bringing new expressive power, usually consists of new function symbols and axioms. If the axioms are triggered only by the new function symbols, parts of the code not using the feature do not suffer performance problems. Using a build server infrastructure to track the time of executions of different parts of the test suite is a good way of making sure that introduction of a new feature does not interfere with existing use cases.

Performance problems tend to manifest themselves in larger benchmarks, especially ones coming from users of the verification tool. An SMT solver usually allows one to gather some simple statistical data at the end of its run. In the case of Z3 such statistics include the number of quantifier instances, conflicts, as well as various statistics from the arithmetic module (which is usually the only non-core theory of interest for VCC). Most of the performance problems we had were due to excessive quantifier instantiation, but some were due to inefficiencies in Z3, particularly in the arithmetic module. The statistics help to distinguish between those two cases: for example low number of instances per second (in the case of Z3 and VCC less than about 10 000) points to a problem different than quantifier instantiation.

The Causal DAG

In case of quantifier problems, one can instruct Z3 to save to a log file the data concerning all quantifier instances made during the search. For example, given the formula:

$$\psi \equiv (\forall x.\ \{f(g(x))\}\ f(g(x)) \Rightarrow h(x,x))$$

if in the current model the terms $f(d)$ and $g(c)$ are active and $d = g(c)$, then Z3 creates an instantiation tautology $\psi \Rightarrow \psi_c$, where:

$$\psi_c \equiv f(g(c)) \Rightarrow h(c,c)$$

After that, the following tuple is added to the log file:

$$L_c \equiv \langle \psi, [x := c], \{f(d), g(c)\}, \{h(c,c)\} \rangle$$

The tuple lists the reference to the quantified formula, the substitution, the active terms that caused the match, and the outcome of the instantiation (i.e., terms that were created in the course of instantiation; the particular log entry above assumes this is the first time in the current search branch where $h(c,c)$ was needed).

Later, in the same search branch, if an instantiation tautology $\phi \Rightarrow \phi_t$ is generated, where its log entry is:

$$L_t \equiv \langle \phi, ..., \{..., h(c,c), ...\}, ... \rangle$$

we call L_c a *cause* of L_t. The directed graph formed by instances with the causal relation (where the edge goes from L_c to L_t) is acyclic (instances can

only cause other instances later in time). We refer to it as the *causal DAG* (directed acyclic graph). This information might be imprecise (e.g., the instance where a term first appeared does not necessarily have to be the one that caused the term to be activated; the matching algorithm will not log terms from proofs of required equivalence class merges and so on). Still, we have found it invaluable in tracing performance problems.

By the nature of performance problems, the number of instances involved tends to be rather large (up to millions). We have thus implemented a tool for analyzing instantiation log files, called the axiom profiler. It allows one to trace a particular instance through the causal DAG, while also inspecting various summaries. One summary is the cost of an instance, which is the estimated number of instances caused by it. More precisely, the cost of an instance node n in the causal DAG E (where $(n, m) \in E$ means n causes m) is given by $c(n)$:

$$c(n) = 1 + \sum_{(n,m)\in E} \frac{c(m)}{indeg(m)}$$

where $indeg(n) = |\{m \mid (m, n) \in E\}|$, is the number of incoming edges of n. In other words, the cost of an instance is shared equally by all instances that caused it. The other summary is the depth of the instance (which is also computed by Z3 during runtime), which is a maximal sum of weights of nodes on the path from a DAG root to the instance. The weight is user-defined and defaults to 1. We view weights as a promising future direction in restricting solver search space, but do not have much experience with them yet.

The Conflict Tree

Another piece of information that the Z3 log file provides is the *conflict tree*. Consider the assignment stack in the SMT solver (Section 2.4). First a number of literals is pushed, and a conflict is found. We can think of this chain of literals as one long branch of a tree. Then backtracking occurs, taking us somewhere up the chain, and, starting from there, new sequence of assignments happens, creating another branch. Backtracking always ends just below decision literals, thus by analysing the sequence of literal assignments, decisions and backtrack operations, we can reconstruct a tree, where internal nodes are labelled by literals, the tree is branching at decisions, and the leaves are labelled by conflicts.

Since information about literal assignments and quantifier instantiation is interleaved in the log file, the costly places in the search (ones where lots of quantifier instances are generated) can be identified. This information can be later attributed to conflicts, i.e., one can see how many quantifier instances and literal assignments it took to discover a particular conflict. Therefore, one can pick a costly conflict at random and try to see if it "makes sense", i.e., if one expects such conflict to contribute to the final outcome and if it really should be that hard to find it. Similarly, one can find an instantiation at random and see if it "makes sense". This is the most effective way of finding performance

problems we have found. Clearly, this requires an intimate understanding of the inner workings of axiomatization, as well as some understanding of the SMT solver search algorithms, which is why we view the axiom profiler as a tool for the author of a verification tool, and not for the end-users.

Z3 Inspector

In Section 3.3 we have argued how important it is for the SMT solver to return the "satisfiable" answer fast. However, the ideal time constraints on such responses (in range of few seconds) are very hard to meet with the current combination of hardware speed, SMT solver performance and VC complexity. As a partial remedy, Boogie offers its users a way of inspecting the progress of Z3 on the current VC. Unlike in other implementations (Filliâtre, 2003) this is achieved without splitting the VC into subformulas before sending them one-by-one to the SMT solver. Instead, we instruct Z3 to simulate a failure at pre-defined time intervals, and use the standard error-reporting mechanism. This standard mechanism relies on "labels" (Leino et al., 2005), which describe the part of formula, which is satisfied by the current monome, and thus also the assertion, negation of which is satisfiable in the model of the monome. The labels also encode the execution path that lead to the failing assertion.

Boogie, which is normally responsible for translation of labels to error messages, captures continuous label output of Z3 and passes it, along with error translation information, to the Z3 Inspector tool. The Inspector displays the source code of the program, from which the VC was generated, where each line is annotated with possible errors that could be reported for this line, should one of the obligations in the VC fail to prove. As the SMT solver works through the formula, the error message corresponding to the current monome is highlighted.

This works particularly well for the case split strategy in Z3, which tries to satisfy the formula left-to-right. This means Z3 will try to prove assertions top-down and, which is more important, will only move to the next assertion, after the previous one is proven. Translated to Z3 Inspector terms the assertion will blink, one by one, for certain amount of time. If one assertion blinks for a long time, it means the prover is trying too hard to refine a model for the case of this assertion failing. The user of the verification tool can then decide, based on time it took to prove this or similar assertion before, to consider this particular assertion to be faulty and kill the verification process.

The cumulative number of samples, pointing to the particular error message, is also displayed, allowing for analysis of which proof obligation was particularly costly, also after the search is done. One could imagine the conflict clauses being communicated out of Z3, so Z3 Inspector could mark the errors that are already guaranteed not to happen (i.e., we know that there is no model that would satisfy the negation of their assertions). This might be useful with a more random case split strategies.

In general the Z3 Inspector provides progress information about the verification process, which is accessible and meaningful for the end-user. It was

helpful in reducing user frustration with the verification process by providing some basic control over it.

3.5 Conclusion

We have presented some of the triggering patterns used in software verification, as well as development practices for axiomatizations, in hope of establishing the requirements for a possible alternative to E-matching based quantifier instantiation methods. Following the presented operational view, where triggering is a way of programming the SMT solver, we envision such an alternative to admit high degree of control, e.g., a domain specific language based on term rewriting systems. More automatic techniques, e.g., superposition, could be useful in subproblems involving formulas from the program specification, as such formulas are less likely to contain useful guidance to the SMT solver, than the ones from the background axiomatization.

Given some help from the SMT solver side, some of the presented encoding patterns might be expressed more directly, allowing the SMT solver to treat them more efficiently. For example, the "*as* trick", of assuming $f(c) = c$, just to be able to use $f(x)$ in a trigger, is a method for controlling when a certain axiom should be applied, and the function $f(...)$ does not have much logical meaning, just polluting the data-structures of an SMT solver. If a native SMT construct for labelling terms and axioms would be provided, one would not need to do so. Clearly each such feature would need to have its possible performance benefits weighted against complications in the SMT solver implementation and input language.

Benchmarks SMT benchmarks exercising techniques described in this chapter are available in the UFNIA (and AUFLIA with older benchmarks) division of the SMT LIB. Current VCC benchmarks will soon be submitted to the UFNIA division, pending resolution of some known soundness issues.

Chapter 4

E-matching

Chapter 3 discusses various techniques for setting up efficient axiomatizations which make proper use of triggers. This chapter presents two novel, efficient algorithms (Section 4.3 and 4.4) for E-matching of such triggers. These algorithms are compared to a well-known algorithm (Section 4.2) described in literature.

4.1 Definitions

Let $\mathcal{V}$ be the infinite, enumerable set of variables. Let Σ be a set of function and constant symbols. Let $\mathcal{T}$ be a set of first-order terms constructed over Σ and $\mathcal{V}$.

A *substitution* is a function $\sigma : \mathcal{V} \to \mathcal{T}$ that is not the identity only for a finite subset of $\mathcal{V}$. We identify a substitution with its homomorphic extension to all terms (i.e., $\sigma : \mathcal{T} \to \mathcal{T}$). Let $\mathcal{S}$ be the set of all substitutions.

We will use letters x and y, possibly with indices, for variables, f and g for function symbols, c and d for constant symbols (functions of arity zero), σ and ρ for substitutions, t for ground terms, and p for possibly non-ground terms. We will use the notation $[x_1 := t_1, \ldots, x_n := t_n]$ for substitutions, and $\sigma[x := t]$ for a substitution augmented to return t for x.

An instance of an *E-matching problem*[1] consists of a finite set of *active* ground terms $\mathcal{A} \subseteq \mathcal{T}$, a relation $\cong_g \subseteq \mathcal{A} \times \mathcal{A}$, and a finite set of non-variable, non-ground terms $p_1, \ldots, p_n$, which we call the *triggers*. Let $\cong \subseteq \mathcal{T} \times \mathcal{T}$ be the smallest congruence relation over Σ containing $\cong_g$. Let $\texttt{root} : \mathcal{T} \to \mathcal{T}$ be

[1] In the automated reasoning literature, the term E-matching usually refers to a slightly different problem, where $\mathcal{A}$ is a singleton and $\cong$ is not restricted to be finitely generated (Plotkin, 1972). On the other hand, in SMT context, the Simplify technical report (Detlefs et al., 2005) as well as the recent Z3 paper (de Moura and Bjørner, 2008) use the term E-matching in the sense defined above.

a function[2] such that:

$$(\forall t, s \in \mathcal{T}.\, \texttt{root}(t) = \texttt{root}(s) \Leftrightarrow t \cong s) \wedge (\forall t \in \mathcal{T}.\, \texttt{root}(t) \cong t)$$

The solution to the E-matching problem is the set:

$$T = \left\{ \sigma \;\middle|\; \begin{array}{l} \exists t_1, \ldots, t_n \in \mathcal{A}.\, \sigma(p_1) \cong t_1 \wedge \cdots \wedge \sigma(p_n) \cong t_n, \\ \forall x \in \mathcal{V}.\, \sigma(x) = \texttt{root}(\sigma(x)) \end{array} \right\}$$

4.1.1 NP Hardness of E-Matching

The problem of deciding for a fixed $\mathcal{A}$ and $\cong_g$, and a given trigger, if $T \neq \emptyset$, is NP-hard (Kozen, 1977). We give a simple independent proof because it gives intuitions for why this problem is difficult.

We show the reduction of a three-coloring problem instance to an E-matching problem instance. We are given a graph (V, E). Let

$$\begin{array}{l} V = \{v_1, \ldots, v_n\} \\ E = \{(v_{i_1}, v_{j_1}), \ldots, (v_{i_m}, v_{j_m})\} \end{array}$$

We construct a trigger:

$$f(g(x_{i_1}, x_{j_1}), f(g(x_{i_2}, x_{j_2}), \ldots f(g(x_{i_m}, x_{j_m}), d) \ldots))$$

and match it against $\mathcal{A} = \{d, f(c, d), g(c_i, c_j)\}$ for $i, j \in \{1, 2, 3\}$, and $\cong$ being the least congruence containing $d = f(c, d)$ and $c = g(c_i, c_j)$ for $i \neq j$ and $i, j \in \{1, 2, 3\}$. We observe that the terms of the form:

$$f(g(c_{i_1}, c_{j_1}), f(g(c_{i_2}, c_{j_2}), ..., f(g(c_{i_m}, c_{j_m}), c)...))$$

are available for matching for $i_k \neq j_k$. These terms represent all colorings of nodes which do not assign the same color to two connected nodes. When we match such a term against the trigger, all occurrences of a node will need to have the same color (i.e., only one of the constants c_1, c_2, or c_3 will be assigned to the variable representing the node).

This encoding depends on congruence giving rise to exponentially many terms available for matching. In this particular case, the amount of terms is even infinite, but this is not crucial, i.e., we could restrict the congruence graph not to have cycles, and still get an NP-hard problem.

NP-hardness is the reason why each solution to the problem is inherently backtracking in nature. In practice, though, the triggers are small, and the problem is not the complexity of a backtracking search for a particular trigger, but rather the fact that in a given proof search there are often hundreds of thousands of matching problems to solve.

```
fun simplify_match([p_1, ..., p_n])
  R := ∅
  proc match(σ, j)
    if j = nil then R := R ∪ {σ}
    else case hd(j) of
      (c, t) ⇒      /* 1 */
        if c ≅ t then match(σ, tl(j))
        else skip
      (x, t) ⇒      /* 2 */
        if σ(x) = x then match(σ[x := root(t)], tl(j))
        else if σ(x) = root(t) then match(σ, tl(j))
        else skip
      (f(p_1, ..., p_n), t) ⇒      /* 3 */
        foreach f(t_1, ..., t_n) in A do
          if t = * ∨ root(f(t_1, ..., t_n)) = t then
            match(σ, (p_1, root(t_1)) :: ··· :: (p_n, root(t_n)) :: tl(j))
  match([], [(p_1, *), ..., (p_n, *)])     /* 4 */
  return R
```

Figure 4.1: Simplify's matching algorithm

4.2 Simplify's Matching Algorithm

Simplify is a legacy SMT system, the first one to efficiently combine theory and quantifier reasoning. This combination made it a popular target for various software verification systems. The Simplify technical report (Detlefs et al., 2005) describes a recursive matching algorithm *simplify_match* given in Figure 4.1. The symbol :: denotes a list constructor, *nil* is an empty list, $[x_1, \ldots, x_n]$ is a shorthand for $x_1 :: \cdots :: x_n :: nil$, and [] is an empty (identity) substitution. *hd* and *tl* are the functions returning, respectively, head and tail of a list (i.e., $hd(x :: y) = x$ and $tl(x :: y) = y$). The command **skip** is a no-op.

The *simplify_match* algorithm maintains the current substitution and a stack (implemented as a list) of pairs of a trigger and a ground term to be matched. We refer to these pairs as *jobs*. Additionally, it uses the special variable $*$ in place of a ground term to say that we are not interested in matching against any specific term, as any active term will do.

We start (line marked /* 4 */) by putting the set of triggers to be matched on the stack and then proceed by taking the top element of the stack.

If the trigger in the top element is a constant (/* 1 */), we just compare it against the ground term, and if the comparison succeeded, continue with the remaining jobs be a recursive call.

[2]Such a function exists by virtue of $\cong$ being an equivalence relation, and is provided by the typical data structure used to represent $\cong$, namely the E-graph (see Simplify technical report (Detlefs et al., 2005) for details on E-graph).

If the trigger is a variable x ($/* \, 2 \, */$), we check if the current substitution already assigns some value to that variable, and if so, we just compare it against the ground term t. Otherwise, we extend the current substitution by mapping x to t and recurse. Observe that t cannot be $*$ since we do not allow a trigger to be a single variable, and $*$ is only paired with triggers in the initial call, never with subtriggers.

If the trigger is a complex term $f(p_1, \ldots, p_n)$ ($/* \, 3 \, */$), we iterate over all the terms with f in the head (possibly checking if they are equivalent to the ground term we are supposed to match against), construct the set of jobs matching respective children of the trigger against respective children of the ground term, and recurse.

The important invariants of *simplify_match* are:

1. the jobs lists contain stars instead of ground terms only for non-variable, non-constant triggers
2. all the ground terms t in job lists satisfy $\texttt{root}(t) = t$
3. for all x either $\sigma(x) = x$ or $\sigma(x) = \texttt{root}(\sigma(x))$

The detailed discussion of this procedure is given in the Simplify technical report (Detlefs et al., 2005).

4.3 Subtrigger Matcher

This section describes a novel matching algorithm, optimized for linear triggers. A *linear trigger* is a trigger in which each variable occurs at most once. Most triggers used in the program verification problems we have inspected are linear. The linearity means that matching problems for subterms of a trigger are independent, which allows for more efficient processing.

However, even if triggers are not linear, it pays off to treat them as linear, and only after the matching algorithm is complete discard the resulting substitutions that assign different terms to the same variable. This technique is often used in term indexes (Ramakrishnan et al., 2001) used in automated reasoning. The algorithm, therefore, does not require the trigger to be linear.

This matcher algorithm is given in Figure 4.2. It uses operations $\sqcap$ and $\sqcup$, which are defined on sets of substitutions:

$$
\begin{aligned}
A \sqcap B &= \{\sigma \oplus \rho \mid \sigma \in A, \rho \in B, \sigma \oplus \rho \neq \bot\} \\
A \sqcup B &= A \cup B \\
\sigma \oplus \rho &= \begin{cases} \bot & \text{when } \exists x.\, \sigma(x) \neq x \wedge \rho(x) \neq x \wedge \sigma(x) \neq \rho(x) \\ \sigma \cdot \rho & \text{otherwise} \end{cases} \\
\sigma \cdot \rho(x) &= \begin{cases} \sigma(x) & \text{when } \sigma(x) \neq x \\ \rho(x) & \text{otherwise} \end{cases}
\end{aligned}
$$

$\sqcap$ returns a set of all possible non-conflicting combinations of substitutions from two sets. $\sqcup$ sums two such sets. The next section shows an implementation of these operations that does not use explicit sets.

```
fun fetch(S, t, p)
  if S = ⊤ then return {[p := root(t)]}
  else if S = × ∧ t ≅ p then return {[]}
  else if S = × then return ∅
  else return S(root(t))

fun match(p)
  case p of
    x ⇒ return ⊤
    c ⇒ return ×
    f(p_1, ..., p_n) ⇒
      foreach i in 1...n do S_i := match(p_i)        /* 1 */
      if ∃i. S_i = ⊥ then return ⊥                   /* 2 */
      if ∀i. S_i = × then return ×                   /* 3 */
      S := {t ↦ ∅ | t ∈ A}
      foreach f(t_1, ..., t_n) in A do               /* 4 */
        t := root(f(t_1, ..., t_n))
        S := S[t ↦ S(t) ⊔ (fetch(S_1, t_1, p_1) ⊓ ··· ⊓ fetch(S_n, t_n, p_n))]
      if ∀t. S(t) = ⊥ then return ⊥
      else return S

fun topmatch(p)                                      /* 5 */
  S := match(p)
  return ⨆_{t∈A} S(t)

fun subtrigger_match([p_1, ..., p_n])
  return topmatch(p_1) ⊓ ··· ⊓ topmatch(p_n)
```

Figure 4.2: Subtrigger matching algorithm

The *match(p)* function returns the set of all substitutions σ, such that $\sigma(p) \cong t$, for a term $t \in \mathcal{A}$, categorized by $\texttt{root}(t)$. More specifically, *match* returns a map from $\texttt{root}(t)$ to such substitutions, or one of the special symbols $\top$, $\bot$, $\times$. Symbol $\top$ means that p was a variable x, and therefore the map is: $\{t \mapsto \{[x := t]\} \mid t \in \mathcal{A},\ \texttt{root}(t) = t\}$, symbol $\bot$ represents no matches (i.e., $\{t \mapsto \emptyset \mid t \in \mathcal{A}, \texttt{root}(t) = t\}$), and $\times$ means p was ground, so the map is $\{\texttt{root}(p) \mapsto \{[]\}\} \cup \{t \mapsto \emptyset \mid t \in \mathcal{A}, t = \texttt{root}(t), t \neq p\}$[3].

The only non-trivial control flow case in the *match* function is the case of a complex trigger $f(p_1, \ldots, p_n)$, which works as follows:

- /* 1 */ recurse on subtriggers. Conceptually, we consider the subtriggers to be independent of each other (i.e., $f(p_1, \ldots, p_n)$ is linear). If they

[3]Here we assume all the ground subterms of triggers to be in $\mathcal{A}$. This is easily achieved and does not affect performance in our tests.

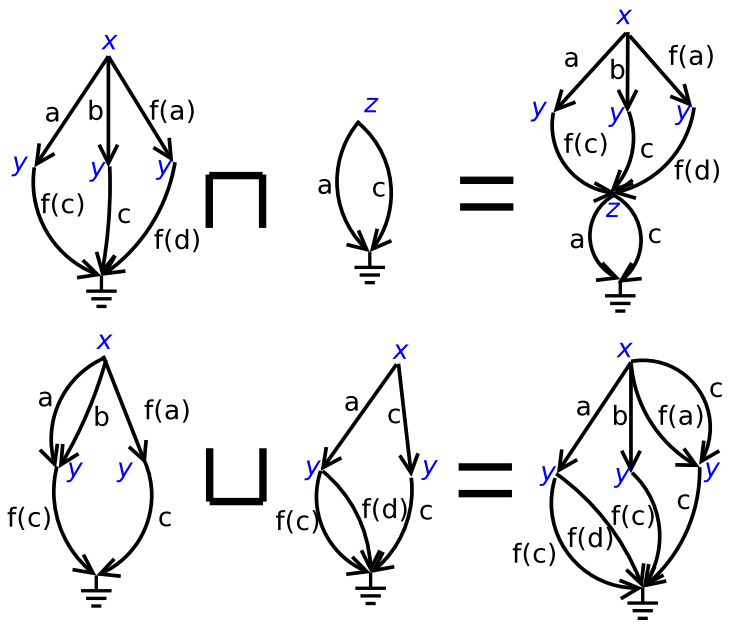

Figure 4.3: Example of s-tree operations

are, however, dependent, then the $\sqcap$ operation filters out conflicting substitutions.

- /* 2 */ check if there is any subtrigger that does not match anything, in which case the entire trigger does not match anything.
- /* 3 */ check if all the children of $f(p_1, \ldots, p_n)$ are ground, in which case $f(p_1, \ldots, p_n)$ is ground as well.
- /* 4 */ otherwise we start with an empty result map S and iterate over all terms with the correct head symbol. For each such term $f(t_1, \ldots, t_n)$, we combine (using $\sqcup$) the already present results for $\texttt{root}(f(t_1, \ldots, t_n))$ with results of matching p_i against t_i. The *fetch* function is used to retrieve results of subtrigger matching by ensuring the special symbols are treated as the maps they represent.

Finally (/* 5 */) the *topmatch* function just collapses the maps into one big set.

4.3.1 S-Trees

This section introduces a new data structure, s-tree, which is used to compactly represent sets of substitutions, so they can be efficiently manipulated during the matching.

The s-trees data structure itself can be viewed as a special case of substitution trees in automated reasoning (Ramakrishnan et al., 2001) with rather severe restrictions on their shape. We, however, do not use the trees as an index and, as a consequence, require a different set of operations on s-trees than those defined on substitution trees.

S-trees require a strict, total order $\prec \subseteq \mathcal{A} \times \mathcal{A}$ and are defined inductively:

1. ϵ is an s-tree
2. if $T_1, \ldots, T_n$ are s-trees and $t_1, \ldots, t_n$ are ground terms, then the pair $x \rhd [(t_1, T_1), \ldots, (t_n, T_n)]$ is an s-tree

The invariant of the s-tree data structure is that in each node the term $t_1, \ldots, t_n$ are sorted according to $\prec$ (i.e. for all i and j, $i < j \Rightarrow t_i \prec t_j$)[4], and that there exists a sequence of variables $x_1 \ldots x_k$ such that the root has the form $x_1 \rhd \ldots$ and each node (including the root) has the form:

1. $x_i \rhd [(t_1, x_{i+1} \rhd \ldots), \ldots, (t_n, x_{i+1} \rhd \ldots)]$ or
2. $x_k \rhd [(t_1, \epsilon), \ldots, (t_n, \epsilon)]$

for some n, $t_1, \ldots, t_n$ and $1 \leq i < k$. In other words, the variables at a given level of a tree are the same.

The *yield* function maps a s-tree into the set of substitutions it is intended to represent.

$$
\begin{aligned}
\mathit{yield}(\epsilon) &= \{[]\} \\
\mathit{yield}(x \rhd [(t_1, T_1), \ldots, (t_n, T_n)]) &= \left\{ \sigma[x := t_i] \;\middle|\; \begin{array}{l} i \in \{1, \ldots, n\}, \\ \sigma \in \mathit{yield}(T_i) \\ \sigma(x) = x \vee \sigma(x) = t_i \end{array} \right\}
\end{aligned}
$$

Example s-trees are given in Figure 4.3. The trees are represented as ordered directed acyclic graphs with aggressive sharing. An s-tree:

$$x \rhd [(t_1, T_1), \ldots, (t_n, T_n)]$$

has the label x on the node, label t_i on the edges and each edge leads to another tree T_i. The ground symbol corresponds to the empty tree ϵ. E.g., the middle bottom one represents

$$x \rhd [(a, y \rhd [(f(c), \epsilon), (f(d), \epsilon)]), (c, y \rhd [(c, \epsilon)])]$$

which yields

$$\{[x := a, y := f(c)], [x := a, y := f(d)], [x := c, y := c]\}$$

The ordering of terms used in the example is:

$$a \prec b \prec c \prec d \prec f(a) \prec f(c) \prec f(d)$$

[4] This invariant is employed in implementation of the $\sqcup$ operator.

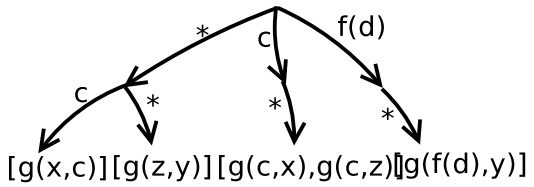

Figure 4.4: Example of an index for flat triggers with g in head

We now define an analogous for s-trees of the operators $\sqcup$ and $\sqcap$ we defined earlier for sets of substitutions. Formally, the operators are defined so that *yield* is a homomorphism from s-trees to sets of substitutions.

$$
\begin{aligned}
\epsilon \sqcap T &= T \\
x \rhd [(t_1, T_1), \ldots, (t_n, T_n)] \sqcap T &= x \rhd [(t_1, T_1 \sqcap T), \ldots, (t_n, T_n \sqcap T)] \\
\epsilon \sqcup \epsilon &= \epsilon \\
x \rhd X \sqcup x \rhd Y &= x \rhd merge(X, Y) \\
merge((t, T) :: X, (t', T') :: Y) &= \begin{cases} (t, T \sqcup T') :: merge(X, Y) & \text{if } t = t' \\ (t, T) :: merge(X, (t', T') :: Y) & \text{if } t < t' \\ (t', T') :: merge((t, T) :: X, Y) & \text{if } t' < t \end{cases} \\
merge(nil, X) &= X \\
merge(X, nil) &= X
\end{aligned}
$$

The $\sqcap$ corresponds to stacking trees one on top of another, while $\sqcup$ does a recursive merge. Example applications are given in Figure 4.3.

The precondition of the $\sqcup$ operator is that the operands have the same shape, meaning the $x_1 \ldots x_k$ sequence from the invariant is the same for both trees; otherwise, $\sqcup$ is undefined. This precondition is fulfilled by the subtrigger matcher, since it only combines trees resulting from matching of the same trigger, which means the variables are always accessed in the same order.

To change *subtrigger_match* to use s-trees, we need to change the *fetch* function, to return $p \rhd [(\texttt{root}(t), \epsilon)]$ instead of $[p := \texttt{root}(t)]$, ϵ instead of $\{[]\}$ and $x \rhd nil$ for some variable x instead of $\emptyset$. After this is done we only call *yield* at the very end, to transform an s-tree into a set of substitutions.

4.4 Flat Matcher

During performance testing, we found that most triggers shared the head symbol and matching them was taking a considerable amount of time. Moreover, the triggers had a very simple form: $f(x, c)$[5]. This form is a specific example of something we call flat triggers. A *flat trigger* is a trigger in which each variable occurs at most once and at depth one.

[5]The actual function symbol was a subtyping predicate in ESC/Java2's (Kiniry and Cok, 2005) Simplify-based object logic.

Flat triggers with a given head can be matched all at once by constructing a tree that indexes all the triggers with the given function symbol in the head. Such a tree can be viewed as a special kind of a discrimination tree (Ramakrishnan et al., 2001), where we consider each child of the pattern as a constant term, instead of traversing it pre-order. Unlike in discrimination trees used for matching our index has non-ground terms and queries are ground.

We assume, without loss of generality, each function symbol to have only one arity. A node in the index tree is either a set of triggers $\{p_1, \ldots, p_n\}$, or a set of pairs $\{(t_1, I_1), \ldots, (t_n, I_n)\}$, where each of the t_i is a ground term or a special variable $*$, and I_i are index trees.

We call $(t_1, \ldots, t_n, p)$ a *path* in I if and only if either:

1. $n = 0$ and $p \in I$; or
2. $(t_1, I') \in I$ and $(t_2, \ldots, t_n, p)$ is a path in I'

Let $star(x) = *$ for a variable x and $star(t) = t$, for any non-variable term t. We say that I indexes a set of triggers Q if for any $f(p_1, \ldots, p_n) \in Q$ there exists a path $(star(p_1), \ldots, star(p_n), f(p_1, \ldots, p_n))$ in I, and for every path there exists a corresponding trigger.

Given an index I, we find all the triggers that match the term $f(t_1, \ldots, t_n)$ by calling $match'(f(t_1, \ldots, t_n), [t_1, \ldots, t_n], \{I\})$, where $match'$ is defined as follows:

$$
\begin{aligned}
&match'(t, [t_1, \ldots, t_n], A) = \\
&\quad match'(t, [t_2, \ldots, t_n], \{I' \mid I \in A, (p, I') \in I,\ p \cong t_1 \vee p = *\}) \\
&match'(f(t_1, \ldots, t_n), nil, A) = \\
&\quad \{f(p_1, \ldots, p_n) \mapsto \sqcap_{i=1\ldots n,\, p_i \in \mathcal{V}} [p_i := \texttt{root}(t_i)] \ \mid\ I \in A, f(p_1, \ldots, p_n) \in I\}
\end{aligned}
$$

The algorithm works by maintaining the set of trigger indices A containing triggers that still possibly match t. At the bottom of the tree we extract the children of t corresponding to variables in the trigger. We only consider position at which the trigger has variables, not ground terms (the condition $p_i \in \mathcal{V}$).

A flat-aware matcher is implemented by replacing the *topmatch* function from Figure 4.2 with the one from Figure 4.5. The point of using it, though, is to cache I_f and S_p across calls to *subtrigger_match*.

4.5 Implementation and Experiments

We have implemented all three algorithms inside the Fx7 SMT solver[6]. Fx7 is implemented in the Nemerle (Moskal et al., 2007) language and runs on the .NET platform. In each case the implementation is highly optimized and only unsatisfactory results with the *simplify_match* algorithm led to designing and implementing the second and the third algorithm.

[6] Available online at `http://nemerle.org/fx7/`.

fun $topmatch(p')$
  if p is flat **then**
    let $f(p_1, ..., p_n) = p$
    $I_f :=$ index for all triggers with head f
    foreach p **in** I_f **do** $S_p := \emptyset$
    foreach $f(t_1, \ldots, t_n)$ **in** $\mathcal{A}$ **do**
      foreach $p \mapsto T$ **in** $match'(f(t_1, \ldots, t_n), [t_1, \ldots, t_n], I_f)$ **do**
        $S_p := S_p \sqcup T$
    return $S_{p'}$
  else
    $S := match(p')$
    return $\bigsqcup_{t \in \mathcal{A}} S(t)$

Figure 4.5: Flat matcher

The implementation makes heavy use of memoization. Both terms and s-trees use aggressive (maximal) sharing. The implementations of $\sqcap$ and $\sqcup$ exploit this sharing, by memoizing results to avoid processing the same (shared) subtree more than once.

An important point to consider in the design of matching algorithms is incrementality. The prover will typically match, assert some facts, and then match again. The prover is then interested only in receiving the new results. The Simplify technical report (Detlefs et al., 2005) cites two optimizations to deal with incrementality. We have implemented one of them, the *mod-time* optimization, in all three algorithms. The effects are mixed, mainly since our usage patterns of the matching algorithm are different than those of Simplify: we generally change the E-graph more between matchings due to our proof search strategy.

To achieve incrementality we memoize s-trees returned on a given proof path and then use the subtraction operation on s-trees to remove substitutions that had been returned previously. The subtraction on s-trees corresponds to set subtraction and its implementation is very similar to the one of $\sqcup$.

Another fine point is that the loop over all active terms in the implementations of all three algorithms skips some terms: if we have inspected $f(t_1, \ldots, t_n)$ then we skip $f(t'_1, \ldots, t'_n)$ given that $t_i \cong t'_i$ for $i = 1 \ldots n$. Following work on fast, proof-producing congruence closure (Nieuwenhuis and Oliveras, 2005), we encode all the terms using only constants and a single binary function symbol $\cdot(\ldots)$. E.g., $f(t_1, \ldots, t_n)$ is represented by $\cdot(f, \cdot(t_1, \cdots \cdot (t_{n-1}, t_n)))$. Therefore the loop over active terms is skipped when $\texttt{root}(\cdot(t_1, \cdots \cdot (t_{n-1}, t_n)))$ was already visited.

Yet another issue is that we map all the variables to one special symbol during the matching, do not store the variable names in s-trees, and only introduce the names when iterating the trees to get the final results (inside the

yield function). This allows for more sharing of subtriggers between different triggers and is fairly common practice in term indexing.

The tests were performed on a 1 GHz Pentium III computer with 512 MiB of RAM running Linux and Nemerle r7446 on top of Mono 1.2.3. The memory used was always under 200 MiB. We ran the prover on a randomly selected set of verification queries generated by the ESC/Java (Flanagan et al., 2002) and Spec# (Barnett et al., 2005) tools. The benchmarks are now available as part of SMT-LIB (Ranise and Tinelli, 2006).

The subtrigger matcher helps speed up matching by around 20% in the Boogie benchmarks and around 50% in the ESC/Java benchmarks. The flat matcher is around 2 times faster than Simplify's matcher in the Boogie benchmarks and around 10 times in the ESC/Java benchmarks. The detailed results are given in the Section 4.7.

Now we give some intuitions behind the results. For example, consider the trigger $f(g_1(x_1), \ldots, g_n(x_n))$. If each of $g_i(x_i)$ returns two matches, except for the last one which does not match anything, the subtrigger matcher exits after $O(n)$ steps, while the Simplify matcher performs $O(2^n)$ steps. Even when $g_n(x_n)$ actually matches something (which is more common), the subtrigger algorithm still performs $O(n)$ steps to construct the s-tree and only performs $O(2^n)$ steps walking that tree. These steps are much cheaper (as the tree is rather small and fits in the CPU cache) than matching the g_is several times, which Simplify's algorithm does. The main point of the subtrigger matcher is therefore not to repeat work for a given (sub)trigger more than once.

The benefits of the flat matcher seem to be mostly CPU cache-related. For example, a typical problem might have one hundred triggers with head f, and one thousand ground terms with the head f. The flat matcher processes a data structure (of size one hundred) one thousand times, while the subtrigger matcher (and also Simplify's matcher) processes a different data structure (of size one thousand) one hundred times. Consequently, given that these data structures occupy a considerable amount of memory, frequently the smaller data structures in the former case fit the cache, while the larger ones in the latter case do not.

4.6 Conclusions and Related Work

We have presented two novel algorithms for E-matching. They are shown to outperform the well-known Simplify E-matching algorithm.

The E-matching problem was first described, along with a solution, in the Simplify technical report (Detlefs et al., 2005). We know several SMT solvers, like Zap (Ball et al., 2005), CVC3 (Barrett and Tinelli, 2007), Verifun (Flanagan et al., 2004), Yices (Dutertre and de Moura, 2006) and Ergo (Conchon et al., 2007) include matching algorithms, though there seem to be no publications describing their algorithms. Specifically, Zap uses a different algorithm that also relies on the fact of triggers being linear and uses a different kind of s-trees. Zap, however, does not do anything special about flat triggers.

In a recent paper (de Moura and Bjørner, 2008) on Z3 SMT solver, a way of compiling patterns into a code tree that is later executed against ground terms is defined. Such a tree is beneficial if there are many triggers that share the top part of triggers. We, on the other hand, exploit sharing in the bottom parts of triggers, and the flat matcher handles the case of simple triggers that share only the head symbol. The Z3 authors also propose an index on the ground terms that is used to speed up matching in an incremental usage pattern. Such an index could perhaps be used also with our approach. Of course, the usefulness of all these techniques largely depends on benchmarks and the particular search strategy employed in an SMT solver.

During the 2007 SMT competition (Barrett et al., 2008) there were four solvers participating in the Arithmetic, Uninterpreted Functions, Linear Integer Arithmetic and Arrays (AUFLIA) division. The AUFLIA division includes software verification problems and is the only one involving heavy use of quantifier reasoning (see Section 1.1.1 for details). Z3 was first and Fx7 was second, with the same number of solved benchmarks but much worse run time. Both solvers used improved matching algorithms, while other participants (CVC3 and Yices) did not, which is some indication of importance of E-matching in this kind of benchmarks.

Some of the problems in the field of term indexing (Ramakrishnan et al., 2001) in saturation-based theorem provers are also related. As mentioned earlier, our work uses ideas similar to substitution trees and discrimination trees. It seems to be the case, however, that the usage patterns in the saturation provers are different than those in SMT solvers. Matching SMT solvers must deal with several orders of magnitude fewer non-ground terms, a similar number of ground terms, but the time constraints are often much tighter. This different set of constraints and goals consequently leads to the construction of different algorithms and data structures.

4.7 Appendix: Detailed Experimental Results

The first column lists the benchmark name, the second, third and fourth columns are the average times spent matching a single trigger during proof search for a given benchmark. The times are given in microseconds. The second column refer to the subtrigger matcher (Figure 4.3), the third one for the subtrigger matcher combined with the flat matcher (Figure 4.4) and the last column refers to the Simplify matcher (Figure 4.1).

Benchmark name	**Subtrig.**	**Subtrig. + Flat**	**Simpl.**
AssignToNonInvariantField.ClientClass..ctor	248	180	520
Assumptions.Sub..ctor	214	166	444
Branching.T.M.T.notnull.System.Int32	237	217	359
Chunker0.Chunker..ctor.System.String.Int32	133	102	250
DefaultLoopInv0.A.M-modifiesOnLoop-noinfer	173	146	293
Immutable.test3.C..ctor	232	177	549
Interval.Interval.Shift.System.Int32	261	216	564
ModifyOther.Test..ctor	246	179	438
PeerFields.Child..ctor.System.Int32	285	209	602
PeerFields.Parent.M	269	234	494
PeerFields.Parent.P	201	174	369
PeerFields.PeerFields.Assign1.PeerFields	176	140	258
PeerFields.PeerFields.M.System.Int32	284	258	470
PeerModifiesClauses.Homeboy.T-level.2	224	204	313
PureReceiverMightBeCommitted.C..ctor	199	156	351
PureReceiverMightBeCommitted.C.N	189	174	363
QuantifierVisibilityInvariant.A..ctor	245	251	524
QuantifierVisibilityInvariant.B..ctor.int	275	242	479
QuantifierVisibilityInvariant.C..ctor.int	280	233	482
Strengthen.MainClass.Main.HARD	190	163	458
Strings.test3.MyStrings.StringCoolness2.bool	184	153	418
Strings.test3.MyStrings.StringCoolness3	162	129	369
Types.T..ctor.D.notnull-orderStrength.1	142	83	428
ValidAndPrevalid.Interval.Foo4	251	194	545
Average	221	183	431

Benchmark name	**Subtrig.**	**Subtrig. + Flat**	**Simpl.**
javafe.CopyLoaded.019	322	262	826
javafe.ast.AmbiguousMethodInvocation.007	112	70	709
javafe.ast.AmbiguousVariableAccess.007	114	77	775
javafe.ast.ArrayRefExpr.007	106	67	726
javafe.ast.CastExpr.007	113	77	744
javafe.ast.ClassLiteral.007	113	70	764
javafe.ast.CondExpr.007	111	77	739
javafe.ast.FieldAccess.009	118	70	725
javafe.ast.InstanceOfExpr.007	112	72	726
javafe.ast.MethodInvocation.009	122	71	800
javafe.ast.NewArrayExpr.009	112	73	750
javafe.ast.NewInstanceExpr.003	540	272	3959
javafe.ast.NewInstanceExpr.008	108	72	709
javafe.ast.ParenExpr.007	113	76	758
javafe.ast.ThisExpr.007	115	75	723
javafe.ast.VariableAccess.008	109	74	714
javafe.parser.Lex.006	218	189	1056
javafe.parser.Lex.018	256	190	874
javafe.parser.Parse.005	275	202	1864
javafe.parser.Parse.006	354	296	2107
javafe.reader.ASTClassFileParser.005	730	373	5498
javafe.reader.ASTClassFileParser.022	690	415	4544
javafe.tc.Env.007	742	517	5980
javafe.tc.EnvForLocals.001	268	190	869
Average	249	164	1581

Chapter 5

Proof Checking

In Chapter 3 we have shown means to develop an SMT theory corresponding to a verification methodology. Integrity of such a theory (Section 3.4.1) can be either ensured formally, possibly with an interactive proof assistant, or informally through testing (which is the dominating practice). There is however another side to the verification tool integrity, namely soundness of the SMT solver used. Fortunately, significant parts of the SMT solver are only heuristic and do not impact the overall soundness. This is true, for example, for the E-matching algorithms developed in Chapter 4: should they miss some substitutions, we get incompleteness; otherwise, any substitution is allowed by the semantics of first-order logic, even if the E-matching algorithm is faulty and produces substitutions not allowed by the triggers.

Still, there are two problems with soundness of SMT solvers. One problem is that there might be a bug in soundness-impacting part of the SMT solver, whose implementation can be largely opaque to others than the developer. The other problem is that we might want to provide the evidence of program being correct to someone else, like in Proof-Carrying Code (Necula, 1997) scenarios.

It is therefore desirable for an SMT solver to produce the proof of the unsatisfiability of formulas. The problem is that in program verification, the queries are rather huge and so are the proofs. For example formulas in the AUFLIA division of the SMT-LIB (cf. Section 1.1.1) contain up to 130 000 distinct subterms, with an average of 8 000. The proofs we have generated are on average five times bigger than the formulas. The most complicated proof we have encountered contains around 40 000 basic resolution steps and around 1 000 000 (sub)terms in size. What is worth noting however, is that state of the art SMT solvers are able to check a vast majority of such queries in under a second. As the general expectation is that proof checking should be faster than proof generation, it becomes clear that we need very efficient means of proof checking.

Contributions

The contributions of this chapter are:

- we introduce a simple, yet expressive term rewrite formalism (in Section 5.2), and show it is strong enough to encode and check proofs of theory tautologies and boolean tautologies (Section 5.3), and also NNF/CNF (negation normal form/conjunctive normal form) conversions with skolemization (Section 5.4),
- we discuss two highly efficient implementations of the proposed rewrite system (Section 5.6). In particular we discuss performance issues (Section 5.6.2) and we describe techniques to help ensure soundness of the rewrite rules (Section 5.6.1).

There are two reasons to use term rewriting as a proof checking vehicle. One is that the term rewriting is a simple formalism, therefore it is relatively easy to reason about the correctness of an implementation of the proof checker. The bulk of soundness reasoning goes at term rewrite rules level, which is much better understood and simpler to reason about than a general purpose (often low level) programming language used to implement a proof checker.

The second reason is memory efficiency, which on modern CPUs is also a key to time efficiency. We encode proof rules as rewrite rules and handle non-local conditions (like uniqueness of Skolem functions) at the meta level, which allows for the rewrite rules to be local. The idea behind the encoding of the proof rules is to take proof terms and rewrite them into the formulas that they prove. This allows for memory held by the proof terms to be immediately reclaimed and reused for the next fragment of the proof tree read from a proof file.

Proof Search in SMT Solvers

As described in Section 2.2, in order to check unsatisfiability of a formula, an SMT solver will usually first transform it into an equisatisfiable CNF formula, while simultaneously performing skolemization. Subsequent proof search alternates between boolean reasoning, theory reasoning, and quantifier instantiation. Both for the boolean part and for deriving the final empty clause we use resolution. Theory reasoning produces conflict clauses, which are tautologies under respective theories, e.g., $\neg(a > 7) \vee a \geq 6$ or $\neg(c = d) \vee \neg(f(c) = 42) \vee \neg(f(d) < 0)$. Quantifier reasoning is based on instantiating universal quantifiers and producing tautologies like $\neg(\forall x.\, f(x) > 0 \rightarrow P(x)) \vee \neg(f(3) > 0) \vee P(3)$. It can be thought of as just another background theory.

To make this search procedure return a proof, we need proofs of: CNF translation, boolean tautologies and theory tautologies. By taking these three together, we should obtain a proof that the formula is unsatisfiable.

5.1 The Idea

We introduce a rather straightforward proof system for boolean resolution and theory conflicts, consisting of a number of rules. Each proof rule r allows one to derive a formula ϕ based on premises $\phi_1, \ldots, \phi_n$. Per each such rule we introduce a function symbol r and a rewrite rule of the form $r(\phi_1, \ldots, \phi_n) \blacktriangleright \phi$. The application of the term representing a proof rule to premises yields the consequence of that proof rule. Therefore, if we represent the entire proof tree as a corresponding term, then the subterms representing premise-less, leaf rules will rewrite to the formulas they prove, which will allow the subterms containing them to rewrite to the formulas they prove and so on. Finally, the entire term representing the proof tree will rewrite to the formula in the root.

A proof of unsatisfiability of formula ψ, is a proof of false, which can use the formula ψ in leaves as a premise. Our proof checking machinery will thus check if a proof term, using ψ in leaves, rewrites to a constant representing false.

The proof of correctness of the initial skolemization is slightly different. There we introduce several satisfiability preserving transformations, name them with function symbols and construct a term encoding in which order and to what part of the formula to apply them. Then we define rewrite rules taking the input formula and transformation as input and giving the transformed formula as output.

Both operations are performed using a bottom-up rewrite strategy (given as $\boldsymbol{nf}(...)$ below). The term language we are using contains the standard first-order terms, the λ-binder allowing for efficient encoding of rules involving quantified formulas and assumptions, and the **let**-binder to exploit the DAG structure of typical proofs.

5.2 Definitions

Let $\mathcal{V}$ be an infinite, enumerable, set of variables. We use x and y (all symbols possibly with indices) as meta-variables ranging over $\mathcal{V}$. Let Σ be an infinite, enumerable set of function symbols, we use meta-variable f ranging over Σ. We define the set of *terms* $\mathcal{T}$, and the set of *patterns* $\mathcal{P} \subseteq \mathcal{T}$ as follows:

$$
\begin{array}{rcl}
\mathcal{T} & ::= & x \mid f(\mathcal{T}_1, \ldots, \mathcal{T}_n) \mid \lambda x.\, \mathcal{T}_1 \mid \mathbf{cons} \cdot (\mathcal{T}_1, \mathcal{T}_2) \mid \mathbf{nil} \cdot () \mid \mathbf{build} \cdot (f, \mathcal{T}_1) \mid \\
 & & \mathbf{apply} \cdot (\mathcal{T}_1, \mathcal{T}_2) \mid \mathbf{fold} \cdot (\mathcal{T}_1) \\
\mathcal{P} & ::= & x \mid f(\mathcal{P}_1, \ldots, \mathcal{P}_n)
\end{array}
$$

where $n \geq 0$. The notion $s \cdot (...)$ stands for a *special form*, which have particular interpretations in the term rewrite system. We will use $t_1 :: t_2$ as a syntactic sugar for $\mathbf{cons} \cdot (t_1, t_2)$, and $\mathbf{nil}$ for $\mathbf{nil} \cdot ()$.

The set of *free variables* of a term, $FV : \mathcal{T} \to \mathcal{P}(\mathcal{V})$, is defined as usual:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(f(t_1,\ldots,t_n)) &= \textstyle\bigcup_{1\le i\le n} FV(t_i) \\ FV(\lambda x.\, t) &= FV(t) \setminus \{x\} \\ FV(s \cdot (t_1,\ldots,t_n)) &= \textstyle\bigcup_{1\le i\le n} FV(t_i) \end{aligned}$$

Note that it is also defined on $\mathcal{P}$, as $\mathcal{P} \subseteq \mathcal{T}$. Let $\mathcal{T}(A, B)$ be a set of terms built from function symbols from the set A and variables from the set $B \subseteq \mathcal{V}$ (i.e. if $t \in \mathcal{T}(A, B)$ then $FV(t) \subseteq B$). A *substitution* is a function $\sigma : \mathcal{V} \to \mathcal{T}$, which we identify with its homomorphic, capture free extension to $\sigma : \mathcal{T} \to \mathcal{T}$.

A *rewrite rule* is a pair (p, t), where $p \in \mathcal{P}$, $t \in \mathcal{T}$ and $FV(t) \subseteq FV(p)$. Let $\mathcal{R}$ be a set of such rewrite rules, such that for distinct $(p, t), (p', t') \in \mathcal{R}$, p and p' do not unify. We define a *normal form* of a term t, with respect to $\mathcal{R}$ as $\boldsymbol{nf}(t)$, with the rules below. Since the function defined below is recursive, it is possible for it not to terminate. If the rules below do not result in a single unique normal form for term t, then we say that $\boldsymbol{nf}(t) = \otimes$. If term has $\otimes$ as subterm, it is itself regarded as equal to $\otimes$. In practice this condition is enforced by limiting running time of the proof checker.

$$\begin{aligned} \boldsymbol{nf}(x) &= x \\ \boldsymbol{nf}(f(t_1,\ldots,t_n)) &= \begin{cases} \boldsymbol{nf}(t\sigma) \\ \quad \text{for } (p,t) \in \mathcal{R} \text{ such that} \\ \quad \exists\sigma.\, p\sigma = f(\boldsymbol{nf}(t_1),\ldots,\boldsymbol{nf}(t_n)) \\ f(\boldsymbol{nf}(t_1),\ldots,\boldsymbol{nf}(t_n)) \\ \quad \text{otherwise} \end{cases} \\ \boldsymbol{nf}(\lambda x.\, t_1) &= \lambda x.\, \boldsymbol{nf}(t_1) \\ \boldsymbol{nf}(\mathbf{apply} \cdot (\lambda x.\, t_1, t_2)) &= \boldsymbol{nf}(t_1[x := t_2]) \\ \boldsymbol{nf}(\mathbf{build} \cdot (f, t_1 :: \cdots :: t_n :: \mathbf{nil})) & \\ &= \boldsymbol{nf}(f(t_1,\ldots,t_n)) \\ \boldsymbol{nf}(\mathbf{build} \cdot (f, t_1)) &= \mathbf{build} \cdot (f, \boldsymbol{nf}(t_1)) \\ &\quad \text{if none of the above apply} \\ \boldsymbol{nf}(s \cdot (t_1,\ldots,t_n)) &= s \cdot (\boldsymbol{nf}(t_1),\ldots,\boldsymbol{nf}(t_n)) \\ &\quad \text{if none of the above apply} \end{aligned}$$

where $t_1[x := t_2]$ denotes a capture free substitution of x with t_2 in t_1[1].

The semantics of the $\mathbf{fold} \cdot (t)$ is not defined above. Its role is to perform theory-specific constant folding on t. Folding is implemented either inside the proof checker or by an external tool called by the proof checker. In this chapter we use integer constant folding (for example $\boldsymbol{nf}(\mathbf{fold} \cdot (\mathtt{add}(20, 22))) = 42$).

The signature used throughout this chapter can be divided in four categories:

1. logical connectives: `false`, `implies`, `and`, `or`, `forall`, `neg`
2. theory specific symbols: `eq`, `add`, `leq`, `minus` and natural number literals $(0, 1, 2, ...)$

[1]The actual implementation uses de Bruijn indices, so the "capture free" part comes at no cost.

$\dfrac{\Gamma \vdash \psi_1 \to \psi_2 \quad \Gamma \vdash \psi_1}{\Gamma \vdash \psi_2}$ (mp)	$\texttt{mp}(\Box(\texttt{implies}(x_1, x_2)), \Box(x_1)) \blacktriangleright \Box(x_2)$
$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash \psi}$ (absurd)	$\texttt{absurd}(\Box(\texttt{false}), x) \blacktriangleright \Box(x)$
$\dfrac{\Gamma \vdash (\psi \to \bot) \to \bot}{\Gamma \vdash \psi}$ (nnpp)	$\texttt{nnpp}(\Box(\texttt{implies}(\texttt{implies}(x, \texttt{false}), \texttt{false}))) \blacktriangleright \Box(x)$
$\dfrac{\Gamma \cup \{\psi_1\} \vdash \psi_2}{\Gamma \vdash \psi_1 \to \psi_2}$ (assume)	$\texttt{assume}(x_1, x_2) \blacktriangleright \texttt{lift\_known}(\texttt{implies}(x_1, \textbf{apply} \cdot (x_2, \Box(x_1))))$ $\texttt{lift\_known}(\texttt{implies}(x_1, \Box(x_2))) \blacktriangleright \Box(\texttt{implies}(x_1, x_2))$
$\dfrac{\psi \in \Gamma}{\Gamma \vdash \psi}$ (assumption)	no rewrite rule associated

Figure 5.1: A complete system for $\to$ and $\bot$. We list the proof rule, followed by the associated rewrite rule(s).

3. technical machinery: $\texttt{lift\_known}, \Box, \texttt{sk}$
4. rule names

5.3 Boolean Deduction

Consider the logical system from Figure 5.1. It is complete for boolean logic with connectives $\to$ and $\bot$. Three of the derivation rules there ((mp), (absurd) and (nnpp)) fit a common scheme:

$$\frac{\Gamma \vdash \Xi_1(\psi_1, \ldots, \psi_m) \ldots \Gamma \vdash \Xi_n(\psi_1, \ldots, \psi_m)}{\Gamma \vdash \Xi(\psi_1, \ldots, \psi_m)}\ (r)$$

where Ξ_i and Ξ are formulas built from the boolean connectives and formula meta-variables $\psi_1, \ldots, \psi_m$, while (r) is the name of the rule. We call such rules *standard rules*. Additional boolean connectives can be handled by adding more standard rules. To encode a standard derivation rule, we use the following rewrite:

$$r(\Box(\Xi_1(x_1, \ldots, x_m)), \ldots, \Box(\Xi_n(x_1, \ldots, x_m)), x_{i_1}, \ldots, x_{i_l}) \blacktriangleright \Box(\Xi(x_1, \ldots, x_m))$$

where x_{i_j} are additional technical arguments, to fulfill the condition that the left-hand side of a rule has to contains all the free variables in the right-hand

$$\frac{}{\Gamma \vdash t = t}\ (\texttt{eq\_refl}) \qquad \texttt{eq\_refl}(x) \blacktriangleright \Box(\texttt{eq}(x, x))$$

$$\frac{\Gamma \vdash t_1 = t_2 \quad \Gamma \vdash t_2 = t_3}{\Gamma \vdash t_1 = t_3}\ (\texttt{eq\_trans})$$

$$\texttt{eq\_trans}(\Box(\texttt{eq}(x_1, x_2)), \Box(\texttt{eq}(x_2, x_3))) \blacktriangleright \Box(\texttt{eq}(x_1, x_3))$$

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_2 = t_1}\ (\texttt{eq\_trans}) \qquad \texttt{eq\_symm}(\Box(\texttt{eq}(x_1, x_2))) \blacktriangleright \Box(\texttt{eq}(x_2, x_1))$$

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash \psi(t_1) \to \psi(t_2)}\ (\texttt{eq\_sub})$$

$$\texttt{eq\_sub}(\Box(\texttt{eq}(x_1, x_2)), y) \blacktriangleright \Box(\texttt{implies}(\mathbf{apply} \cdot (y, x_1), \mathbf{apply} \cdot (y, x_2)))$$

$$\frac{\Gamma \vdash x + x_1 \leq c_1 \quad \Gamma \vdash -x + x_2 \leq c_2}{x_1 + x_2 \leq c_1 + c_2}\ (\texttt{utvpi\_trans})$$

$$\texttt{utvpi\_trans}(\Box(\texttt{leq}(\texttt{add}(x_1, x_2), x_3)), \Box(\texttt{leq}(\texttt{add}(\texttt{minus}(x_1), y_2), y_3))) \blacktriangleright \Box(\texttt{leq}(\texttt{add}(x_2, y_2), \mathbf{fold} \cdot (\texttt{add}(x_3, y_3)))$$

Figure 5.2: The equality rules, and an example of an UTVPI rule.

side and r is a function symbol used to encode this particular rule. Therefore we can model (`mp`), (`absurd`) and (`nnpp`) using the rewrite rules listed in Figure 5.1. The intuition behind the $\Box(...)$ is that if term P rewrites to $\Box(\psi)$, then P represents a proof of ψ.

We are left with the (`assume`)/(`assumption`) pair, which is modeled using lambda expressions. There is no explicit rewrite for (`assumption`) rule. In the rule for $\texttt{assume}(x_1, x_2)$ (Figure 5.1), the term x_2 is expected to be of the form $\lambda y.\, t$, where t is a proof using y in places where (`assumption`) should be used. This is very similar to the encoding of the IMP-I rule in Edinburgh Logical Framework (Harper et al., 1987).

We call *restricted* the terms of the form $\Box(...)$, $\texttt{lift\_known}(...)$ or $\texttt{sk}(...)$ (the last one is used in the next section). We say that $P \in \mathcal{T}$ is a pre-proof (written $\textbf{\textit{preproof}}(P)$), if it does not contain a restricted subterm, or a subterm which is a $s \cdot (...)$ special form. The idea is that the proof shall be constructed only using the predefined proof rules and not using auxiliary technical machinery.

Lemma 1. *For any pair (P, σ), such that $\textbf{\textit{preproof}}(P)$, $\forall x \in \mathcal{V}.\, x\sigma = x \vee \exists \phi.\, x\sigma = \Box(\phi)$ and $\textbf{\textit{nf}}(P\sigma) = \Box(\psi)$, there exists a derivation $\Gamma \vdash \psi$ where $\Gamma = \{\phi \mid x \in \mathcal{V}, x\sigma = \Box(\phi)\}$.*

Proof. The proof is by induction on the size of P. Since $\Box(...)$ is not a subterm of P, the head of P must be either:

1. a variable x, in which case $x\sigma$ is $\Box(\psi)$ and the (`assumption`) rule can be used, since $\psi \in \Gamma$,

2. $P = r(P_1, \ldots, P_n, t_1, \ldots, t_m)$, where a rewrite, obtained from a derivation rule (r), is applicable to:

$$r(\boldsymbol{nf}(P_1\sigma), \ldots, \boldsymbol{nf}(P_n\sigma), \boldsymbol{nf}(t_1), \ldots, \boldsymbol{nf}(t_m))$$

   We use the induction hypothesis on (P_i, σ), where $\boldsymbol{nf}(P_i\sigma) = \Box(\psi_i)$ and build the derivation using the (r) rule.

3. $P = \texttt{assume}(P_1, \psi)$, which rewrites to $\Box(...)$ in two steps, through the `lift_known`$(...)$ (which cannot be used explicitly because $\boldsymbol{preproof}(P)$). $\textbf{apply}\cdot(P_1, \Box(\psi))$ needs to be reduced to $\Box(...)$ for the `lift_known`$(...)$ to be applied, so $\boldsymbol{nf}(P_1) = \lambda x.\, P_2$, for some P_2. Since no rule can result in a rewrite to a lambda term (all of the rewrite rules have a term of the form $f(...)$ as their right hand side), then not only $\boldsymbol{nf}(P_1)$, but also P_1 itself needs to start with a lambda binder. Therefore $P_1 = \lambda x.\, P_3$, for some P_3. In this case we use the induction hypothesis on $(P_3, \sigma[x := \Box(\psi)])$, and then use the (`assume`) rule to construct the implication.

There are no other cases, since no other rewrite rule has $\Box(...)$ as the right-hand side. □

Applying this lemma with $(P, \emptyset)$ gives the theorem.

Theorem 1. *For any P, such that $\boldsymbol{preproof}(P)$ and $\boldsymbol{nf}(P\sigma) = \Box(\psi)$ there exists a derivation $\vdash \psi$.*

Theory Conflicts. Proving theory conflicts clearly depends on the particular theory. Figure 5.2 lists rules for the theory of equality. The encoding is the same as for the standard rules from Figure 5.1. For arithmetic we currently support the UTVPI fragment (Harvey and Stuckey, 1997) of integer linear arithmetic. It consists of inequalities of the form $ax + by \leq c$, where $a, b \in \{-1, 0, 1\}$ and c is an integer. The decision procedure closes set of such inequalities, with respect to a few rules, of the form similar to the one listed in Figure 5.2. Again, the encoding is the same as for ordinary deduction rules.

5.4 Skolemization Calculus

Figure 5.3 lists rules for a skolemization calculus. The $\uplus$ is disjoint set union (i.e. $A \cup B$ if $A \cap B = \emptyset$ and undefined otherwise). The intuition behind $S; Q \vdash \psi \rightsquigarrow \phi$ is that for each model M of $\forall Q.\, \psi$ there exists an extension of M on the symbols from S that satisfies $\forall Q.\, \phi$. We formalize it using second order logic with the following lemma:

Lemma 2. *Let $Q = \{x_1, \ldots, x_n\}$ and $S = \{f_1 \ldots f_n\}$. If $S; Q \vdash \psi \rightsquigarrow \phi$ where $\psi \in \mathcal{T}(\Sigma, Q)$, $\phi \in \mathcal{T}(\Sigma \uplus S, Q)$, then $\models \exists^2 f_1 \ldots \exists^2 f_n.\, \forall x_1, \ldots, x_n.\, \psi \rightarrow \phi$.*

$$\frac{\emptyset; Q \vdash \neg\psi(f(Q)) \rightsquigarrow \phi}{\{f\}; Q \vdash \neg\forall x.\, \psi(x) \rightsquigarrow \phi}\ (\texttt{skol})$$

$$\texttt{sk}(y, \texttt{skol}(f, y_1), \texttt{neg}(\texttt{forall}(x))) \blacktriangleright \texttt{sk}(y, y_1, \texttt{neg}(\textbf{apply} \cdot (x, \textbf{build} \cdot (f, y))))$$

$$\frac{S; Q, x \vdash \psi(x) \rightsquigarrow \phi(x)}{S; Q \vdash \forall x.\, \psi(x) \rightsquigarrow \forall x.\, \phi(x)}\ (\texttt{skip}_\forall)$$

$$\texttt{sk}(y, \texttt{skip}_\forall(y_1), \texttt{forall}(x_1)) \blacktriangleright \texttt{forall}(\lambda x.\, \texttt{sk}(x :: y, y_1, \textbf{apply} \cdot (x_1, x)))$$

$$\frac{}{\emptyset; Q \vdash \psi \rightsquigarrow \psi}\ (\texttt{id}) \qquad \texttt{sk}(y, \texttt{id}, x_1) \blacktriangleright x_1$$

$$\frac{S_1; Q \vdash \psi_1 \rightsquigarrow \phi_1 \quad S_2; Q \vdash \psi_2 \rightsquigarrow \phi_2}{S_1 \uplus S_2; Q \vdash \psi_1 \wedge \psi_2 \rightsquigarrow \phi_1 \wedge \phi_2}\ (\texttt{rec}_\wedge)$$

$$\texttt{sk}(y, \texttt{rec}_\wedge(y_1, y_2), \texttt{and}(x_1, x_2)) \blacktriangleright \texttt{and}(\texttt{sk}(y, y_1, x_1), \texttt{sk}(y, y_2, x_2))$$

$$\frac{S_1; Q \vdash \psi_1 \rightsquigarrow \phi_1 \quad S_2; Q \vdash \psi_2 \rightsquigarrow \phi_2}{S_1 \uplus S_2; Q \vdash \psi_1 \vee \psi_2 \rightsquigarrow \phi_1 \vee \phi_2}\ (\texttt{rec}_\vee)$$

$$\texttt{sk}(y, \texttt{rec}_\vee(y_1, y_2), \texttt{or}(x_1, x_2)) \blacktriangleright \texttt{or}(\texttt{sk}(y, y_1, x_1), \texttt{sk}(y, y_2, x_2))$$

$$\frac{S_1; Q \vdash \neg\psi_1 \rightsquigarrow \phi_1 \quad S_2; Q \vdash \neg\psi_2 \rightsquigarrow \phi_2}{S_1 \uplus S_2; Q \vdash \neg(\psi_1 \vee \psi_2) \rightsquigarrow \phi_1 \wedge \phi_2}\ (\texttt{rec}_{\neg\vee})$$

$$\texttt{sk}(y, \texttt{rec}_{\neg\vee}(y_1, y_2), \texttt{neg}(\texttt{or}(x_1, x_2))) \blacktriangleright \texttt{and}(\texttt{sk}(y, y_1, \texttt{neg}(x_1)), \texttt{sk}(y, y_2, \texttt{neg}(x_2)))$$

$$\frac{S_1; Q \vdash \neg\psi_1 \rightsquigarrow \phi_1 \quad S_2; Q \vdash \neg\psi_2 \rightsquigarrow \phi_2}{S_1 \uplus S_2; Q \vdash \neg(\psi_1 \wedge \psi_2) \rightsquigarrow \phi_1 \vee \phi_2}\ (\texttt{rec}_{\neg\wedge})$$

$$\texttt{sk}(y, \texttt{rec}_{\neg\wedge}(y_1, y_2), \texttt{neg}(\texttt{and}(x_1, x_2))) \blacktriangleright \texttt{or}(\texttt{sk}(y, y_1, \texttt{neg}(x_1)), \texttt{sk}(y, y_2, \texttt{neg}(x_2)))$$

$$\frac{S_1; Q \vdash \psi_1 \rightsquigarrow \phi_1}{S_1; Q \vdash \neg\neg\psi_1 \rightsquigarrow \phi_1}\ (\texttt{rec}_{\neg\neg})$$

$$\texttt{sk}(y, \texttt{rec}_{\neg\neg}(y_1), \texttt{neg}(\texttt{neg}(x_1))) \blacktriangleright \texttt{sk}(y, y_1, x_1)$$

Figure 5.3: The skolemization calculus.

The key point of the proof is that for the rules of the common form:

$$\frac{S_1; Q \vdash \Xi_1(\psi_1, \ldots, \psi_k) \rightsquigarrow \phi_1 \quad \ldots \quad S_m; Q \vdash \Xi_m(\psi_1, \ldots, \psi_k) \rightsquigarrow \phi_m}{S_1 \uplus \ldots \uplus S_m; Q \vdash \Xi(\psi_1, \ldots, \psi_k) \rightsquigarrow \Xi'(\psi_1, \ldots, \psi_k, \phi_1, \ldots, \phi_m)} \ (r)$$

where ψ_j and ϕ_j range over first-order formulas it is enough to show the following:

$$\begin{array}{l}
\forall Q : \mathbf{Type}. \forall S_1 \ldots S_n : \mathbf{Type}. \\
\forall \psi_1 \ldots \psi_k : Q \to \mathbf{Prop}. \\
\forall \phi_1 : S_1 \times Q \to \mathbf{Prop}. \ldots \forall \phi_m : S_m \times Q \to \mathbf{Prop}. \\
\quad \bigwedge_{i=1\ldots m} (\exists f_i : S_i. \forall x : Q. \Xi_i(\psi_i(x), \ldots, \psi_k(x)) \to \phi_i(f_i, x)) \to \\
\quad (\exists f_1 : S_1. \ldots \exists f_m : S_m. \forall x : Q. \Xi(\psi_1(x), \ldots, \psi_k(x)) \to \\
\qquad \Xi'(\psi_m(x), \ldots, \psi_k(x), \phi_1(f_1, x), \ldots, \phi_m(f_m, x)))
\end{array}$$

which is much like the formula from the lemma, except that there is only one Skolem constant f_i per premise and also there is only one free variable in all the formulas, namely x. However these symbols are of arbitrary type, so they can be thought of as representing sequences of symbols.

We prove such formulas for each rule, the reader can find the proof scripts for Coq proof assistant online [2].

Rewrite encoding The common form of a rule is encoded as:

$$\begin{array}{l}
\mathtt{sk}(y, r(y_1, \ldots, y_m), \Xi(x_1, \ldots, x_k)) \blacktriangleright \\
\quad \Xi'(x_1, \ldots, x_k, \mathtt{sk}(y, y_1, \Xi_1(x_1, \ldots, x_k)), \ldots, \mathtt{sk}(y, y_m, \Xi_m(x_1, \ldots, x_k)))
\end{array}$$

The first argument of $\mathtt{sk}(\ldots)$ is the list of universally quantified variables in scope. The second argument is a rule name, along with proofs of premises. The third argument is the formula to be transformed.

The encoding of non-common rules (as well as the common rules used here) is given in the Figure 5.3.

Lemma 3. *If* $\textbf{\textit{preproof}}(P)$, $\textbf{\textit{nf}}(\mathtt{sk}(x_1 :: \cdots :: x_n :: \mathbf{nil}, P, \psi)) = \psi'$, *and for each occurrence of* $\mathtt{skol}(f)$ *as a subterm of* P, *the function symbol* f *does not occur anywhere else in* P *nor in* ψ, *then there exists* S, *such that* $S; x_1, \ldots, x_n \vdash \psi \rightsquigarrow \psi'$.

Proof. By structural induction over P. □

Theorem 2. *If* $\textbf{\textit{preproof}}(P)$, $\textbf{\textit{nf}}(\mathtt{sk}(\mathbf{nil}, P, \psi)) = \psi'$, *and for each occurrence of* $\mathtt{skol}(f)$ *as a subterm of* P, *the function symbol* f *does not occur anywhere else in* P *nor in* ψ, *and* ψ' *is unsatisfiable then* ψ *is unsatisfiable.*

Proof. By Lemmas 2 and 3. □

[2] `http://nemerle.org/fx7/`

5.5 The Checker

The proof checker reads three files: (1) rewrite rules describing the underlying logic; (2) a query in SMT-LIB concrete syntax; and (3) the proof. The concrete syntax for both the rewrite rules and the proof term is similar to the one used in SMT-LIB. The proof term language includes the following commands:

- **let** $x := t_1$: bind the identifier x to the term $\boldsymbol{nf}(t_1)$
- **initial** t_1 t_2: check if $\texttt{skol}(f)$ is used in t_1 only once with each f, that the f symbols do not occur in t_2, compares t_2 against the query read from the SMT-LIB file, and, if everything succeeds, binds $\Box(\boldsymbol{nf}(\texttt{sk}(\textbf{nil}, t_1, t_2)))$ to the special identifier `initial`; this command can be used only once in a given proof
- **final** t_1: checks if $\boldsymbol{nf}(t_1) = \Box(\texttt{false})$, and if so reports success and exits
- **assert_eq** t_1 t_2: checks if $\boldsymbol{nf}(t_1) = \boldsymbol{nf}(t_2)$ (and aborts if this is not the case)
- **assert_ok** t_1 t_2: checks if $\boldsymbol{nf}(t_1) = \Box(\boldsymbol{nf}(t_2))$ (and aborts if this is not the case)
- **print** t_1: prints a string representation of t_1

The last three commands are used to debug the proofs.

The proofs, after initial skolemization, are structured as a sequence of clause derivations, using either resolution, theory conflicts, instantiation or CNF-conversion steps. All these clauses are **let**-bound, until we reach the empty clause. Basically we end up with a proof-tree in natural deduction, deriving the boolean false constant from the initial formula. The tree is encoded as a DAG because **let**-bound clauses can be used more than once.

All those steps are best described through an example[3]. Figure 5.4 lists rules not previously mentioned in this chapter that were used in the proof. The real proof system has more rules. As described in Section 5.6.1, we mechanically check all rules. Our example formula is:

$$P(c) \wedge (c = d) \wedge (\forall x.\, \neg P(x) \vee \neg(\forall y.\, \neg Q(x, y))) \wedge (\forall x.\, \neg Q(d, x))$$

The first step is the initial skolemization:

let $q_1 := \texttt{forall}(\lambda x.\, \texttt{or}(\texttt{neg}(P(x)), \texttt{neg}(\texttt{forall}(\lambda y.\, \texttt{neg}(Q(x, y))))))$
let $q_2 := \texttt{forall}(\lambda x.\, \texttt{neg}(Q(d, x)))$
let $f_{in} := \texttt{and}(P(c), \texttt{and}(\texttt{eq}(c, d), \texttt{and}(q_1, q_2)))$
let $sk := \texttt{rec}_{\wedge}(\texttt{id}, \texttt{rec}_{\wedge}(\texttt{id}, \texttt{rec}_{\wedge}(\texttt{skip}_{\forall}(\texttt{rec}_{\vee}(\texttt{id}, \texttt{skol}(f, \texttt{rec}_{\neg\neg}(\texttt{id})))), \texttt{id})))$
initial sk f_{in}

[3]The proof presented here is not the simplest possible of this very formula. However it follows the steps that our SMT solver does and we expect other SMT solvers to do.

$\dfrac{\Gamma \vdash \psi_1 \wedge \psi_2}{\Gamma \vdash \psi_1}$ $(\texttt{elim}_{\wedge 1})$	$\texttt{elim}_{\wedge 1}(\Box(\texttt{and}(x_1, x_2))) \blacktriangleright$ $\Box(x_1)$
$\dfrac{\Gamma \vdash \psi_1 \wedge \psi_2}{\Gamma \vdash \psi_2}$ $(\texttt{elim}_{\wedge 2})$	$\texttt{elim}_{\wedge 2}(\Box(\texttt{and}(x_1, x_2))) \blacktriangleright$ $\Box(x_2)$
$\dfrac{\Gamma \vdash \psi_1 \vee \psi_2}{\Gamma \vdash \neg\psi_1 \to \psi_2}$ $(\texttt{elim}_{\vee})$	$\texttt{elim}_{\vee}(\Box(\texttt{or}(x_1, x_2))) \blacktriangleright$ $\Box(\texttt{implies}(\texttt{neg}(x_1), x_2))$
$\dfrac{\Gamma \vdash \neg\psi \quad \Gamma \vdash \psi}{\Gamma \vdash \bot}$ $(\texttt{elim}_{\neg})$	$\texttt{elim}_{\neg}(\Box(\texttt{neg}(x_1))), \Box((x_1)) \blacktriangleright$ $\Box(\texttt{false})$
$\dfrac{\Gamma \vdash \psi}{\Gamma \vdash \neg\neg\psi}$ $(\texttt{add}_{\neg\neg})$	$\texttt{add}_{\neg\neg}(\Box(x_1)) \blacktriangleright$ $\Box(\texttt{neg}(\texttt{neg}(x_1)))$
$\dfrac{\Gamma \vdash \psi \to \bot}{\Gamma \vdash \neg\psi}$ $(\texttt{intro}_{\neg})$	$\texttt{intro}_{\neg}(\Box(\texttt{implies}(x_1, \texttt{false}))) \blacktriangleright$ $\Box(\texttt{neg}(x_1))$
$\dfrac{\Gamma \vdash \neg\psi \to \bot}{\Gamma \vdash \psi}$ $(\texttt{elim}_{\neg\to})$	$\texttt{elim}_{\neg\to}(\Box(\texttt{implies}(\texttt{neg}(x_1), \texttt{false}))) \blacktriangleright$ $\Box(x_1)$
$\dfrac{\Gamma \vdash \forall x.\, \psi(x)}{\Gamma \vdash \psi(t)}$ $(\texttt{inst})$	$\texttt{inst}(y, \Box(\texttt{forall}(x))) \blacktriangleright$ $\Box(\textbf{apply} \cdot (x, y))$

Figure 5.4: Additional rules for the example.

Here our expectation, as the proof generator, is that

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

will be replaced by:

$$\forall x.\, \neg P(x) \vee Q(x, f(x))$$

which we express as:

let $q_3 := \texttt{forall}(\lambda x.\, \texttt{or}(\texttt{neg}(P(x)), Q(x, f(x))))$
let $f_{sk} := \texttt{and}(P(c), \texttt{and}(\texttt{eq}(c, d), \texttt{and}(q_3, q_2)))$
assert_ok `initial` f_{sk}

The first step of the actual proof is a partial CNF-conversion. Our CNF conversion uses Tseitin scheme, which introduces proxy literals for subformulas. This produces equisatisfiable set of clauses, yet the proof maps the proxy literals back to the original subformulas. Then the defining clauses of proxy literals become just basic boolean facts. We therefore derive clauses of the form $f_{sk} \to \neg\psi \to \bot$, where ψ is one of the conjuncts of f_{sk}, for example:

let $tmp := \lambda f.\, \texttt{assume}(\texttt{neg}(\texttt{eq}(c, d)), \lambda p.\, \texttt{elim}_{\neg}(p, \texttt{elim}_{\wedge 2}(\texttt{elim}_{\wedge 1}(f))))$
let $c_1 := \texttt{assume}(f_{sk}, tmp)$
assert_ok c_1 $\texttt{implies}(f_{sk}, \texttt{implies}(\texttt{neg}(\texttt{eq}(c, d)), \texttt{false}))$

and similarly we derive:

$$\begin{array}{l}\textbf{assert\_ok}\ c_0\ \texttt{implies}(f_{sk}, \texttt{implies}(\texttt{neg}(P(c)), \texttt{false}))\\ \textbf{assert\_ok}\ c_2\ \texttt{implies}(f_{sk}, \texttt{implies}(\texttt{neg}(q_3), \texttt{false}))\\ \textbf{assert\_ok}\ c_3\ \texttt{implies}(f_{sk}, \texttt{implies}(\texttt{neg}(q_2), \texttt{false}))\end{array}$$

Next we instantiate the quantifiers:

$$\begin{array}{l}\textbf{let}\ c_4 := \texttt{assume}(q_2, \lambda q.\, \texttt{assume}(Q(d, f(c)), \lambda i.\, \texttt{elim}_{\neg}(\texttt{inst}(f(c), q), i)))\\ \textbf{assert\_ok}\ c_4\ \texttt{implies}(q_2, \texttt{implies}(Q(d, f(c)), \texttt{false}))\\ \textbf{let}\ i_1 := \texttt{or}(\texttt{neg}(P(c)), Q(c, f(c)))\\ \textbf{let}\ c_5 := \texttt{assume}(q_3, \lambda q.\, \texttt{assume}(\texttt{neg}(i_1), \lambda i.\, \texttt{elim}_{\neg}(i, \texttt{inst}(c, q))))\\ \textbf{assert\_ok}\ c_5\ \texttt{implies}(q_3, \texttt{implies}(\texttt{neg}(i_1), \texttt{false}))\end{array}$$

Then we need to clausify i_1:

$$\begin{array}{l}\textbf{let}\ c_6 := \texttt{assume}(i_1, \lambda i.\, \texttt{assume}(P(c), \lambda o_1.\, \texttt{assume}(\texttt{neg}(Q(c, f(c))), \lambda o_2.\\ \qquad \texttt{elim}_{\neg}(o_2, \texttt{mp}(\texttt{elim}_{\vee}(i), \texttt{add}_{\neg\neg}(o_1))))))\\ \textbf{assert\_ok}\ c_6\ \texttt{implies}(i_1, \texttt{implies}(P(c), \texttt{implies}(\texttt{neg}(Q(c, f(c)), \texttt{false}))))\end{array}$$

Then we do some equality reasoning:

$$\begin{array}{l}\textbf{let}\ c_7 := \texttt{assume}(\texttt{neg}(Q(d, f(c))), \lambda l_n.\, \texttt{assume}(\texttt{eq}(c, d), \lambda e.\, \texttt{assume}(Q(c, f(c)),\\ \qquad \lambda l_p.\, \texttt{elim}_{\neg}(l_n, \texttt{mp}(\texttt{eq\_sub}(e, \lambda x.\, Q(x, f(c))), l_p)))))\\ \textbf{assert\_ok}\ c_7\ \texttt{implies}(\texttt{neg}(Q(d, f(c))),\\ \qquad\qquad \texttt{implies}(\texttt{eq}(c, d), \texttt{implies}(Q(c, f(c)), \texttt{false})))\end{array}$$

What remains is a pure boolean resolution. The resolution is realized by assuming the negation of the final clause and then using unit resolution of the assumed literals and some previous clauses, to obtain new literals, and as a last step, the `false` constant. We first resolve c_4 with c_7:

$$\begin{array}{l}\textbf{let}\ c_8 := \texttt{assume}(q_2, \lambda l_1.\, \texttt{assume}(\texttt{eq}(c, d), \lambda l_2.\, \texttt{assume}(Q(c, f(c)), \lambda l_3.\\ \qquad \texttt{mp}(\texttt{mp}(\texttt{mp}(c_7, \texttt{intro}_{\neg}(\texttt{mp}(c_4, l_1))), l_2), l_3)\\ \textbf{assert\_ok}\ c_8\ \texttt{implies}(q_2, \texttt{implies}(\texttt{eq}(c, d), \texttt{implies}(Q(c, f(c)), \texttt{false})))\end{array}$$

and finally we derive (also through resolution) the false constant:

$$\begin{array}{l}\textbf{let}\ kq_2 := \texttt{elim}_{\neg\rightarrow}(\texttt{mp}(c_3, \texttt{initial}))\\ \textbf{let}\ kq_3 := \texttt{elim}_{\neg\rightarrow}(\texttt{mp}(c_2, \texttt{initial}))\\ \textbf{let}\ kp := \texttt{elim}_{\neg\rightarrow}(\texttt{mp}(c_0, \texttt{initial}))\\ \textbf{let}\ ke := \texttt{elim}_{\neg\rightarrow}(\texttt{mp}(c_1, \texttt{initial}))\\ \textbf{let}\ kq := \texttt{elim}_{\neg\rightarrow}(\texttt{mp}(\texttt{mp}(c_6, \texttt{elim}_{\neg\rightarrow}(\texttt{mp}(c_5, kq_3))), kp))\\ \textbf{let}\ c_9 := \texttt{mp}(\texttt{mp}(\texttt{mp}(c_8, kq_2), ke), kq)\\ \textbf{final}\ c_9\end{array}$$

5.6 Implementation

We have implemented two versions of the proof checker: one full version in OCaml and a simplified one written in C. Proof generation was implemented inside the Fx7 (Moskal, 2007) SMT solver, implemented in the Nemerle programming language. The solver came second in the AUFLIA division of 2007 SMT competition, being much slower, but having solved the same number of benchmarks as the winner, Z3 (de Moura and Bjørner, 2008).

An important point about the implementation, is that at any given point, we need to store only terms that can be referenced by **let**-bound name, and thus the memory used by other terms can be reclaimed. As in our encoding the proof terms actually rewrite to formulas that they prove, there is no need to keep the proof terms around. We suspect this to be the main key to memory efficiency of the proof checker. The C implementation exploits this fact, the OCaml one does not.

Both implementations use de Bruijn (de Bruijn, 1972) indices in representation of lambda terms. We also use hash consing, to keep only a single copy of a given term. We cache normal forms of the terms, we remember what terms are closed (which speeds up beta reductions). Also a local memoization is used in function computing beta reduction to exploit the DAG structure of the term. The rewrite rules are only indexed by the head symbol, if two rules share the head symbol, linear search is used.

All the memoization techniques used are crucial (i.e., we have found proofs, where checking would not finish in hours without them).

The OCaml implementation is about 900 lines of code, where about 300 lines is pretty printing for Coq and Maude formats. The C implementation is 1500 lines. Both implementation include parsing of the proof and SMT formats and command line option handling. The implementations are available online along with the Fx7 prover.

5.6.1 Soundness Checking

The OCaml version of the checker has also a different mode of operation, where it reads the rewrite rules and generates corresponding formulas to be proven in the Coq proof assistant. There are three proof modes for rules:

- for simple facts about boolean connectives, arithmetic and equality, the checker generates a lemma and a proof, which is just an invocation of appropriate tactic
- for other generic schemas of proof rules from Section 5.3 and 5.4, the checker produces proof obligations, and the proofs need to be embedded in the rule descriptions
- for non-generic proof rules, the user can embed both the lemma and the proof in the rule description file, just to keep them close

Directory	Total	UNSAT	Fake	Fail
front_end_suite	2320	2207	101	12
		95.13%	4.35%	0.52%
boogie	908	866	25	17
		95.37%	2.75%	1.87%
simplify	833	729	44	60
		87.52%	5.28%	7.20%
piVC	41	17	10	14
		41.46%	24.39%	34.15%
misc	20	16	0	4
		80.00%	0.00%	20.00%
Burns	14	14	0	0
		100.00%	0.00%	0.00%
RicartAgrawala	14	13	0	1
		92.86%	0.00%	7.14%
small_suite	10	8	0	2
		80.00%	0.00%	20.00%

Figure 5.5: Results on the AUFLIA division of SMT-LIB.

This semiautomatic process helps preventing simple, low-level mistakes in the proof rules. The checker provides commands to define all these kinds of rules and associated proofs.

5.6.2 Performance Evaluation

When running Fx7 on a query there are five possible outcomes:

- it reports that the query is unsatisfiable, and outputs a proof
- it reports that the query is unsatisfiable, but since the proof generation is only implemented for the UTVPI fragment of linear arithmetic, the proof is correct only if we assume the theory conflicts to be valid (there is typically a few of them in each of such "fake" proofs)
- it reports the query is satisfiable, timeouts or runs out of memory

Tests were performed on AUFLIA benchmarks from the SMT-LIB (Ranise and Tinelli, 2006) This division includes first-order formulas, possibly with quantifiers, interpreted under uninterpreted function symbols, integer linear arithmetic and array theories. They are mostly software verification queries (cf. Section 1.1.1). The machine used was a 2.66GHz Pentium 4 PC with 1GB of RAM, running Linux. The time limit was set to ten minutes.

The results are given in Figure 5.5. The "Total" column refers to the number of benchmarks marked unsatisfiable in the SMT-LIB; "UNSAT" refers to the number of cases, where the benchmark was found unsatisfiable and

a correct proof was generated; "Fake" is the number of benchmarks found unsatisfiable, but with "fake" proofs; finally "Fail" is the number of cases, where Fx7 was unable to prove it within the time limit. It should be the case that UNSAT + Fake + Fail = Total. The percentages are with respect to the Total.

With the **C implementation**, proof checking a single proof never took more than 7 seconds. It took more than 2 seconds in 4 cases and more than 1 second in 19 cases (therefore the average time is well under a second). The maximal amount of memory consumed for a single proof was never over 7MB, with average being 2MB.

We have also tested the C implementation on a Dell x50v **PDA** with a 624MHz XScale ARM CPU and 32MB of RAM, running Windows CE. It was about 6 times slower than the Pentium machine, but was otherwise perfectly capable of running the checker. This fact can be thought of as a first step on a way to PCC-like scenarios on small, mobile devices. Other devices of similar computing power and, what is more important, RAM amount include most smart phones and iPods.

The **OCaml implementation** was on average 3 times slower than the C version, it also tends to consume more memory, mostly because it keeps all the terms forever (which is because of our implementation, not because of OCaml).

We have also experimented with translating the proof objects into the **Maude syntax** (Clavel et al., 2001). We have implemented lambda terms and beta reduction using the built-in Maude integers to encode de Bruijn indices and used the standard equational specifications for the first-order rules. The resulting Maude implementation is very compact (about 60 lines), but the performance is not as good as with the OCaml or C implementation — it is between 10 and 100 times slower than the OCaml one. It also tends to consume a lot more memory. The reason is mainly the non-native handling of lambda expressions. Beta reductions translate to large number of first-order rewrites, which are then memoized, and we were unable to instrument Maude to skip memoization of those.

We have performed some experiments using **Coq metalogic** as the proof checker. We did not get as far as implementing our own object logic. The main obstacle we have found was the treatment of binders. Performing skolemization on a typical input results in hundreds of Skolem functions. When using a higher order logic prover, such functions are existentially quantified and the quantifiers need to be pushed through the entire formula to the beginning. Later, during the proof, we need to go through them to manipulate the formula. This puts too much pressure on the algorithms treating of binders in the higher order prover. In our approach Skolem functions are bound implicitly, so there is no need to move them around. This is especially important in SMT queries, where the vast majority of the input formula is ground and quantified subformulas occur only deep inside the input. We can therefore keep most of the formula binder-free. We were not able to perform any realistic tests, as

Coq was running out of memory.

Both Maude and Coq are far more general purpose tools than just proof checkers. However relatively good results with Maude suggest that using a simple underlying formalism is beneficial in proof checking scenarios.

5.7 Related and Future Work

CVC3 (Barrett and Tinelli, 2007) and Fx7 were the only solvers participating in the 2007 edition of the SMT competition to produce formal proofs. The proof generation in CVC3 is based on the LF framework. We are not aware of a published work evaluating proof checking techniques on large industrial benchmarks involving quantifiers.

Formalisms for checking SMT proofs have been proposed in the past, most notably using an optimized implementation (Stump and Dill, 2002) of Edinburgh Logical Framework (Harper et al., 1987). However even with the proposed optimizations, the implementations has an order of magnitude higher memory requirements than our solution. Also the implementation of the checker is much more complicated.

Recently a Signature Compiler tool has been proposed (Zeller et al., 2008). It generates a custom proof checker in C++ or Java from a LF signature. We have run our proof checker on a 1:1 translation of the artificial EQ benchmarks from the paper. It is running slightly faster than the generated C++ checker. The memory requirements of our implementation are way below the size of the input file on those benchmarks. The checkers remain to be compared on real benchmarks involving richer logics and quantifiers.

In the context of saturation theorem provers it is very natural to output the proof just as a sequence of resolution or superposition steps. What is missing here, is the proof of CNF translation, though proof systems has been proposed (de Nivelle, 2005, 2003) to deal with that.

Finally, work on integrating SMT solvers as decision procedures inside higher order logic provers include (McLaughlin et al., 2006; Fontaine et al., 2006), (Conchon et al., 2007). The main problem with these approaches is that proof generation is usually at least order of magnitude faster than proof checking inside higher order logic prover. The Ergo (Conchon et al., 2007) paper mentions promising preliminary results with using proof traces instead of full proofs with Coq for theory conflicts. It is possible that using traces could also work for CNF conversion and skolemization. Yet another approach mentioned there is verifying the SMT solver itself.

An important remaining problem is the treatment of theory conflicts. One scenario here is to extend the linear arithmetic decision procedure to produce proofs. It should be possible to encode the proofs with just a minor extensions to the rewrite formalism. Another feasible scenario is to use a different SMT solver as a oracle for checking the harder (or all) theory conflicts. This can be applied also to other theories, like bit vectors or rational arithmetic.

5.8 Conclusions

We have shown how term rewriting can be used for proof checking. The highlights of our approach are:

1. time and space efficiency of the proof checker
2. simplicity of the formalism, and thus simplicity of the implementation
3. semiautomatic checking of proof rules

The main technical insight is that the proof rules can be executed locally. Therefore the memory taken by proofs trees can be reclaimed just after checking them and reused for the subsequent fragments of the proof tree.

Chapter 6

Conclusions and Future Research

This thesis tackled several issues concerning interconnection between software verification tools and SMT solvers. After introduction and setting up the context, Chapter 3 gave a survey of methods for programming a verification background theory in an SMT solver, using triggering annotations. Chapter 4 gave algorithms to be employed in an SMT solver, so it can respond promptly to such programming. Finally, Chapter 5 talked about correctness assurance of such responses.

The particular area of interest was verification it its full glory: sound, formal, concerned with high-level, functional properties. This is where the axiomatization techniques of Chapter 3 are most useful, and this is also where the PCC scenarios enabled by techniques of Chapter 5 allow for maximal trust improvement. The grand question, however, is whether it makes sense at all. Is it advisable to use SMT solvers for full-blown verification of functional properties of software? Or should we just stay with interactive theorem proving or, worse yet, informal correctness arguments, augmented by testing?

The experience with VCC, in the Hypervisor verification and otherwise, suggests that SMT is surprisingly powerful, when it comes to functional verification. This is supported by successful verification of several highly concurrent, but also significant in implementation size, algorithms in the Hypervisor. A small, self contained example is a reader-writer lock, verified inside the system with very natural external specifications (Hillebrand and Leinenbach, 2009).

Unfortunately, it is also very brittle, both in expressive capabilities and performance. While Chapter 3 lists several tools and methods for solving some of those problems, it would definitely be beneficial to work with a more predictable technology, and thus more research is needed on effective approaches to programming verification theories in SMT solvers. Still, SMT seems like a preferable choice, compared to interactive theorem proving, for main-stream verification.

When it comes to the theory programming, the triggering annotations

have much in common with term rewriting. Chapter 5 have demonstrated how first-order rewriting combined with lambda abstractions can be used to program a logic for a proof checker. The lambda abstractions, while theoretically very powerful, were easy to implement efficiently, giving rise to fast and lean proof checker. Possibly a similar approach could be used in the SMT solver instead, for programming verification theories.

□

Bibliography

(Babić and Hutter, 2008) Domagoj Babić and Frank Hutter. Spear theorem prover. In *Proc. of the SAT 2008 Race*, 2008.

(Ball et al., 2005) Thomas Ball, Shuvendu K. Lahiri, and Madanlal Musuvathi. Zap: Automated theorem proving for software analysis. In Geoff Sutcliffe and Andrei Voronkov, editors, *LPAR*, volume 3835 of *Lecture Notes in Computer Science*, pages 2–22. Springer, 2005. ISBN 3-540-30553-X.

(Barnett et al., 2005) Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2005. URL `http://www.springerlink.com/content/0m789xre652nuv06`.

(Barnett et al., 2006) Mike 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-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, September 2006.

(Barrett and Berezin, 2004) Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the Cooperating Validity Checker. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the* 16^{th} *International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, July 2004.

(Barrett and Tinelli, 2007) Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007. ISBN 978-3-540-73367-6.

(Barrett et al., 1996) Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods In Computer-Aided Design*, volume

1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, November 1996. Palo Alto, California, November 6–8.

(Barrett et al., 2008) Clark Barrett, Morgan Deters, Albert Oliveras, and Aaron Stump. Design and results of the 3rd annual satisfiability modulo theories competition (smt-comp 2007). *International Journal on Artificial Intelligence Tools*, 17(4):569–606, 2008.

(Barthe et al., 2006) Gilles Barthe, Lilian Burdy, Julien Charles, Benjamin Grégoire, Marieke Huisman, Jean-Louis Lanet, Mariela Pavlova, and Antoine Requet. Jack - a tool for validation of security and behaviour of java applications. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4709 of *Lecture Notes in Computer Science*, pages 152–174. Springer, 2006. ISBN 978-3-540-74791-8.

(Baumann et al., 2009) Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Better avionics software reliability by code verification – A glance at code verification methodology in the Verisoft XT project. In *Embedded World 2009 Conference*, Nuremberg, Germany, March 2009. Franzis Verlag. URL `http://www.uni-koblenz.de/~beckert/pub/embeddedworld2009.pdf`. To appear.

(Böhme et al., 2009) Sascha Böhme, Michał Moskal, Wolfram Schulte, and Burkhart Wolff. HOL-Boogie: An interactive prover-backend for the Verifiying C Compiler. *Journal of Automated Reasoning*, 2009. To appear.

(Bozzano et al., 2005) Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter Rossum, Stephan Schulz, and Roberto Sebastiani. MathSAT: Tight integration of SAT and mathematical decision procedures. *J. Autom. Reason.*, 35(1-3):265–293, 2005. ISSN 0168-7433. doi: http://dx.doi.org/10.1007/s10817-005-9004-z.

(Bryant et al., 2002) Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 78–92, London, UK, 2002. Springer-Verlag. ISBN 3-540-43997-8.

(Clavel et al., 2001) Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

(Cohen et al., 2009a) Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Markus Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of

Lecture Notes in Computer Science, pages 23–42, Munich, Germany, 2009a. Springer. Invited paper.

(Cohen et al., 2009b) Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. A practical verification methodology for concurrent programs. Technical Report MSR-TR-2009-15, Microsoft Research, February 2009b. Available from `http://research.microsoft.com/pubs`.

(Cohen et al., 2009c) Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. A Precise Yet Efficient Memory Model For C. In *Proceedings of Systems Software Verification Workshop (SSV 2009)*, 2009c. To appear.

(Conchon et al., 2007) Sylvain Conchon, Evelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. Lightweight integration of the Ergo theorem prover inside a proof assistant. In *Second Automated Formal Methods workshop series (AFM07)*, Atlanta, Georgia, USA, November 2007.

(Darvas and Leino, 2007) Ádám Darvas and K. Rustan M. Leino. Practical reasoning about invocations and implementations of pure methods. In Matthew B. Dwyer and Antónia Lopes, editors, *FASE*, volume 4422 of *Lecture Notes in Computer Science*, pages 336–351. Springer, 2007. ISBN 978-3-540-71288-6.

(Davis and Putnam, 1960) Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

(Davis et al., 1962) Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

(de Bruijn, 1972) N.G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

(de Moura and Bjørner, 2008) Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, April 2008.

(de Nivelle, 2003) Hans de Nivelle. Implementing the clausal normal form transformation with proof generation. In *fourth workshop on the implementation of logics*, pages 69–83, Almaty, Kazachstan, 2003. University of Liverpool, University of Manchester.

(de Nivelle, 2005) Hans de Nivelle. Translation of resolution proofs into short first-order proofs without choice axioms. *Information and Computation*, 199(1): 24–54, April 2005.

(DeLine and Leino, 2005) Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, March 2005.

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

(Detlefs et al., 1998) David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, December 1998.

(Dijkstra, 1972) E. W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–886, October 1972. Reprinted in *Programming Methodology, A Collection of Articles by Members of IFIP WG2.3*, D. Gries (ed.), Springer-Verlag, 1978.

(Dutertre and de Moura, 2006) Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006. ISBN 3-540-37406-X.

(Eén and Sörensson, 2003) Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. ISBN 3-540-20851-8.

(Filliâtre, 2003) Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

(Flanagan et al., 2002) Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37 of *SIGPLAN Notices*, pages 234–245. ACM, May 2002.

(Flanagan et al., 2004) Cormac Flanagan, Rajeev Joshi, and James B. Saxe. An explicating theorem prover for quantified formulas. Technical Report 199, HP Labs, 2004.

(Fontaine et al., 2006) Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *LNCS*, pages 167–181. Springer-Verlag, 2006.

(Ganzinger et al., 2004) H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In R. Alur and D. Peled, editors, *16th International Conference on Computer Aided Verification, CAV'04*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004.

(Ge et al., 2007) Yeting Ge, Clark Barrett, and Cesare Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In Frank Pfenning, editor, *CADE*, volume 4603 of *LNCS*, pages 167–182. Springer, 2007. ISBN 978-3-540-73594-6.

(Harper et al., 1987) Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings 2nd Annual IEEE Symp. on Logic in Computer Science, LICS'87, Ithaca, NY, USA, 22–25 June 1987*, pages 194–204. IEEE Computer Society Press, New York, 1987. URL `citeseer.ist.psu.edu/harper87framework.html`.

(Harvey and Stuckey, 1997) W. Harvey and P. Stuckey. A unit two variable per inequality integer constraint solver for constraint logic programming, 1997. URL `citeseer.ist.psu.edu/harvey97unit.html`.

(Hillebrand and Leinenbach, 2009) Mark A. Hillebrand and Dirk C. Leinenbach. Formal verification of a reader-writer lock implementation in C. In *4th International Workshop on Systems Software Verification (SSV 2009)*, Electronic Notes in Theoretical Computer Science. Elsevier Science B.V., 2009. To appear.

(Janota et al., 2007) Mikolás Janota, Radu Grigore, and Michał Moskal. Reachability analysis for annotated code. In *Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007)*, pages 23–30. ACM, September 2007. URL `http://www.cs.iastate.edu/~leavens/SAVCBS/2007/papers/Janota-Grigore-Moskal.pdf`.

(Kiniry and Cok, 2005) Joseph R. Kiniry and David R. Cok. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS 2004*, volume 3362 of *LNCS*, 2005.

(Kozen, 1977) Dexter Kozen. Complexity of finitely generated algebras. In *Proceedings of the* 9^{th} *Symposium on Theory of Computing*, pages 164–177, 1977.

(Lahiri and Qadeer, 2008) Shuvendu K. Lahiri and Shaz Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In George C. Necula and Philip Wadler, editors, *POPL*, pages 171–182. ACM, 2008. ISBN 978-1-59593-689-9.

(Leavens et al., 1999) Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

(Leino, 2009) K. Rustan M. Leino. This is Boogie 2, 2009. http://research.microsoft.com/~leino/papers/krml178.pdf.

(Leino and Monahan, 2009) K. Rustan M. Leino and Rosemary Monahan. Reasoning about comprehensions with first-order SMT solvers. In Sung Y. Shin and Sascha Ossowski, editors, *SAC*, pages 615–622. ACM, 2009. ISBN 978-1-60558-166-8.

(Leino et al., 2005) K. Rustan M. Leino, Todd Millstein, and James B. Saxe. Generating error traces from verification-condition counterexamples. *Sci. Comput. Program.*, 55(1-3):209–226, 2005. ISSN 0167-6423.

(Luckham et al., 1979) David C. Luckham, Steven M. German, Friedrich W. von Henke, Richard A. Karp, P. W. Milne, Derek C. Oppen, Wolfgang Polak, and William L. Scherlis. Stanford Pascal Verifier user manual. Technical Report CS-TR-79-731, Stanford University, Department of Computer Science, March 1979.

(Malik et al., 2001) Sharad Malik, Ying Zhao, Conor F. Madigan, Lintao Zhang, and Matthew W. Moskewicz. Chaff: Engineering an efficient sat solver. *Design Automation Conference*, 0:530–535, 2001. doi: http://doi.ieeecomputersociety.org/10.1109/DAC.2001.935565.

(Marché et al., 2004) Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The krakatoa tool for certificationof java/javacard programs annotated in jml. *J. Log. Algebr. Program.*, 58(1-2):89–106, 2004.

(McLaughlin et al., 2006) 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.

(Moskal, 2007) Michał Moskal. Fx7 or it is all about quantifiers, 2007. Also, http://nemerle.org/fx7/.

(Moskal et al., 2007) Michał Moskal, Kamil Skalski, Paweł Olszta, et al. Nemerle programming language, 2007. http://nemerle.org/.

(Moy, 2009) Yannick Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, January 2009.

(Necula, 1997) George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, jan 1997. URL citeseer.ist.psu.edu/article/necula97proofcarrying.html.

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

(Nieuwenhuis and Oliveras, 2005) R. Nieuwenhuis and A. Oliveras. Proof-Producing Congruence Closure. In J. Giesl, editor, *16th International Conference on Term Rewriting and Applications, RTA'05*, volume 3467 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 2005.

(Owre et al., 1992) S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag. URL `http://www.csl.sri.com/papers/cade92-pvs/`.

(Plotkin, 1972) Gordon D. Plotkin. Building-in equational theories. In D. Michie and B. Meltzer, editors, *Machine Intelligence*, pages 73–90. Edinburgh University Press, 1972.

(Ramakrishnan et al., 2001) I. V. Ramakrishnan, R. C. Sekar, and Andrei Voronkov. Term indexing. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1853–1964. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9, 0-262-18223-8.

(Ranise and Tinelli, 2006) Silvio Ranise and Cesare Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`.

(Saxe and Leino, 1999) Jim Saxe and K. Rustan M. Leino. ESC/Java design note 8a: The logic of ESC/Java, 1999. Available at `http://secure.ucd.ie/products/opensource/ESCJava2/ESCTools/docs/design-notes/escj08a.html`.

(Stump and Dill, 2002) A. Stump and D. Dill. Faster Proof Checking in the Edinburgh Logical Framework. In *18th International Conference on Automated Deduction*, 2002.

(Stump et al., 2002) Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A cooperating validity checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of the* 14^{th} *International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, July 2002. Copenhagen, Denmark.

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

(Zeller et al., 2008) Michael Zeller, Aaron Stump, and Morgan Deters. Signature compilation for the Edinburgh Logical Framework. *Electr. Notes Theor. Comput. Sci.*, 196:129–135, 2008.