PhD Dissertation

International Doctorate School in Information and Communication Technologies

DIT - University of Trento

Efficient Solving of the Satisfiability Modulo Bit-Vectors Problem and Some Extensions to SMT

Anders Franzén

Advisor:

Alessandro Cimatti

FBK-irst

Co-Advisor:

Roberto Sebastiani

Università degli Studi di Trento

March 2010

Abstract

Decision procedures for expressive logics such as linear arithmetic, bit-vectors, uninterpreted functions, arrays or combinations of theories are becoming increasingly important in various areas of hardware and software development and verification such as test pattern generation, equivalence checking, assertion based verification and model checking.

In particular, the need for bit-precise reasoning is an important target for research into decision procedures. In this thesis we will describe work on creating an efficient decision procedure for Satisfiability Modulo the Theory of fixed-width bit-vectors, and how such a solver can be used in a real-world application.

We will also introduce some extensions of the basic decision procedure allowing for optimisation, and compact representation of constraints in a SMT solver, showing how these can be succinctly and elegantly described as a theory allowing for the extension with minimal changes to SMT solvers.

Keywords

Decision Procedures, Satisfiability Modulo Theories, Bit-Vectors, Optimisation, Pseudo-Boolean Optimization, MAX-SAT

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Hardware and software systems have become incredibly complex and their usage is widespread. In designing these systems, verification is a highly challenging task. Hardware development costs are now routinely dominated by the verification effort, and there is a growing gap between our ability to design complex systems and the effort required to verify them, sometimes called the *verification gap*.

One avenue of research into closing this gap has been formal or semi-formal verification. A large number of techniques have been developed, including abstract interpretation, model checking, assertion based verification, equivalence checking, automatic test pattern generation, and many others. Many fully automated verification techniques depend on *bit-precise* reasoning to some extent, and this calls for the development of efficient and scalable techniques able to solve such formulae.

One approach to bit-precise reasoning is translation into the SAT problem and take advantage of the impressive performance of modern SAT solvers. These solvers are highly efficient, and are able to handle extremely large formulae. However, they are still very low level, and they also require some expertise in order to get the best possible performance out of them. For a given high-level formula at the RTL or bit-vector level, there are typically a vast array of different ways of encoding it into a corresponding SAT problem. Which encoding is chosen can have a very large impact on performance, and it may not be obvious which encoding is preferable without some knowledge of the inner workings of modern SAT solvers.

Solvers which reason at the level of bit-vectors can be seen as an answer to this problem. With these, the user can work at a higher level, not needing to worry about the low-level details of SAT solvers. Even if the bit-vector reasoner is implemented to ultimately use a SAT solver, the knowledge for how it is best used can be captured within the solver itself rather than in every application that needs to use it. A solver at the bit-vector level may also take advantage of the higher level of abstraction to either simplify the formula before attempting to solve it, or use alternative techniques in solving which may scale better for the formulae of interest to the user.

In recent years there has been an interest in solvers that support richer logics than propositional logic, and the field of Satisfiability Modulo Theories (SMT) has risen as a response to this interest. SMT combines propositional logic with one or more decidable theories such as linear arithmetic or bit-vectors to form a fully automated decision procedure at a higher level of abstraction. In the last few years, SMT has made tremendous progress, and SMT solvers are now being fielded in real-world applications both in academia and industry.

In this thesis, we will see how an efficient SMT solver for the theory of bit-vectors can be constructed, how it can be used, and what use it may be in a real-world application. We will also see some extensions to the SMT paradigm, pushing the growing usefulness of SMT into new areas.

1.1 Contribution

The contributions of this thesis lie in several novel techniques for solving of SMT formulae over the bit-vectors as well as some extensions to SMT. The main contributions can be summarised as follows:

- We introduce a simple and flexible framework for simplifying formulae based on term rewriting which is able to manage the potential complexity of simplification of bit-vector terms.
- We introduce several techniques, such as partitioning or variable splitting, that enhances other well known preprocessing or solving techniques
- We show how models can be computed while still applying all preprocessing techniques, rather than resorting to disabling one or more of them when models are requested.
- We show how the major preprocessing techniques can be used in an incremental solver.
- We introduce some novel under- and over-approximation techniques for bit-vector formulae.
- We introduce a lazy clustering scheme for dividing the theory solver consistency checks into multiple independent partitions.
- We show how it is possible to use minimal model enumeration in the lazy schema of SMT solving.
- We demonstrate how reusing learnt information from solving previous formulae can deliver significant performance enhancements in a real-world application without added implementation complexity
- We provide an extensive experimental evaluation, giving insight into the efficiency of the various techniques.
- We introduce a novel theory, the theory of costs, which allows for extension of satisfiability modulo theories into optimisation and compact

representation of Pseudo-Boolean constraints without modification to the standard SMT solver architecture.

Apart from the contributions of this thesis, we also try to give an overview of some of the techniques implemented within our SMT solver MathSAT that are relevant for the theory of bit-vectors, in the hope that it may give some insight into the observed performance of the solver.

Rather than just present techniques which have been proven to work in practise, this thesis will also discuss some techniques whose value has not (yet) been proven, or seem to have limited applicability. Using experimental evaluation, we will attempt to conclude which technique provide a benefit, which have limited use, and which may be unhelpful.

1.2 Acknowledgements

The work presented here has been in part supported by Semiconductor Research Corporation (SRC) under Global Research Collaboration (GRC) Custom Research Project 2009-TJ-1880 “WOLFLING”.

1.3 Overview

The thesis is organised as follows. Preliminaries such as notation and basic concepts are introduced in chapter 2. Chapter 3 describes techniques for solving formulae, chapter 4 covers techniques which simplify formulae before solving, and chapter 5 introduces approximation techniques.

Experimental evaluation of the techniques thus far introduced is found in chapter 6. In chapter 7 an industrial case study is presented, together with techniques for improving the usage of the solver in a real-world application. Chapter 8 introduces the theory of costs, which allows us to solve optimisation problems and encode Pseudo-Boolean constraints efficiently.

Finally, chapter 9 gives an overview of related work, and some conclusions and several suggestions for future work can be found in chapter 10.

Chapter 2

Preliminaries

Some background to the work presented in this thesis is necessary. In this chapter, we will introduce the concepts of propositional logic and the DPLL-style decision procedures often used to decide satisfiability in this logic. We will also introduce Satisfiability Modulo Theories, and how decisions procedures for this problem often works. Finally, we will give a brief overview of the MathSAT SMT solver, which is used as the proving ground for the techniques described in this thesis.

2.1 SAT

The satisfiability problem (SAT) is the problem of deciding satisfiability of formulae in propositional logic. Given a set of propositions $\mathcal{B}$, a propositional logic formula can be defined as

– $\top$ and $\bot$ are formulae.
– If $p \in \mathcal{B}$, p is a formula
– If α and β are formulae, then $\neg\alpha$ and $\alpha \wedge \beta$ are formulae.

A *truth assignment* or *interpretation* μ is a mapping from propositions to truth values $\{\text{false}, \text{true}\}$. We will also see a truth assignment as a set $\{p_1, \ldots, p_m\} \cup \{\neg q_1, \ldots, \neg q_n\}$ where the p_is are mapped to true and the q_is are mapped to false.

A truth assignment μ models the formula ϕ, denoted $\mu \models \phi$ given by the following.

$$\begin{array}{lll}
\mu \models \top & & \\
\mu \not\models \top & & \\
\mu \models p & \text{iff} & p \in \mu \\
\mu \models \neg\alpha & \text{iff} & \mu \not\models \alpha \\
\mu \models \alpha \wedge \beta & \text{iff} & \mu \models \alpha \text{ and } \mu \models \beta
\end{array}$$

The SAT problem can now be stated as the problem of determining for a given formula ϕ if there exists an interpretation such that $\mu \models \phi$. It is common to extend the language with several other connectives

- Disjunction $\vee$ defined as $\alpha \vee \beta$ iff $\neg(\neg\alpha \wedge \neg\beta)$
- Implication $\Rightarrow$ defined as $\alpha \Rightarrow \beta$ iff $(\neg\alpha) \vee \beta$
- Equivalence $\Leftrightarrow$ defined as $\alpha \Leftrightarrow \beta$ iff $\alpha \Rightarrow \beta$ and $\beta \Rightarrow \alpha$

We will call a *truth assignment* that gives values to all propositions in a formula ϕ a *total* truth assignment. A truth assignment which gives values to a strict subset of the propositions in a formula is called *partial*. We will define $\mathcal{A}toms(\phi)$ as the set of atoms in the formula ϕ.

Decision procedures for the SAT problem typically accept formulae in a particular form called Conjunctive Normal Form (CNF) defined as follows:

- An *atom* is a proposition $p \in \mathcal{B}$
- A *literal* is either an atom p or its negation $\neg p$. We say that a negated atom is a *negative literal* and a non-negated atom a *positive literal*. If l is a negative literal, by $\neg l$ we mean the corresponding positive literal.
- A *clause* is a disjunction of literals, often seen as a set of literals. A clause containing a single literal will be called a *unit clause*.
- A formula in CNF is a conjunction of clauses, often seen as a set of clauses.

All propositional formulae can be translated into CNF in linear time, if we are allowed to introduce fresh propositions as described in [Tse68]. Many variations on this technique have been proposed, and they may all be called Tseitin-style encodings meaning that they introduce fresh propositions to "give names" to subformulae allowing for a linear time translation.

2.2 Solving the SAT problem

The most popular approach to solving the SAT problem today is using a DPLL-style [DLL62] algorithm. In its most basic form, the algorithm may be outlined as in algorithm 2.1 taking a set of clauses as input and returning either $\top$ (the formula is satisfiable) or $\bot$ (the formula is unsatisfiable).

Algorithm 2.1: Basic DPLL algorithm DPLL(ϕ)

```
if $\phi = \emptyset$ then
    return $\top$
end
if $\emptyset \in \phi$ then
    return $\bot$
end
if Some $\{l\} \in \phi$ then
    return DPLL($\{c \setminus \{\neg l\} \mid c \in \phi \wedge l \notin c\}$)
end
$p \leftarrow$ some atom in $\phi$
return DPLL($\phi \cup \{p\}\}$) $\vee$ DPLL($\phi \cup \{\neg p\}$)
```

Many improvements have been proposed to the basic algorithm, which allows it to scale to large formulae. A good overview of techniques is [BHvMW09].

2.3 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) can be seen as an extension of propositional logic with some theory of interest such as linear arithmetic. The following introduction to SMT follows standard lines, a good reference is [BHvMW09].

We let $\Sigma = \langle \mathcal{F}, \mathcal{P} \rangle$ be a signature containing a set of function symbols $\mathcal{F}$ and a set of predicate symbols $\mathcal{P}$, each with an associated arity. We call the 0-arity function symbols constants, and the 0-arity predicates propositional symbols. We will call $\mathcal{F}^n$ the set of function symbols in $\mathcal{F}$ with arity n and $\mathcal{P}^n$ the set of predicate symbols in $\mathcal{P}$ with arity n. In this thesis we will focus on the quantifier free formulae constructed using this signature, which we will call ground formulae. The (free) variables in formulae will be seen as uninterpreted constant symbols in Σ. Given a signature $\Sigma = \langle \mathcal{F}, \mathcal{P} \rangle$ formulae can be built according to the following

- If $c \in \mathcal{F}^0$ then c is a term
- If $f \in \mathcal{F}^n$ and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term
- $\bot$ and $\top$ are formulae
- If $P \in \mathcal{P}^0$ then P is a formula
- If $P \in \mathcal{P}^n$ and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is a formula
- If α, β are formulae, then $\neg\alpha$, $\alpha \vee \beta$, $\alpha \wedge \beta$ $\alpha \Rightarrow \beta$ and $\alpha \Leftrightarrow \beta$ are formulae

The concepts of atoms, literals, clauses, CNF, and unit clauses lifts from propositional logic in the natural way. We let $\mathcal{V}ar(\phi)$ be the set of variables in the formula ϕ and $\mathcal{A}toms(\phi)$ be the set of atoms.

To provide semantics for this logic, we need a universe (the domain of terms), a mapping $[\![\cdot]\!]$ which assigns elements in the domain for every constant in $\mathcal{F}^0$. This mapping extends into an assignment over arbitrary terms over constants and predicates over constant terms into truth values.

A Σ-structure is a tuple consisting of a universe, an assignment of variables σ to elements in the domain, and an interpretation $[\![\cdot]\!]$ of all other nonlogical symbols. A Σ-formula is a formula using nonlogical symbols in Σ. A *sentence* is a formula without variables. A theory is a set of Σ-sentences. Given a theory T, a Σ-formula is satisfiable iff there exists a Σ-structure that satisfies both the formula and all sentences of T. A Σ-formula is valid in T iff all Σ-structures that satisfy the sentences of T also satisfies the formula.

The SMT-LIB[1] provides a publicly available benchmarks library for SMT formulae in a number of different theories, as well as definitions of several theories of interest.

2.4 Fixed-width bit-vectors

We define a theory of fixed-width bit-vectors similar to the theory defined in the SMT-LIB, an overview of the operators can be found in figure 2.1. The operators in the SMT-LIB bit-vector theory which are not included here are still supported in MathSAT, but translated into the operators shown here rather than handled natively.

We define semantics for bit-vector atoms in a way similar to Brinkmann and Drechsler [BD02].

- A bit-vector constant $x^{\langle n\rangle} \in \{0,1\}^n$ is a vector of n bits denoted $(x_{n-1}, \ldots, x_0)$.
- If $x^{\langle n\rangle}$ is a bit-vector, then $x^{\langle n\rangle}[i]$ is the ith bit x_i in $x^{\langle n\rangle}$.
- We define the auxiliary functions nat_n and bv_n such that $\mathsf{nat}_n(x^{\langle n\rangle}) = \sum_{i\in 1\ldots n} 2^n x(i-1)$ and we define bv_n to be the inverse of nat_n ($\mathsf{bv}_n = \mathsf{nat}_n^{-1}$).

[1] Available at `http://smt-lib.org/`

$t_1^{\langle m\rangle} :: t_2^{\langle n\rangle}$	Concatenation of two bit-vectors $t_1^{\langle m\rangle}$ and $t_2^{\langle n\rangle}$
$t^{\langle n\rangle}[i : j]$	Selection of bits j to i inclusively of $t^{\langle n\rangle}$
$\mathbf{not}\, t^{\langle n\rangle}$	Bit-wise negation of all bits in t
$t_1^{\langle n\rangle}\, \mathbf{and}\, t_2^{\langle n\rangle}$	Bit-wise and of all bits in t_1 with all bits in t_2
$t_1^{\langle n\rangle}\, \mathbf{or}\, t_2^{\langle n\rangle}$	Bit-wise or of all bits in t_1 with all bits in t_2
$t_1^{\langle n\rangle} \ll t_2^{\langle n\rangle}$	Shift left of t_1 by the amount given by t_2
$t_1^{\langle n\rangle} \gg_{\mathrm{l}} t_2^{\langle n\rangle}$	Logical shift right of t_1 by the amount given by t_2
$t_1^{\langle n\rangle} \gg_{\mathrm{a}} t_2^{\langle n\rangle}$	Arithmetic shift right of t_1 by the amount given by t_2
$t_1^{\langle n\rangle}\, \mathbf{rol}\, c$	Rotate left of t_1 by the amount given by $c \in [0, n-1]$
$t_1^{\langle n\rangle}\, \mathbf{ror}\, c$	Rotate right of t_1 by the amount given by $c \in [0, n-1]$
$\mathrm{zext}^{\langle m\rangle}(t^{\langle n\rangle})$	Zero extension of t to a bit-vector of m bits $(m \geq n)$
$\mathrm{sext}^{\langle m\rangle}(t^{\langle n\rangle})$	Sign extension of t to a bit-vector of m bits $(m \geq n)$
$t_1^{\langle n\rangle} + t_2^{\langle n\rangle}$	Addition of t_1 and t_2
$t_1^{\langle n\rangle} - t_2^{\langle n\rangle}$	Subtraction of t_1 and t_2
$-t^{\langle n\rangle}$	Unary subtraction
$t_1^{\langle n\rangle} * t_2^{\langle n\rangle}$	Multiplication of t_1 and t_2
$t_1^{\langle n\rangle} /_{\mathrm{u}} t_2^{\langle n\rangle}$	Unsigned division between t_1 and t_2
$t_1^{\langle n\rangle} /_{\mathrm{s}} t_2^{\langle n\rangle}$	Signed division between t_1 and t_2
$t_1^{\langle n\rangle}\, \mathbf{rem}_{\mathrm{u}}\, t_2^{\langle n\rangle}$	Unsigned remainder
$t_1^{\langle n\rangle}\, \mathbf{rem}_{\mathrm{s}}\, t_2^{\langle n\rangle}$	Signed remainder
$t_1^{\langle n\rangle} <_{\mathrm{u}} t_2^{\langle n\rangle}$	Unsigned less than
$t_1^{\langle n\rangle} <_{\mathrm{s}} t_2^{\langle n\rangle}$	Signed less than
$t_1^{\langle n\rangle} \leq_{\mathrm{u}} t_2^{\langle n\rangle}$	Unsigned less than or equal
$t_1^{\langle n\rangle} \leq_{\mathrm{s}} t_2^{\langle n\rangle}$	Signed less than or equal

Figure 2.1: Bit-vector operations

– We define $+$ as addition, $\cdot$ as multiplication, $/$ as division over natural numbers.
– We let σ be an assignment of variables to values in their domain, and define $[\![t]\!]_\sigma$ as the interpretation of the bit-vector term or atom t.

The semantics for most of the bit-vector operators can be seen in figure 2.2, and the rest are defined in terms of other operators in figure 2.3. When the specific width of a bit-vector term $t^{\langle n \rangle}$ is either irrelevant or clear from the context, it will often be dropped and we will simply write t.

2.5 Approaches to SMT

There are several different approaches to solving SMT formulae, they can be divided into two main categories of techniques called the *eager* and the *lazy* approaches.

2.5.1 Eager encoding into SAT

In the eager encoding into SAT the formula is translated into an equisatisfiable SAT instance which can then be solved in any SAT solver. How this translation is performed is theory-specific, for the theory of bit-vectors it can be performed by a process called *bit-blasting* or *flattening* which is basically the same technique used in hardware sysnthesis to generate a netlist from combinational RTL.

2.5.2 Lazy encoding

The lazy approach, sometimes also referred to as the DPLL(T) schema [NOT06], integrates solvers for the theories of interest into a SAT solver. The SAT solver, normally a solver in the DPLL-style, performs search on the logical structure of the formula, treating all predicates as propositional

$$
\begin{array}{rcl}
\llbracket 0^{\langle 1\rangle} \rrbracket_\sigma & = & (0) \\
\llbracket 1^{\langle 1\rangle} \rrbracket_\sigma & = & (1) \\
\llbracket c \rrbracket_\sigma & = & c \\
\llbracket c \rrbracket_\sigma & = & c \\
\llbracket v \rrbracket_\sigma & = & \sigma(v) \\
\llbracket t_1^{\langle m\rangle} :: t_2^{\langle n\rangle} \rrbracket_\sigma & = & \mathsf{bv}_{m+n}(2^m\, \mathsf{nat}(\llbracket t_1 \rrbracket_\sigma) + \mathsf{nat}(\llbracket t_2 \rrbracket_\sigma)) \\
\llbracket t[i:j] \rrbracket_\sigma & = & \mathsf{bv}_{i-j+1}(\llbracket t \rrbracket_\sigma / 2^j) \\
\mathbf{not}\, t^{\langle n\rangle} & = & \mathsf{bv}_n(2^n - 1 - \mathsf{nat}_n(\llbracket t^{\langle n\rangle} \rrbracket_\sigma)) \\
t_1^{\langle n\rangle}\, \mathbf{and}\, t_2^{\langle n\rangle} & = & (c_{n-1} \cdot d_{n-1}, \ldots, c_0 \cdot d_0) \text{ where } \llbracket t_1^{\langle n\rangle} \rrbracket_\sigma = (c_{n-1}, \ldots, c_0), \\
 & & \llbracket t_2^{\langle n\rangle} \rrbracket_\sigma = (d_{n-1}, \ldots, d_0) \\
\llbracket t_1^{\langle n\rangle} \ll t_2^{\langle n\rangle} \rrbracket_\sigma & = & \mathsf{bv}_n(2^k \cdot \mathsf{nat}_n(\llbracket t_1^{\langle n\rangle} \rrbracket_\sigma)) \text{ where } k = \mathsf{nat}_n(\llbracket t_2^{\langle n\rangle} \rrbracket_\sigma) \\
\llbracket t_1^{\langle n\rangle} \gg_{\mathrm{l}} t_2^{\langle n\rangle} \rrbracket_\sigma & = & \mathsf{bv}_n(\mathsf{nat}_n(\llbracket t_1^{\langle n\rangle} \rrbracket_\sigma)/2^k) \text{ where } k = \mathsf{nat}_n(\llbracket t_2^{\langle n\rangle} \rrbracket_\sigma) \\
\llbracket t_1^{\langle n\rangle}\, \mathbf{rol}\, k \rrbracket_\sigma & = & (c_{n-k-1}, \ldots, c_0, c_{n-1}, \ldots c_{n-k-2}) \text{ where } \llbracket t^{\langle n\rangle} \rrbracket_\sigma = (c_{n-1}, \ldots, c_0) \\
\llbracket t_1^{\langle n\rangle}\, \mathbf{ror}\, k \rrbracket_\sigma & = & (c_{k-1}, \ldots, c_0, c_{n-1}, \ldots, c_{n-k-2}) \text{ where } \llbracket t^{\langle n\rangle} \rrbracket_\sigma = (c_{n-1}, \ldots, c_0) \\
\llbracket \mathrm{zext}^{\langle m\rangle}(t^{\langle n\rangle}) \rrbracket_\sigma & = & \mathsf{bv}_m(\mathsf{nat}(\llbracket t \rrbracket_\sigma)) \\
\llbracket \mathrm{sext}^{\langle m\rangle}(t^{\langle n\rangle}) \rrbracket & = & (c_{m-1}, \ldots, c_0) \text{ where } c_i = d_{n-1} \text{ if } i \geq n \text{ and } c_i = d_i \text{ otherwise} \\
 & & \text{and } (d_{n-1}, \ldots, d_0) = \llbracket t \rrbracket_\sigma \\
\llbracket t_1^{\langle n\rangle} + t_2^{\langle n\rangle} \rrbracket_\sigma & = & \mathsf{bv}_n(\mathsf{nat}_n(\llbracket t_1 \rrbracket_\sigma) + \mathsf{nat}_n(\llbracket t_2 \rrbracket_\sigma)) \\
\llbracket -t^{\langle n\rangle} \rrbracket_\sigma & = & \mathsf{bv}_n(2^n - \mathsf{nat}_n(\llbracket t^{\langle n\rangle} \rrbracket_\sigma)) \\
\llbracket t_1^{\langle n\rangle} * t_2^{\langle n\rangle} \rrbracket_\sigma & = & \mathsf{bv}_n(\mathsf{nat}_n(\llbracket t_1^{\langle n\rangle} \rrbracket_\sigma) \cdot \mathsf{nat}_n(\llbracket t_2^{\langle n\rangle} \rrbracket_\sigma)) \\
\llbracket t_1^{\langle n\rangle} /_{\mathrm{u}}\, t_2^{\langle n\rangle} \rrbracket_\sigma & = & \mathsf{bv}_n(\mathsf{nat}_n(\llbracket t_1 \rrbracket_\sigma) / \mathsf{nat}_n(\llbracket t_2 \rrbracket_\sigma)) \\
\llbracket t_1^{\langle n\rangle}\, \mathbf{rem}_{\mathrm{u}}\, t_2^{\langle n\rangle} \rrbracket_\sigma & = & \mathsf{bv}_n(\mathsf{nat}_n(t_1^{\langle n\rangle})\, \mathbf{rem}\, \mathsf{nat}_n(t_1^{\langle n\rangle})) \\
\llbracket t_1^{\langle n\rangle} = t_2^{\langle n\rangle} \rrbracket_\sigma & = & \top \text{ if } \mathsf{bv}_n(\mathsf{nat}_n(\llbracket t_1 \rrbracket_\sigma) = \mathsf{nat}_n(\llbracket t_2 \rrbracket_\sigma)), \bot \text{ otherwise} \\
t_1^{\langle n\rangle} <_{\mathrm{u}} t_2^{\langle n\rangle} & = & \top \text{ if } \mathsf{bv}_n(\mathsf{nat}_n(\llbracket t_1 \rrbracket_\sigma) < \mathsf{nat}_n(\llbracket t_2 \rrbracket_\sigma)), \bot \text{ otherwise}
\end{array}
$$

Figure 2.2: Bit-vector semantics

$$\begin{array}{rcl}
t_1^{\langle n\rangle} \mathbf{or}\, t_2^{\langle n\rangle} & = & \mathbf{not}((\mathbf{not}\, t_1^{\langle n\rangle})\, \mathbf{and} (\mathbf{not}\, t_2^{\langle n\rangle})) \\
t_1^{\langle n\rangle} \gg_{\mathrm{a}} t_2^{\langle n\rangle} & = & \mathrm{ite}(t_1^{\langle n\rangle}[n-1] = 0^{\langle 1\rangle}, t_1^{\langle n\rangle} \gg_{\mathrm{l}} t_2^{\langle n\rangle}, \mathbf{not}((\mathbf{not}\, t_1^{\langle n\rangle}) \gg_{\mathrm{a}} t_2^{\langle n\rangle})) \\
t_1^{\langle n\rangle} - t_2^{\langle n\rangle} & = & t_1^{\langle n\rangle} + (-t_2^{\langle n\rangle}) \\
t_1^{\langle n\rangle} /_{\mathrm{s}}\, t_2^{\langle n\rangle} & = & \mathrm{ite}(t_1^{\langle n\rangle} \leq_{\mathrm{s}} 0 \wedge t_2^{\langle n\rangle} \leq_{\mathrm{s}} 0, t_1^{\langle n\rangle} /_{\mathrm{u}}\, t_2^{\langle n\rangle}, \mathbf{not}(u_1 /_{\mathrm{u}}\, u_2)) \text{ where} \\
 & & u_1 = \mathrm{ite}(t_1^{\langle n\rangle} <_{\mathrm{s}} 0, \mathbf{not}\, t_1^{\langle n\rangle}, t_1^{\langle n\rangle}) \text{ and } u_2 = \mathrm{ite}(t_2^{\langle n\rangle} <_{\mathrm{s}} 0, \mathbf{not}\, t_2^{\langle n\rangle}, t_2^{\langle n\rangle}) \\
t_1^{\langle n\rangle} \mathbf{rem}_{\mathrm{s}}\, t_2^{\langle n\rangle} & = & \mathrm{ite}(t_1^{\langle n\rangle} \leq_{\mathrm{s}} 0 \wedge t_2^{\langle n\rangle} \leq_{\mathrm{s}} 0, t_1^{\langle n\rangle} \mathbf{rem}_{\mathrm{u}}\, t_2^{\langle n\rangle}, \mathbf{not}(u_1 \mathbf{rem}_{\mathrm{u}}\, u_2)) \text{ where} \\
 & & u_1 = \mathrm{ite}(t_1^{\langle n\rangle} <_{\mathrm{s}} 0, \mathbf{not}\, t_1^{\langle n\rangle}, t_1^{\langle n\rangle}) \text{ and } u_2 = \mathrm{ite}(t_2^{\langle n\rangle} <_{\mathrm{s}} 0, \mathbf{not}\, t_2^{\langle n\rangle}, t_2^{\langle n\rangle}) \\
t_1^{\langle n\rangle} \leq_{\mathrm{u}} t_2^{\langle n\rangle} & = & \neg(t_2^{\langle n\rangle} <_{\mathrm{u}} t_1^{\langle n\rangle}) \\
t_1^{\langle n\rangle} \leq_{\mathrm{s}} t_2^{\langle n\rangle} & = & \neg(t_2^{\langle n\rangle} <_{\mathrm{s}} t_1^{\langle n\rangle}) \\
t_1^{\langle n\rangle} <_{\mathrm{s}} t_2^{\langle n\rangle} & = & \mathrm{ite}(t_1^{\langle n\rangle}[n-1] = t_2^{\langle n\rangle}[n-1], t_1^{\langle n\rangle} <_{\mathrm{u}} t_2^{\langle n\rangle}, t_1^{\langle n\rangle} = 1^{\langle n\rangle})
\end{array}$$

Figure 2.3: Syntactic sugar

atoms. We call the logical structure of the formula the *propositional abstraction* [Pla81] of the formula.

Definition The *propositional abstraction* of a ground formula ϕ is a propositional formula where all predicates in ϕ are replaced with propositions.

The lazy approach to SMT divides the reasoning into two parts; reasoning on the propositional abstraction of the formula, and reasoning in the theories of the formula. For the propositional abstraction a *boolean enumerator* is used, typically implemented using a DPLL-style SAT solver. The SAT solver proceeds by assigning truth values to atoms in the propositional abstraction, keeping track of the current truth assignment.

The SAT solver communicates the current truth assignments to the theory solvers, which given a set of such truth assignments determines consistency of the assignment in the theory. In the theory solver, this truth assignment is seen as a set of literals $L_1, \ldots, L_n$ which are positive iff the atom was assigned to true in the truth assignment. These truth assignments are communicated to the theory solvers during search in the SAT solver using an incremental and backtrackable interface to the theory solvers. Given a current partial truth assignment, it can be extended with a set of literals, or the last extension of the truth assignment can be retracted.

Further, when the theory solvers determine that the current truth assignment is inconsistent in the theory, a *conflict set* is produced which encapsulates the reason for the inconsistency. The conflict set is a subset of the current truth assignment, which in itself is inconsistent in the theory. This conflict set is used by the SAT solver to produce a conflict clause, which is used to prune further search.

Theory solvers are also allowed to deduce truth assignments to currently unassigned theory atoms. A deduced truth assignment is one which is a logical consequence in the theory of the current truth assignment, and will help the SAT solver prune the search.

Several improvements to the basic lazy approach have been proposed, see [Seb07, BHvMW09] for an overview.

2.6 DAG representation of formulae

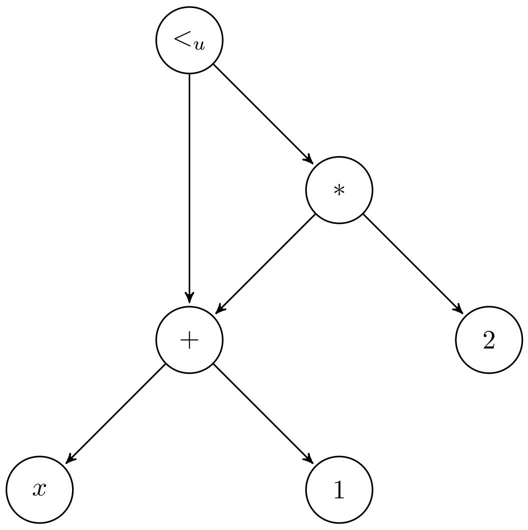
It has become a staple in SMT solvers to represent formulae using *perfect sharing*, sometimes also called aggressive sharing, structural hashing, hash consing, or common subexpression elimination. The plethora of names may be due to its popularity in many different fields such as functional programming languages [Got76], theorem proving [RV01] and compiler optimisation [Coc70]. Instead of storing a formula as a tree, it is stored as a directed acyclic graph (DAG). A subformula or term which is used several times in the formula will be represented with a single node in this DAG. Because formulae are represented in this way, we will use this when considering the number of occurrences of subformula or terms in a given formula.

Definition A term or formula is said to occur n times in a formula iff the node representing it in the DAG representation of the formula has n

incoming edges.

Example 2.1

Take the formula $x + 1 <_{\mathrm{u}} (x + 1) * 2$. The DAG representation of this term is shown below



and we can see that $x + 1$ occurs twice in this formula, while the variable x only occurs once.

■

2.7 MathSAT

MathSAT [BCF+08] is a SMT solver following the lazy schema, an overview of the architecture can be seen in 2.4. It accepts input in a number of different input formats, and also provides an API allowing MathSAT to be linked into other applications.

Roughly, the system consists of the following parts: A preprocessor, a DPLL-style boolean enumerator, and theory solvers.

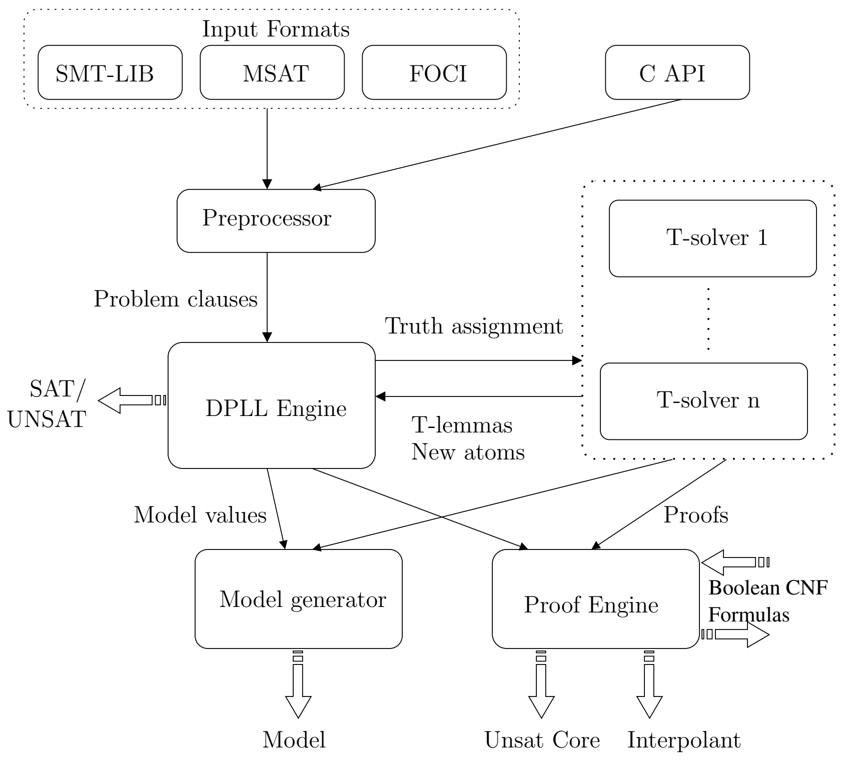


Figure 2.4: MathSAT architecture overview

2.7.1 Preprocessing

The preprocessor consists of several different parts: Simplification, conversion to CNF, static learning and initialisation of the solver.

Simplification During simplification the goal is to produce a new simpler formula which is equisatisfiable to the original. It is also required that given any model for the simpler formula it is possible to compute a model for the original formula. Examples of simplifications are computing a canonical form for atoms in linear arithmetic.

CNF conversion Conversion into CNF is performed with a Tseitin-style algorithm [Tse68].

Static learning Static learning will add some lemmas for the atoms occurring the formula. These lemmas are clauses which may help prune search, an example is lemmas for transitivity.

Solver initialisation Lastly the CNF is fed to the solver, and the solver is initialised. In this step the preprocessor instructs the solver to allocate the appropriate theory solvers and also provide some heuristic information about the formula which may help improve performance.

2.7.2 The solver proper

The heart of the solver is a boolean enumerator based on the MiniSat DPLL-style SAT solver, which enumerates models of the propositional abstraction of the formula. These models correspond to a truth assignment to the theory atoms, and this truth assignment is communicated to the theory solvers. The theory solvers receives these, and determines if this truth assignment is consistent in the theory.

2.7.3 Theories

MathSAT supports many of the theories of interest in practical applications, namely

– Equality and uninterpreted functions (EUF)
– Extensional arrays (ARR)
– Difference logic (DL)
– Unit two variable per inequality (UTVPI).
– Linear arithmetic over the real numbers and integers (LA)
– Fixed width bit-vectors (BV)

2.7.4 API

The MathSAT API is similar to that of Yices or Z3. The relevant operations are

Assert ϕ Assert that a formula must be true
Push backtrack point Remember the current state
Pop backtrack point Restore the state at the last backtrack point
Solve Solve the conjunction of assertions in the current state

In order to solve the three formulae α, $\alpha \wedge \beta$ and $\alpha \wedge \gamma$ we can do this in the following way

1. Assert α
2. Solve
3. Push backtrack point
4. Assert β
5. Solve
6. Pop backtrack point
7. Assert γ
8. Solve

When backtracking to a previous state, the solver is free to retain information which has been learnt previously which may help solve future formulae. An example is the theory conflicts which are universally valid and can therefore be reused regardless of what future formulae may look like.

2.7.5 Performance

Since the start of the annual SMT competition[2] MathSAT has been taking part in all categories it can support. The results are in brief:

[2]SMT-COMP is available at `http://www.smtcomp.org/`

- In 2005, MathSAT competed in 6 categories, placing second in one and third in 5.
- In 2006, MathSAT competed in 8 categories, placing second in two and third in 4.
- In 2007, MathSAT competed in 7 categories, placing second in two and third in two.
- In 2008, MathSAT competed in 9 categories, placing second in 3 and third in 4.
- In 2009, MathSAT competed in 12 categories, placing first in the bit-vector category and the category combining uninterpreted functions and integer difference logic. It also placed second in 7 categories and third in one.

2.7.6 Further reading

A more in-depth look at MathSAT wrt theories other than bit-vectors can be found in the PhD thesis of Alberto Griggio [Gri09].

Chapter 3

Solving techniques

There are two main techniques for solving bit-vector formulae that are used within MathSAT: Translation into SAT and the lazy approach to SMT, also called DPLL(T). Both the eager encoding into SAT and the lazy approach are in this work based on bit-blasting to handle bit-vector atoms. In this chapter, we will look at how these two techniques are used within MathSAT, as well as some auxiliary techniques such as layering, static learning, or modifications to the basic boolean enumeration algorithm.

We will start this chapter by looking at bit-blasting, since it is used in both approaches to solving. Then we will discuss the lazy approach and give some details on the bit-vector theory solver, followed by the eager approach. We continue with the use of EUF layering in MathSAT, static learning, clustering, and minimal model enumeration.

3.1 Bit-blasting

The bit-blasting used in MathSAT converts bit-vector atoms directly to CNF, rather than first creating a propositional formula in some other format such as general propositional logic or And Inverter Graphs (AIGs). The reason for this is purely historical, an early version which bit-blasted atoms first to general propositional logic which was then converted into

CNF turned out to be unnecessarily slow on large but trivial instances. A disadvantage of going directly to CNF is that propositional preprocessing techniques such as those described in [BB06] can not be applied.

Each atom is converted using a Tseitin-style CNF transformation [Tse68], taking care that each atom is represented by a propositional literal, which is not added (as a unit) to the CNF but kept separate. In this way, the CNF for a particular literal is guaranteed to be satisfiable. Adding the representative literal asserts that the atom is true and adding the negation of the literal asserts that the atoms is false. Example 3.1 shows how this might work in practise.

Example 3.1

We can bit-blast and convert the atom $x^{\langle 1\rangle} <_{\mathrm{u}} y^{\langle 1\rangle}$ into CNF by adding one fresh Tseitin variable v which is meant to "represent" the atom and create the clauses

$$\{\neg v, \neg x\},\ \{\neg v, y\},\ \{v, x, \neg y\}$$

If in the truth assignment the atom is true, we solve these clauses under the assumption of v. It the atom is false, we solve under the assumption $\neg v$. ∎

Most bit-vector terms are bit-blasted in a straightforward way. E.g. relations are bit-blasted using comparators, addition and subtraction using ripple-carry adders and so on. Some small concessions to performance have been done, such as strength reduction for multiplication or division by constant [War02].

3.2 DPLL(T) or the lazy schema

The minimal requirements for a theory solver in the DPLL(T) or lazy framework is in short

Incrementality The current truth assignment is extended incrementally by communicating the new truth assignments to the theory solver.

Backtrackability During backtracking, a number of the literals on the truth assignment are retracted. The retracted literals are always those last added

Consistency checking Given a particular truth assignment, the theory solver should be able to detect inconsistent truth assignments, and be able to provide models for consistent truth assignments.

Conflict set generation For inconsistent sets of literals a conflict set should be communicated back to the boolean enumerator. This is a subset of the current truth assignment, which in itself is inconsistent.

The underlying SAT solver is a modified version of MiniSat [ES04], with the following modifications

– Component caching [PD07b], sometimes also called progress saving or phase caching. This is a technique which stores the phase of all assignments made, and when making a new decision it checks the last decision made on this variable and makes the same one.
– Blocking literals [SE08]. This helps reduce the number of memory references for unit-propagation on already satisfied clauses. A copy of one of the literals is kept in the watch list data structure. If this literal is satisfied, there is no need to visit the clause itself.
– More frequent restarts than the restart strategy implemented in MiniSat.

Atoms are bit-blasted as described in section 3.1 taking care that the Tseitin-style CNF conversion for each bit-vector atom A produces a Tseitin literal l which is equivalent to the bit-blasted atom, and we keep a mapping between bit-vector atoms and the corresponding Tseitin literal. Normally this literal would be added to the CNF, but here we will not add it to the

SAT solver; In this way, the CNF for each atom is guaranteed to be satisfiable. In order to check consistency of a set of bit-vector literals, $L_1, \ldots, L_n$ we collect the corresponding Tseitin literals $l_1, \ldots, l_n$, negating them iff the bit-vector literal was negative. We then solve the bit-blasted formula assuming this set of literals enforcing the truth values of the atoms. Solving under assumption of a number of literals is supported in a number of SAT solvers, like MiniSat [ES04] or PicoSAT [Bie08].

Should the formula be unsatisfiable under these assumption, we can compute a conflict in terms of the assumed literals. This can be used to compute a conflict set, which although not guaranteed to be minimal, often is minimal or close to minimal in practise.

There are also a number of other features of theory solvers, which although not strictly necessary may be advantageous:

Early pruning Checking consistency of partial truth assignments

Deduction Deducing literals not currently on the truth assignment

For early pruning, we will use what we shall call bounded SAT reasoning.

3.2.1 Bounded SAT reasoning

When checking partial truth assignments, it is possible to check consistency in the same way as for total truth assignments, but this may cause considerable overhead. Therefore an incomplete procedure is often used. In MathSAT, the theory solver performs search with an upper bound on the number of conflicts. The default is to only perform unit propagation, and report any conflicts found. It is however also possible to do search for up to a given number of conflicts. If the truth assignment is found to be inconsistent within this limit, a conflict set is returned. Otherwise search in the boolean enumerator continues.

3.2.2 Deduction

It is also possible to deduce literals. A simple way is to perform unit propagation, and then deduce all literals which have been given a truth value by the unit propagation. However this is not yet implemented in MathSAT.

3.3 Eager encoding to SAT

The *eager* approach to SMT consists of solving formulae by translation into an equisatisfiable SAT problem, and solving that in a standard SAT solver. For bit-vectors this translation is straightforward, by what is called *bit-blasting* or *flattening*.

3.3.1 Implementation issues

To achieve this encoding in a DPLL(T) style solver like MathSAT without major modification, there are two different approaches:

- Bit-blast the formula in preprocessing. This will produce a purely propositional formula, which can be solved by the boolean enumerator without the help of any theory solver.
- Convert the formula into a bit-vector atom. Propositions can be replaced with fresh single-bit bit-vector variables, and the logical structure of the formula can be encoded using bit-wise operators. This atom can then be solved by the bit-vector theory solver.

In MathSAT, both techniques are supported, but for pure bit-vector formulae the second approach is the default. The first approach can still be useful, for instance in cases with formulae that contains other theories disjointly, with a small number of bit-vector atoms. Implementation-wise, in the second approach the formula is implicitly transformed into a conjunction of bit-vector atoms. The implicit transformation is simply implemented by marking all conjuncts as bit-vector atoms. The boolean enumerator will then treat these as if they were real bit-vector atoms, and the theory solver is extended with support for bit-blasting bit-vector formulae instead of just atoms.

3.4 Theory solver layering

A technique which has been in use for some time in MathSAT is *layering* [BBC+05b]. The underlying idea is that given a truth assignment that is unsatisfiable it is frequently very "obviously" inconsistent. Reasonably it should therefore be correspondingly easy to detect the inconsistency, and there should be no need to use a potentially expensive decision procedure to do so. In MathSAT, it is possible to allocate a number of different theory solvers which will each handle some subset of the atoms in the formula or reason on an abstraction of the atoms. As an example, for linear arithmetic, one can allocate one EUF solver treating all arithmetic operators as uninterpreted functions, one difference logic solver only considering the subset of atoms in difference logic, and finally a theory solver for full linear arithmetic, which can be used when all else fails. For bit-vectors, it is possible, apart from using a solver for bit-vectors, to also use a solver for EUF as a layer above the bit-vector solver. The intuition is that in cases when it can aid in search it will do so at low cost, and in cases where it cannot it is a very low overhead compared to a full bit-vector solver.

In order to enhance the power of the EUF solver, it has been extended to support some semantics of other theories. In particular, different bit-vector constants are detected as different from each other, relations over bit-vector constants can be checked for consistency, and strict less than or greater than relations over other terms are interpreted as disequalities.

This gives the EUF some extra capability of detecting conflicts over bit-vectors while still maintaining the same computational complexity and efficiency.

3.5 Static learning

The idea behind static learning [BBC+05b, YM06] is to add lemmas which are valid in the theory. This is done by instantiating a few basic axioms, such as axioms for transitivity of equality, mutual exclusion of inequality and similar.

3.6 Clustering

The idea of dividing the set of theory atoms into independent sets, called *clustering* was first introduced in [BBC+05a] for EUF and linear arithmetic, but it generalises also to other theories. Before search starts, the theory atoms are divided into a set of independent sets each of which can not interfere with the satisfiability of any other. Then for each such set, a separate theory solver is used. In this way we can reduce the amount of theory literals each theory solvers need to reason with, and hopefully avoid some unnecessary complexity.

Definition Two atoms A_1, A_2 belong to the same cluster iff $\mathcal{V}ar(A_1) \cap \mathcal{V}ar(A_2) \neq \emptyset$

Definition A *clustering* of a set of atoms A is a partition of this set induced by the cluster relation.

There are at least two ways of using clustering to split the problem into several hopefully simpler problems to solve. One is to partition the set of atoms statically before solving, the other is to perform clustering on the truth assignment of literals when using the lazy schema. The former has been applied in MathSAT in the past [BBC+05b], and applied for the theory of bit-vectors in [BCF+07]. For the industrial bit-vector instances used in the latter paper, the clustering typically generated hundreds of clusters, so it would appear to be very efficient. Looking closer however, most of these clusters contain a handful of atoms containing a single variable, making these clusters trivial. There was also often one large cluster containing most of the atoms. Most of the complexity of reasoning remains in this large cluster.

To achieve a more fine-grained clustering, a *local* rather than global approach must be taken, and if we are using the lazy schema this is possible to do. Instead of clustering all theory atoms up-front during preprocessing, we can attempt to cluster all literals occurring on each truth assignment. Since this may be a subset of all atoms, there is the possibility that this will produce more clusters, and simpler problems to solve. For every truth assignment of literals L, we perform clustering of the set of atoms $\mathcal{A}toms(L)$ on that truth assignment producing several clusters of literals $L_1, L_2, \ldots, L_n$. Each cluster can now be checked for consistency independently.

Clustering of truth assignments makes it more difficult to build a theory solver which retains information from one consistency check to the next however, because the clusters of literals may be different from one call to the next. We can still create an incremental theory solver that retains learnt information, if we relax the requirement for clustering a little.

When the theory solver is asked to check the consistency of its first truth assignment, we perform clustering of this truth assignment and allocate one theory solver for each cluster. When it needs to check consistency of a new truth assignment, we check the atoms on the new truth assignment which have not been seen before. If the variables of some of new atom occurs

Table 3.1: Three value logic semantic of dual rail encoding

$P^\top$	$P^\perp$	Meaning
False	False	No value
False	True	False
True	False	True
True	True	Illegal

in more than one cluster currently created, these clusters are merged into a single cluster. This is done by identifying the largest of these clusters and merging all others into it. The other clusters are then removed and the new atom added to the merged cluster. This is just an approximation of the original clustering technique because for a given truth assignment, some clusters on that truth assignment may now be handled by a single theory solver.

3.7 Minimal model enumeration

We would like to reduce the number of literals sent to the bit-vector theory solver, since each theory solver call is potentially very expensive. One way to do this is to have the boolean enumerator enumerate *minimal* models. In [RC06], Roorda and Claessen uses a technique based on a *dual-rail* encoding which gives minimal models for the SAT problem, and the same technique lifts into SMT.

In a dual rail encoding of a formula, each propositional atom P is replaced by two fresh atoms $P^\top$ and $P^\perp$. These are used to encode a three-valued semantics of propositional logic according to table 3.1. To translate a formula in CNF to dual rail, all positive literals A are replaced with $A^\top$, and all negative literals $\neg A$ are replace with $A^\perp$. To rule out the illegal value, for every atom A the clause $\{\neg A^\top, \neg A^\perp\}$ is added to the CNF.

3.7.1 Sign-Minimal Models

To see why this encoding would help in enumerating minimal models, we can notice that in DPLL, if the decision heuristic always assigns false to decision variables, then any model μ for a set of clauses Γ has the minimal number of positive literals. This means that it is not possible to negate any of the positive literals in μ and still have $\mu \models \Gamma$. We say that such a model is *(positive) sign-minimal*. The reverse is true if the decision heuristic always assigns true to decision variables, and we call such models negative sign-minimal.

To prove this, we show an invariant that holds during search. We show that there exists a subset of Γ such that the current interpretation μ will always be a sign-minimal model for that subset. Let us call $\Sigma \subseteq \Gamma$ the interesting subset. We will only cover the case were the heuristic assign false, the other case is analogous.

Init We have that $\mu = \emptyset$. Let $\Sigma = \emptyset$, and μ will be a sign-minimal model of Σ.

Decision Making a decision on a variable v will add $\neg v$ to μ. The extended interpretation will still be a sign-minimal model of Σ.

Unit Propagation If a literal l is unit-propagated, the reason is a clause in $\Gamma \setminus \Sigma$, since it has to be an unsatisfied clause, and all clauses in Σ are satisfied under μ. If we extend Σ with this new clause, the extended μ will be a sign-minimal model of the extended Σ.

Backtracking If we backtrack to a previous decision level, we can remove any clauses from Σ which were added below that decision level. This will restore both μ and Σ to the same state they were in when we entered that decision level. Therefore, the reduced μ will be a sign-minimal model for the reduced Σ.

Adding a Conflict Clause For any new conflict clause c, $\Gamma \models c$. So, for any μ', $\mu' \models \Gamma$ iff $\mu' \models \Gamma \cup \{c\}$. Therefore, an interpretation μ' is a sign-minimal model of Γ iff it is also a sign-minimal model of $\Gamma \cup \{c\}$.

Complete models In a complete model, the invariant gives us that μ is a sign-minimal model for a subset of the clauses. Therefore, it must also be a sign-minimal model for all clauses.

3.7.2 Minimality for Standard Dual Rail

Sign-minimality and assigning decision variables to false gives us minimality in dual rail, since only assignments to true on dual rail atoms correspond to an assignment in the three value logic.

3.7.3 Minimality in SMT

In MathSAT, all theory conflicts consist of all negative dual rail literals. They can never in themselves force a truth value to any literal, and so minimality for the propositional abstraction is preserved.

For theory deduction, it can be encoded as an implication clause which is identical to the conflict clause that would have been added had the implied literal been assigned the inconsistent truth value. Therefore minimality is preserved.

3.7.4 Encoding of non-CNF formulae

Encoding of non-CNF formulae is straightforward. For every subformula ϕ we can create the dual-rail tuple $\langle \phi^{\top}, \phi^{\perp} \rangle$. So, for a conjunction $\alpha \wedge \beta$ the

Table 3.2: Dual-rail encoding of connectives

Connective	Encoding
$\alpha \wedge \beta$	$\langle \alpha^\top \wedge \beta^\top, \alpha^\perp \vee \beta^\perp \rangle$
$\alpha \vee \beta$	$\langle \alpha^\top \vee \beta^\top, \alpha^\perp \wedge \beta^\perp \rangle$
$\neg\alpha$	$\langle \alpha^\perp, \alpha^\top \rangle$

dual rail encoding would be simply $\langle \alpha^\top \wedge \beta^\top, \alpha^\perp \vee \beta^\perp \rangle$. An encoding for some common connectives can be seen if figure 3.2. Translation to CNF can be performed in the normal way of the two formulae in the tuple.

3.7.5 Redundancy

The minimal model enumeration shown here does come with a price, and the price to pay is in *redundancy* [ABC+02] of enumerated models.

Definition Given a set of interpretations $I = \{\mu_1, \mu_2, \ldots, \mu_N\}$, we say that this set is *non-redundant* iff for every $\mu_i \in I$ the set $I' = I \setminus \{\mu_i\}$ is not a cover of I.

Normally a DPLL-style enumerator will enumerate non-redundant models, once a cover for the formula has been computed, it will terminate. But with the dual rail encoding we will enumerate every minimal model, as can be seen in example 3.2 where we enumerate all models of a simple formula.

Example 3.2

Take the formula $(A \wedge B) \vee (\neg A \wedge C) \vee (\neg B \wedge \neg C)$. This formula has the following minimal models

$$\begin{aligned} \{A, B\} \\ \{\neg A, C\} \\ \{\neg B, \neg C\} \\ \{\neg A, \neg B\} \\ \{B, C\} \\ \{A, \neg C\} \end{aligned}$$

In this example, it is enough to enumerate either the first three or the last three to cover all models of the formula. However, with a dual-rail encoding we will enumerate all six. This is easy to see by stepping through enumeration. The formula can be written in CNF as $\{A, \neg B, C\}$, $\{\neg A, B, \neg C\}$, and encoded in dual rail the formula becomes

$$\{A^\top, B^\perp, C^\top\},\ \{A^\perp, B^\top, C^\perp\}$$

plus the clauses ruling out the forbidden value for each original variable, not show here. One model for this is $\{A^\top, \neg A^\perp, B^\top, \neg B^\perp, \neg C^\top \neg C^\top\}$ corresponding to the minimal model $\{A, B\}$. Adding a blocking clause $\{\neg A^\top, \neg B^\top\}$. We can iterate until we have found the next two models, adding the corresponding blocking clauses

$$\{\neg A^\top, \neg B^\top\},\ \{\neg A^\perp, \neg C^\top\},\ \{\neg B^\perp, \neg C^\perp\}$$

Even though we have now covered all models, the set of clauses are still satisfied, e.g. with the model

$$\{\neg A^\top, \neg A^\perp, B^\top, \neg B^\perp, C^\top \neg C^\top\}$$

and it is only when blocking clauses for all minimal models have been added that the set of clauses become unsatisfiable. ■

Chapter 4

Preprocessing

Many instances, especially those coming from practical application of decision procedures in industry have an inefficient encoding. There may be a great number of redundancies, subformulae which are trivially unsatisfiable, and irrelevant subformulae which do not affect satisfiability. These may cause significant slowdown when trying to solve a formula when compared to a more clever encoding of the same problem.

In this chapter, we will look at some preprocessing techniques which can help in producing a simpler equisatisfiable formula from the input instance which can be fed into the underlying solver. The requirements for all preprocessing is

1. The preprocessed formula must be equisatisfiable to the original
2. For any model of the preprocessed formula, it must be possible to compute a model for the original formula.

A desirable property is also that the preprocessing is relatively inexpensive, but it is not clear that it must be so. If some preprocessing step drastically reduces solving time, even a potentially expensive technique may be worthwhile. It is also desirable that the preprocessing techniques support incremental solving, so that they can be used in the incremental interface to MathSAT. However, some of the techniques described here do not easily support this.

In this chapter we will look at a number of different techniques:

Normalisation Basic simplifications

Substitution Eliminating variables or propositions

Propagation of unconstrained terms Removing irrelevant parts of the formula

Disjunctive partitioning Splitting the formula into independent parts

Packet splitting Splitting variables into several parts

Difference propagation Taking advantage of the fact that we know terms to be different from one another

Miscellaneous A collection of minor techniques

We will also see how model can be efficiently computed while using all the above techniques, how we can support preprocessing techniques in an incremental solver, and a few words on the architecture used for preprocessing in MathSAT.

4.1 Normal form computation

In formulae generated in real-world applications, the encoding of the problem is often filled with terms which can be trivially simplified. Let's look at a small motivational example:

Example 4.1

Given the equality $x + 2 - (y - 1) = 2 * x + 3$ which we would like to solve, we can see several opportunities for simplification. We can start by simplifying the left hand side into $x - y + 3 = 2 * x + 3$ and then into $y = 2 * x - x + 3 - 3$ which further simplifies into $y = x$. ▮

One possibility in achieving this simplification would be to compute a *canonical form* for bit-vector atoms. This is however an expensive proposition, it is in fact NP-hard [BDL98]. A more appealing alternative is to perform simplifications which although not producing a canonical form are both effective in practise and induces a low computational overhead. There are many ways of implementing simplifications such as those seen in example 4.1. In this thesis we will see simplifications as rewrite rules forming a simple term rewrite system.

Definition A *rewrite rule*, written $s \rightarrow t$, has the property that s is not a variable and $\mathcal{V}ar(t) \subseteq \mathcal{V}ar(s)$. A *term rewriting system* (TRS) is a set of rewrite rules.

Example 4.2

A simple rewrite rule for addition is $0 + x \rightarrow x$ ▮

Rewrite rules are unless explicitly specified defined on non-fixed size bit-vectors, the above example can be used to simplify addition with zero for all bit-vector widths, it could also be written as $0^{\langle n\rangle} + x^{\langle n\rangle} \rightarrow x^{\langle n\rangle}$ An example of a rule for a specific width might be

$$1^{\langle 1\rangle} + x^{\langle 1\rangle} \rightarrow \mathbf{not}(x)$$

which is applicable only on bit-vectors of width 1. Simplification is done by applying all rewrite rules to a fix-point, in term rewriting called the *reflexive transitive closure*, denoted $t \xrightarrow{*} t'$. Given a term rewriting system, a term which cannot be rewritten any further is said to be in *normal form*, and hence we will call these basic simplifications of bit-vector terms *normalisation*. For more information on term rewriting, Baader and Nipkow [BN98] is a good introduction. Here we will introduce only the parts necessary in this application.

A rewrite rule $s \rightarrow t$ is applicable on a term u iff the left hand side t matches u.

Definition Given two terms s and t, the *matching problem* is the problem of deciding whether there exists a substitution σ such that $\sigma(s) = t$.

For instance the rewrite rule in example 4.2 is applicable on the term

$$0^{\langle 32\rangle} + (y^{\langle 16\rangle} :: z^{\langle 16\rangle})$$

with the substitution $\sigma = [x \mapsto y^{\langle 16\rangle} :: z^{\langle 16\rangle}]$. In this work we will use *conditional* rewrite rules. A conditional rewrite rules is of the form $s, c_1, \ldots, c_n \rightarrow$ where $c_1, \ldots, c_n$ are conditions which must all be fulfilled for the rule to be applicable. An example of a conditional rewrite rule is $x^{\langle n\rangle}[u : l],\ u = n - 1,\ l = 0 \rightarrow x^{\langle n\rangle}$ which removes “unnecessary” selection operators. We also define some predicates and functions which can be used in conditions, such as

- $\mathsf{const}(t)$, which is true iff t is a bit-vector constant
- $\mathsf{nat}(t)$, which converts t to the corresponding natural number if it is a constant
- $\mathsf{eval}(t)$, which given a bit-vector term not containing any variables, evaluates it to the corresponding bit-vector constant.
- $t_1 \prec t_2$, a total ordering on terms
- $=$, which check if two terms are equal
- The logical connectives $\neg$, $\vee$ with the usual meaning

The eval function can also be used in the right hand side of rules. Using these operators, it is possible to define rules like $x+y,\ \mathsf{const}(x),\ \mathsf{const}(y) \rightarrow \mathsf{eval}(x + y)$ for evaluation of additions, or $x + y,\ y \prec x \rightarrow y + x$ which encodes commutativity of addition as a rewrite rule.

An important property for rewrite systems is termination, which is defined as follows.

Definition A term rewriting system is *terminating* iff there is no infinite rewrite sequence $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \ldots$

There are two ways of causing non-termination for bit-vector rewrites.

- We may have cyclic rewrites such that $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t_1$, e.g. the rewrite rule $x + y \rightarrow y + x$.
- We may have a rewrite system which can grow the size of a term indefinitely. E.g., the rule $x \rightarrow x + 0$.

In general, given a term and a rewrite system, there may be several rules in the rewrite system which are applicable at the same time. A rewrite system that will always produce the same result regardless of the order of rule applications is called *confluent*.

Definition A rewrite system $\mathcal{R}$ is *confluent* iff for all s, t, t', whenever $s \overset{*}{\rightarrow} t$ and $s \overset{*}{\rightarrow} t'$ there exists a u such that $t \overset{*}{\rightarrow} u$ and $t' \overset{*}{\rightarrow} u$.

In our case we do not require confluence. Instead rules are applied in the order in which they are declared, which means that the rewrite system does not need to be confluent, or even terminating with an arbitrary rule application order. This means that for every rule, it is possible to take advantage of the fact that we can assume that none of the previous rules could be applied. As an example of how this can be used, take the following two rules which evaluate addition over constants, and reorders addition with constant and some other term:

$$\begin{array}{rcl} t_1 + t_2,\ \mathsf{const}(t_1),\ \mathsf{const}(t_2) & \rightarrow & \mathsf{eval}(t_1 + t_2) \\ t_1 + t_2,\ \mathsf{const}(t_2) & \rightarrow & t_2 + t_1 \end{array}$$

This rewrite system is clearly not terminating, since for a term $1 + 2$ the second rule could be applied infinitely many times. To achieve termination with an arbitrary rule application order, the second rule would have to be written as

$$t_1 + t_2,\ \neg\,\mathsf{const}(t_1),\ \mathsf{const}(t_2) \rightarrow t_2 + t_1$$

Example 4.3

If we define the following simple term rewrite system for additions

$$\begin{aligned} 0 + t &\rightarrow t \\ t_1 + t_2,\ \neg\,\mathsf{const}(t_1),\ \mathsf{const}(t_2) &\rightarrow t_2 + t_1 \\ t_1 + (t_2 + t_3),\ \neg\,\mathsf{const}(t_1),\ \mathsf{const}(t_2) &\rightarrow t_2 + (t_1 + t_3) \end{aligned}$$

Using this TRS we can rewrite the term $x+(y+2)$ by performing the following rewrites $x + (y + 2) \rightarrow x + (2 + y) \rightarrow 2 + (x + y)$. ▮

4.1.1 A simple rule language

In MathSAT, close to 300 rewrite rules have been defined. Implementing all these rules by hand can be a time-consuming and error prone process, and therefore a simple rule language have been developed which allows for easy definition of new rules, and reduces the risk of introducing errors. Two simple examples of rewrites for trivially unsatisfiable or valid atoms are the following

```
bvult(t, t) ---> false;
bvule(t, t) ---> true;
```

The language supports all bit-vector operators supported by MathSAT and the rewrite rule predicates and operators discussed earlier, and the bit-vector operators are named similarly to the names used in the SMT-LIB. There is also some syntactic sugar meant to make the writing of rules easier. As an example, identifiers starting with `c` are interpreted as constants. So the rule

```
bvadd(t, c) ---> bvadd(c, t);
```

would be equivalent to the rule

```
    bvadd(t1, t2), const(t2) ---> bvadd(t2, t1);
```

To achieve reasonable performance, memoization is used to cache the result of previous rule applications. In addition, some basic filtering on rules are done before checking whether they can be applied to a given term.

Currently, this normalisation language is not available to users. Instead, it is translated into C++ code at compile time and linked into MathSAT.

It may be that some interesting rules can not be expressed in the normalisation language in its current form. In these cases, they can be written by hand and added to the normalisation engine in the same way as generated rules. Another option would be to extend the rule language to support the necessary features. Since rules are generated at compile time, there is not yet any reason to have a rule language that supports any possible rule that may be interesting, and the choice between extending the language or implementing new rules which cannot be expressed in the rule language by hand becomes a pragmatic one.

4.1.2 Rule verification

It is easy to introduce erroneous normalisation rules for bit-vector arithmetic, mostly because of a natural tendency to think in terms of standard arithmetic over the integers. Take for instance the following simple example

Example 4.4

The rule

$$t_1 + t_2 <_{\mathrm{u}} t_3,\ \mathsf{const}(t_2),\ \mathsf{const}(t_3) \to t_1 <_{\mathrm{u}} t_3 - t_2$$

would be correct in ordinary linear arithmetic, but not in bit-vector arithmetic. For instance, consider the atom $v^{\langle 8\rangle} + 2^{\langle 8\rangle} <_{\mathrm{u}} 5^{\langle 8\rangle}$, which would be rewritten

using the rule into $v^{\langle 8\rangle} <_{\mathrm{u}} 3^{\langle 8\rangle}$ (assuming an additional rule for evaluation of the subtraction $5^{\langle 8\rangle} - 2^{\langle 8\rangle}$). But this atom is not equivalent to the original, we have for instance that $\{v = 255\} \models v^{\langle 8\rangle} + 2^{\langle 8\rangle} <_{\mathrm{u}} 5^{\langle 8\rangle}$ but $\{v = 255\} \not\models v^{\langle 8\rangle} <_{\mathrm{u}} 3^{\langle 8\rangle}$.
$\blacksquare$

Mistakes like these can be easy to make, and verification of the correctness of rules is therefore desirable. One possibility is to verify the correctness on the entire solver by conventional means, i.e. testing. But in this case it is possible to be more thorough; Since we have the rules defined in a simple rule language we can attempt for formally verify their correctness. Example 4.5 shows a simple example of verification of a basic rule.

Example 4.5

To verify the rule $t + 0 \longrightarrow t$, we can see that it is correct iff the formula

$$\forall n \in \mathbb{Z}^{+} \forall t \cdot t^{\langle n\rangle} + 0^{\langle n\rangle} = t^{\langle n\rangle}$$

is valid $\blacksquare$

As we can see from this example, rule verification requires reasoning on non-fixed width bit-vectors, which requires the use of a theorem prover with support for this theory. Several such theorem provers exist, such as Isabelle [Daw09] or PVS [BMS+96].

Since rules are frequently defined on non-fixed width bit-vectors, verification requires deciding validity of non-fixed width formulae, which is an undecidable problem. This means that verification may not be fully automatic, requiring user intervention. Another alternative is to use a SMT solver with support for fixed-width bit-vectors for verification. Since rules are defined on non-fixed sized bit-vectors, the width needs to be instantiated before generating verification conditions. Should some rule not hold for some particular width, a counter-example showing the error can be easily produced. Although this would not fully verify the correctness of rules, it is possible to verify the rules for all bit-vector width below some limit, and this should deliver high confidence in the correctness of the rules.

Example 4.6

The rule in example 4.4 can be translated into the verification condition

$$v_1^{\langle n\rangle} + v_2^{\langle n\rangle} <_{\mathrm{u}} v_3^{\langle n\rangle} \wedge \neg(v_1^{\langle n\rangle} <_{\mathrm{u}} v_3^{\langle n\rangle} - v_2^{\langle n\rangle})$$

Selecting a particular width, say $n = 4$ this formula is satisfiable e.g. with the model $\{v_1^{\langle 4\rangle} = 15^{\langle 4\rangle}, v_2^{\langle 4\rangle} = 1^{\langle 4\rangle}, v_3^{\langle 4\rangle} = 2^{\langle 4\rangle}\}$

The rule in example 4.5 above can be translated into the verification condition $v_2^{\langle n\rangle} = 0^{\langle n\rangle} \wedge \neg(v_1^{\langle n\rangle} + v_2^{\langle n\rangle}) = v_1^{\langle n\rangle}$. Regardless of the width n, this formula is unsatisfiable. ▮

4.1.3 Termination

Apart from rule correctness, another problem of the rewrite system is termination. With a potentially large number of rules, ensuring termination may become non-trivial. There are basically two possible scenarios for a non-terminating set of sequences; A cycle in the rewrites, and rewrites which increase the size of the terms indefinitely. Checking for cycles can be done by keeping track of each term in a sequence of rewrites and checking whether it recurs. Whether the rewrites will cause the term to grow indefinitely are not as easy to check, but by bounding the number of rewrites applied, it is possible to avoid this, even if it isn't possible to detect non-termination in this way.

Currently MathSAT does not attempt to prove termination, but instead it tracks all rewrites and discovers the particular case where rewrites are cyclic.

4.1.4 Implementation issues

The current implementation is straightforward, each rule is a function object[1]. Rules in the rules language are translated into C++ at compile time. Storing them as function object makes it easy to add collection of statistics for rule applications and bounds on rule applications at the level of individual rules. The framework keeps lists of all rules, and applies them bottom-up to each subterm/subformula to a fix-point. The only optimisations are division of the rules into sublists, one for each type of operator, and memoization of rule applications.

4.2 Substitution/variable elimination

It is very common for real-world formulae to contain a number of definitions together with a formula that uses these definitions.

$$\left(\bigwedge_i v_i = t_i\right) \wedge \phi$$

Although input languages like the SMT-LIB standard language has support for making such definitions using the `let` and `flet` constructs without introducing fresh variables, these definitions are often encoded as an equality between a fresh variable and the term $v_i = t_i$. For bit-vector formula these definitions would be encoded into SAT as comparators which are entirely unnecessary, and which could cause significant overhead in solving. So, instead, these should be removed in preprocessing, to hopefully generate a more compact and simpler formula to solve.

In addition, in MathSAT, there are a few other substitutions which are performed on formulae

[1]Also called *functors* in programming

– If there is a formula $v = t \wedge \phi$, where $v \notin \mathcal{V}ar(t)$, then it can be rewritten into the equisatisfiable formula $\phi[v \mapsto t]$.
– It there is a formula $v^{\langle n \rangle}[u : l] = t \wedge \phi$ where $v \notin \mathcal{V}ar(t)$, then it can be replaced with

$$\begin{array}{ll}
\phi[v \mapsto v_1^{\langle n-u \rangle} :: t :: v_2^{\langle l \rangle}] & \text{if } u < n \\
\phi[v \mapsto t :: v_2^{\langle l \rangle}] & \text{if } u+1 = n \\
\phi[v \mapsto v_1^{\langle n-u \rangle} :: t] & \text{if } l = 0 \\
\phi[v \mapsto t] & \text{if } l = 0 \wedge u+1 = n
\end{array}$$

where v_1, v_2 are fresh variables.
– If there is a formula $(P \equiv \phi) \wedge \psi$ where $P \notin \mathcal{P}reds(\phi)$ then it can be replaced with $\psi[P \mapsto \phi]$.
– If there is a formula $P \wedge \phi$, this is rewritten into $\phi[P \mapsto \top]$
– If there is a formula $\neg P \wedge \phi$, this is rewritten into $\phi[P \mapsto \bot]$

4.3 Combining normal forms and substitution

Normalisation can cause formulae stored as DAGs using perfect sharing to increase exponentially in size. This may happen in particular in combination with substitution, which tends to increase sharing of terms and subformulae. One possible remedy for this is to only allow normalisation rules that are guaranteed to not cause exponential blowup. Examples of such rule are those that simply evaluate terms or those which never introduces fresh terms.

That solution might not be palatable, since it forces us to give up any more powerful normalisation rules. An alternative is to bound the amount of normalisation that is performed. Several ways to accomplish this is necessary.

– Limit the number of times the normaliser is called

– Keep a bound on the size increase of the formula, and terminate when this bound has been reached.
– Bound the number of rule applications which are allowed

The last alternative can even be implemented at the rule level, bounding each individual rule to a certain maximal number of applications. In this way, those rules guaranteed to not cause blowup in the size of the formula can be used without a bound, bounding only those rules which may cause blowup.

In MathSAT, so far only the removal of more complicated rewrite rules is supported. This is because it is unclear how frequently this problem would occur in practise in real-world instances. Although considerable increase in the sizes of formulae have been observed, for real-world usage the resulting formula still appears to be either easily solvable despite the increased size, or no more difficult than it would be with a more conservative normalisation.

4.4 Propagation of unconstrained terms and formulae

Formulae often contain terms of formulae which are not relevant when determining satisfiability, but may still cause significant overhead when solving either due to their size or the complexity of the operators used. In these cases it can be advantageous to remove these irrelevant parts of the formula before attempting to solve it.

Example 4.7

In the formula $x + (3 * y) /_{\mathrm{u}} (z - 7) <_{\mathrm{u}} 5$, the term $(3 * y) /_{\mathrm{u}} (z - 7)$ is not relevant for determining whether or not it is satisfiable, assuming we are not interested in also computing a model. This is because regardless of the value of this term, it is possible to choose a value of x which gives any desired value for $x + (3 * y) /_{\mathrm{u}} (z - 7)$. This means that the addition could be replaced with a fresh variable v creating the formula $v <_{\mathrm{u}} 5$ which is equisatisfiable to the original but much easier to solve. ∎

The reason we could perform the simplification in example 4.7 is that the variable x was only used once in the formula, and we say in this case that the variable is *unconstrained*.

Definition Given a formula ϕ and a variable v, if v occurs only once in a DAG representation of ϕ with perfect sharing, then the variable is said to be *unconstrained* in ϕ.

Once we have found a number of unconstrained variables, we can propagate this "upwards" in the formula by checking if the terms where these formulae occur are also unconstrained.

Definition Given a term $t^{\langle n \rangle}$ containing a set of unconstrained variables $\mathcal{V}$, if for any interpretation μ not giving an interpretation to any of the variables in $\mathcal{V}$ it is the case that for all bit-vector constants $c^{\langle n \rangle}$ of width n $\mu \models t^{\langle n \rangle} = c$, then $t^{\langle n \rangle}$ is unconstrained.

What the definition says is simply that if we have a term with a number of unconstrained variables, this term is itself unconstrained iff for any value of the variables in the term which are not unconstrained, it is possible to choose values for the unconstrained variables so that the term evaluates to any value. In example 4.7, x is unconstrained, and for any bit-vector values c_1, c_2, it is possible to pick a value for x so that $x + c_1 = c_2$ is satisfiable. This means the addition is unconstrained.

Once all unconstrained terms are identified, it is possible to replace all of them with fresh variables to produce a new formula equisatisfiable to the original. For almost all of the bit-vector operations defined in our theory, it is under the right circumstances possible to propagate unconstrainedness

$::$	Both operands unconstrained
$[:]$	First operand unconstrained
not	Operand unconstrained
and, **or**	Both operands unconstrained
xor	At least one operand unconstrained
$\ll, \gg_{\mathrm{l}}, \gg_{\mathrm{a}}$	Both operands unconstrained
rol, **ror**	First operand unconstrained
$+, -$	At least one operand unconstrained
$*$	Both operands unconstrained, or one unconstrained and the other an odd constant
$/_{\mathrm{u}}, /_{\mathrm{s}}, \mathbf{rem}_{\mathrm{u}}, \mathbf{rem}_{\mathrm{s}}$	Both operands unconstrained
ite	At least two operands unconstrained
$=$	At least one operand unconstrained
$<_{\mathrm{u}}, <_{\mathrm{s}}, \leq_{\mathrm{u}}, \leq_{\mathrm{s}}$	Both operands unconstrained , or one operand unconstrained, the other a constant and the atom is not valid or unsatisfiable

Figure 4.1: Cases where propagation of unconstrained terms can occur

into the term. In figure 4.1, we detail the cases where this is so. Some basic cases like $x /_{\mathrm{u}} 1$, which can be trivially normalised are left out here, we will assume that normalisation has already been performed.

The case $c*x$ where c is odd may need some comment. The requirement for this to be possible is that for any d, $c * x = d$ must have a solution. This is not the case when c is even, for instance in the case $2*x = 1$ which lacks a solution. But for any bit-vector width n, it is the case that c and 2^n are relatively prime[2], and then the equation has a solution for any d.

Theorem 4.4.1 *For any bit-vector equation $a^{\langle n\rangle} * x^{\langle n\rangle} = b^{\langle n\rangle}$ where $a^{\langle n\rangle}$ is odd, the equation has a solution.*

This is a variant of a well known theorem from number theory stating that for any linear congruence $ax \equiv b \pmod{n}$, it has a solution if a and n are

[2]Two number are relatively prime iff they share no common positive factors except 1

relatively prime. This is clearly the case here, since any odd number is relatively prime to any power of 2.

Obviously, propagation of unconstrained terms generalises into propagation of unconstrained formulae as well. A propositional variable occurring only once in the formula is unconstrained, and it is possible to propagate unconstrainedness upwards. E.g., if in an equivalence $\alpha \Leftrightarrow \beta$ the formula α is unconstrained, the formula can be replaced with a fresh propositional variable. In a similar way it is possible to define propagation conditions for any logical connective.

4.5 Disjunctive partitioning

The *core fragment* of the bit-vector theory is a fragment of the bit-vector theory where the only allowed operators are concatenation $t_1 :: t_2$, selection $t[m : l]$ and equality $t_1 = t_2$. In this fragment, there is a well known technique presented by Cyrluk et al. in [CMR97, CMR96] that simplifies the formula into an equisatisfiable but smaller formula.

The core rewrite technique can in short be described as taking the set of equalities and reducing the widths of all bit-vectors by substituting each variable v with a concatenation of fresh variables $v \mapsto v_n :: \ldots :: v_1$ in such a way that in the resulting formula there are no selections in the variables $v_n, \ldots, v_1$ and each resulting equality can be split into a conjunction of equalities between variables $v_i = v'_j$. Every formula of this form is satisfiable iff there is a model such that all bits in each variable have the same value. Therefore, each such variable can be replaced with a single-bit variable to produce an equisatisfiable formula which uses a smaller total number of bits. This core rewrite technique chooses the division which minimises the number of bits in the resulting formula, and we say that in that case the variables are divided into *maximal chunks*.

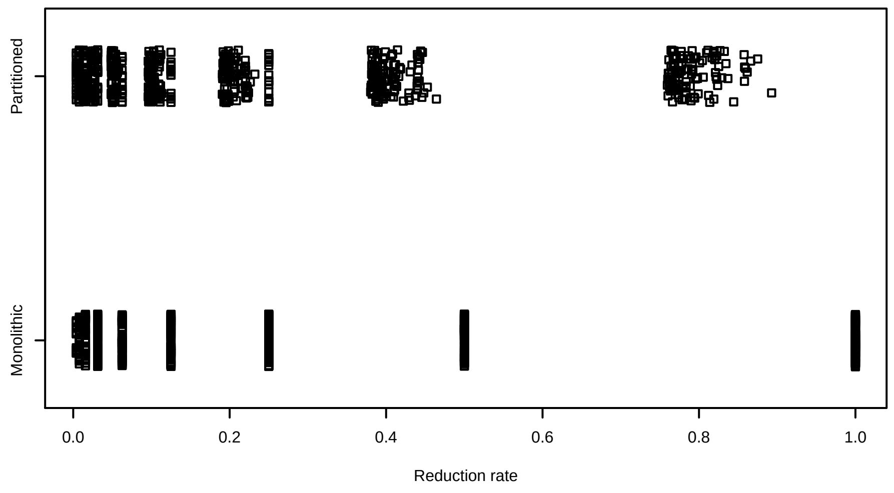


Figure 4.2: Core rewrite reduction

However, this technique is not always effective in reducing the width of the bit-vector variables. In the cases where the maximal chunks are all single bits, no reduction can be made. This is illustrated in some of the instances in the SMT-LIB in the core fragment contributed by Roberto Bruttomesso[3]. Measurements of the reduction rate measured in the reduction of the number of bits in all variables for these instances can be seen in the stripchart in figure 4.2, where "Monolithic" denotes the reduction rate achieved with the core rewrite technique. Although often a significant reduction is achieved, on several instance we see little or no reduction. Out of 672 instances, on 112 instances there is no reduction in formula size by applying the core rewrite technique. One cause of this is that the reduction is done globally in the entire formula. If instead we could partition the formula before applying the reduction in a way which allows us to apply the technique on each partition individually, the technique may be more effective on each partition. One way of producing a set of independent formulae is *disjunctive partitioning*. Here we take a single formula ϕ and

[3]Located in QF_BV/bruttomesso/core in the SMT-LIB

rewrite it into a disjunction of formulae

$$\phi = \bigvee_i \phi_i$$

We call this the *disjunctive partition*, and we can now solve each formula ϕ_i in isolation. The result of this partitioning on the core rewrite technique can be seen in figure 4.2 denoted by "Partitioned". The figure shows measurements on the reduction rate of the bit-vector formulae, is the ratio between the sum of the widths of all variables after and before applying the technique. Performing disjunctive partitioning seems to increase the efficacy of the technique, in particular the cases where no reduction could be achieved on the original formulae. Using disjunctive partitioning some reduction is always achieved on these formulae.

Disjunctive partitioning can also have a beneficial effect on other pre-processing techniques. Take for instance the following formula

$$t_1 - y \leq_{\mathrm{u}} x \vee t_2 + x >_{\mathrm{u}} t_3 + y$$

where x, y are variables and t_1, t_2, t_3 are terms where x, y does not occur. Performing disjunctive partitioning together with propagation of unconstrained terms can give us the disjuncts

$$P,\ Q$$

where P, Q are fresh predicates, and determining that the formula is satisfiable is now trivial regardless of the complexity of the terms t_1, t_2 and t_3.

4.6 Packet splitting

In some cases, we may find formulae where certain reasoning would be very cheap if only we had the right insight into the formula. One such case may be if we compute a number of bit-vector values, put them together in a “packet” by concatenating them together and then “transport” this packet to some other part of the formula through a chain of equalities where they are unpacked again. The fact that the individual bit-vectors in both ends of the chain of equalities are matching would be useful information to discover. If all these atoms are facts, we can perform substitution and normalisation to discover this, but if they are not, we are forced to discover this during search.

In general, if we have 4 terms $t_1^{\langle m\rangle}$, $t_2^{\langle n\rangle}$, $t_3^{\langle m\rangle}$, $t_4^{\langle n\rangle}$, a set of variables $v_1, v_2, \ldots, v_N$ then if we have the following atoms

$$\begin{aligned}
&t_1^{\langle m\rangle} :: t_2^{\langle n\rangle} = v_1 \\
&v_1 = v_2 \\
&v_2 = v_3 \\
&\ldots \\
&v_{N-1} = v_N \\
&v_N = t_3^{\langle m\rangle} :: t_4^{\langle n\rangle}
\end{aligned}$$

Then for each variable v_i we can create two fresh variables $a_i^{\langle n\rangle}, b_i^{\langle m\rangle}$ and apply the substitution $v_i \mapsto a_i^{\langle n\rangle} :: b_i^{\langle m\rangle}$ to the formula. This will have the effect of splitting the equalities into two, which will enhance the capability of the EUF solver to discover inconsistencies and deductions.

Packet splitting will have no effect on bit-blasting, the resulting CNF for each atom will be identical. So this technique should have no adverse effect on the bit-vector solver, while helping us discover more conflicts and deductions using EUF layering.

4.7 Difference propagation

In section 4.2, we saw how we can propagate information from equalities by substitution. If in a formula ϕ it is known that $v = t$ where v is a variable and t is some term not containing v we can substitute v for t in ϕ. But if in the formula it is known that $t_1 \neq t_2$, is it possible to do something similar? To start with, it is easy to realise that if we have that $v_1 \neq v_2 \wedge v_2 = v_3$ we can deduce that $v_1 \neq v_3$ and in the same way if we know $v_1 \neq v_2$ and our formula ϕ contains the subformula $v_2 = v_3$, we can deduce that $v_2 = v_3 \Rightarrow v_1 \neq v_3$ and conjunct that to our formula. This propagation can be continued through further equalities. If $v_3 = v_4$ also exist in the formula, we can add $v_2 = v_3 \wedge v_3 = v_4 \Rightarrow v_1 \neq v_4$.

However, this does not appear to be very useful in a solver which uses EUF layering. In such a solver, these facts will be easily deduced during search. So we should probably look further to gain some advantage of propagating differences in preprocessing. One more promising case where EUF is unable to perform the deduction is for injective functions.

Definition Let f be a function with the domain A and codomain B. Then this function is *injective* iff whenever $f(x) = f(y)$ then $x = y$.

A straightforward consequence of the definition is that for any injective function f, whenever $x \neq y$ we have that $f(x) \neq f(y)$.

Example 4.8

If we have the formula

$$a^{\langle 16\rangle} \neq b^{\langle 16\rangle} \wedge a^{\langle 16\rangle}[7:0] :: a^{\langle 16\rangle}[15:8] = c \wedge c = b^{\langle 16\rangle}[7:0] :: b^{\langle 16\rangle}[15:8]$$

We have that $a \neq b$, and by realising that $f(x) = x[7:0] :: x[15:8]$ is an injective function, we can deduce that $a^{\langle 16\rangle}[7:0] :: a^{\langle 16\rangle}[15:8] \neq b^{\langle 16\rangle}[7:0] :: b^{\langle 16\rangle}[15:8]$. This means that $c \neq c$, and so the formula is unsatisfiable. ∎

In general, if we have a formula ϕ containing two variables x, y, an equality $x = y$ and an equality between two terms $t_1 = t_2$ containing only the variables x and y respectively such that $t_1 = f(x)$ and $t_2 = f(y)$ where f

is some injective function, then we can deduce that $\neg(x = y) \Rightarrow \neg(f(x) = f(y))$ is a valid formula. This means we can add this axiom to the formula, forming

$$\phi \wedge (\neg x = y \Rightarrow \neg(f(x) = f(y)))$$

In example 4.8, this would make this original formula

$$\begin{aligned} & a \neq b \wedge a^{\langle 16\rangle}[7:0] :: a^{\langle 16\rangle}[15:8] = c \wedge c = b^{\langle 16\rangle}[7:0] :: b^{\langle 16\rangle}[15:8] \\ \wedge \quad & (a \neq b \Rightarrow a^{\langle 16\rangle}[7:0] :: a^{\langle 16\rangle}[15:8] \neq b^{\langle 16\rangle}[7:0] :: b^{\langle 16\rangle}[15:8]) \end{aligned}$$

Which can be found to be unsatisfiable using the EUF solver. To see why we can look at the formula as if it is a formula in EUF:

$$a \neq b \wedge f(a) = c \wedge c = f(b) \wedge (a \neq b \Rightarrow f(a) \neq f(b))$$

where f is the injective function over a and b respectively.

There may be many occurrences of the same injective function $f(x)$ modulo variable renaming in a formula, once one injective function has been located it is therefore useful to locate other usage of the same function by simply checking if there is another term in the formula identical modulo variable renaming.

To check that a function $f(x)$ is injective we can generate the verification condition

$$x \neq y \wedge f(x) = f(y)$$

which is unsatisfiable iff the function is injective. This verification condition could be checked in the solver before attempting to solve the formula.

4.8 Other techniques

Several other minor preprocessing techniques are available in MathSAT, some of them are listed here.

Pure literal elimination The pure literal rule is a well known technique in SAT solving. If in a formula in CNF an atom only occurs positively or only negatively it is said to be pure. This variable can then be assigned to a value which satisfies all clauses where it occurs, removing those clauses from the formula. In MathSAT, this can also be used to filter out theory literals which can be ignored by the theory solvers. It is also possible to replace pure propositional atoms with truth values in preprocessing. This step may in turn cause further simplifications by other techniques.

ITE merging Sometimes, formulae may contain multiple conjuncts of the form $\bigwedge_i \mathrm{ite}(\phi, \alpha_i, \beta_i)$ which can be merged into $\mathrm{ite}(\phi, \bigwedge_i \alpha_i, \bigwedge_i \beta_i)$.

4.9 Model computation

Computation of models for satisfiable formulae is an important feature in real-world applications, and preprocessing steps should be compatible with this feature if possible. When that is not possible, one alternative approach to model computation is to simply disable any technique which makes computing models non-trivial. This has two disadvantages:

– The techniques not compatible with model computation lose some value, since models are frequently required in real-world applications
– Disabling some techniques will also affect performance on unsatisfiable formulae

Here, we will make some effort in avoiding this drastic measure, and instead attempt to discover ways of computing models for all preprocessing techniques described in this thesis.

The basic problem can be stated as follows. We have an original formula ϕ which through some preprocessing technique has been rewritten into a formula ϕ'. Assuming we we find a model $\mu \models \phi'$, how can we compute a model $\mu' \models \phi$?

For normalisation as described in section 4.1, the resulting formula is always equivalent to the original, so any model for the normalised formula will also be a model for the original formula. The other techniques produce an equisatisfiable and generally not equivalent formula, they may both remove variables and introduce fresh variables in the formula. Using substitution, variables are removed from the formula by applying some substitution $v \mapsto t$, but in this case computing a model is easy since we know that $v = t$ in the original formula. Given a interpretation for the variables in t we can simply evaluate the term t to compute an interpretation for v so that $\mu \models v = t$.

When applying the core rewrite technique of Cyrluk et al. we know that the original variables relate to their replacements by concatenating the replacements together.

With packet splitting, some variable v is substituted by a concatenation of fresh variables $v_1 :: v_2$. Given an interpretation μ for the fresh variables, we can compute an interpretation for v such that $\mu' \models v = v_1 :: v_2$.

When propagating unconstrained terms, we replace some term t with a fresh variable v because we know that t contains some variable or variables that do not occur elsewhere in the formula such that for any value of v we can find a value for these variable satisfying $t = v$.

To sum up, all techniques that are used can be described as simply applying some substitution $\phi' = \phi[t \mapsto t']$, and we can use these substitutions to help us compute models for satisfiable formulae.

One very simple solution for model computation is to keep track of all substitutions being applied to the formula, and given a model for the simplified formula compute a model for the original formula as well using these substitutions. One simple approach to compute a model is to use a SMT solver to help us. Given a model μ and a substitution $t \mapsto t'$ which was used to rewrite the original formula ϕ into ϕ', we can extend the model by computing a model for the formula $\mu \wedge t = t'$. Working our way backwards through all rewrites performed during preprocessing, we can in this way accumulate a model for the original formula.

Example 4.9

If we have the formula $x + 1 < y \wedge y > z \wedge z = 3$, we can apply the following preprocessing techniques: First we apply substitution on z, and then propagation of unconstrained terms on x, keeping track of all rewrites we perform. We see the rewrites and the resulting formula below:

Formula	*Substitutions*
$x + 1 <_{\mathrm{u}} y \wedge y >_{\mathrm{u}} z \wedge z = 3$	
$x + 1 <_{\mathrm{u}} y \wedge y >_{\mathrm{u}} 3$	$z \mapsto 3$
$v_1 <_{\mathrm{u}} y \wedge y >_{\mathrm{u}} 3$	$z \mapsto 3,\ x + 1 \mapsto v_1$

Let's say the solver produces a model $\mu = \{v_1 = 3, y = 4\}$ for the preprocessed formula. We extend this model iteratively using the rewrites which have been performed in reverse order.

1. We can compute a value for x by solving $v_1 = 3 \wedge y = 4 \wedge x + 1 = v_1$ giving us an extended model $\mu' = \{v_1 = 3, y = 4, x = 2\}$.
2. Finally we can compute a value for z by solving the formula $v_1 = 3 \wedge y = 4 \wedge x + 1 = v_1 \wedge z = 3$ giving us the extended model $\mu' = \{v_1 = 3, y = 4, x = 2, z = 3\}$.

This gives us a final model $\mu''' = \{y = 4, x = 2, z = 3\}$, keeping only the variables in the model which occurred in the original formula. ▮

We will call this approach the *incremental* model computation approach, since it builds the model for the original formula incrementally one substitution at a time. An alternative might be a *monolithic* approach, trying to compute the model in one step.

Naturally, there are several obvious improvements possible on the basic algorithm. In the last model computation step in the above example, there is no need to use a SMT solver to compute a value for z, the rewrite rule already gives us the value. In the same way for the rewrite rule $x+1 \mapsto v_1$ we can replace all known values for the variables giving us the equality $x+1 = 3$. Now we can apply normalisation described in section 4.1 to yield the equality $x = 2$.

The main disadvantage of this technique becomes obvious with a simple example. Consider the formula

$$x /_{\mathrm{u}} y <_{\mathrm{u}} z$$

which can be rewritten using propagation of unconstrained terms into $v <_{\mathrm{u}} z$ where v is a fresh variable, which in turn can be rewritten into the fresh predicate P. This removes the potentially difficult to reason with division operator making the formula trivial to solve. However, when computing a model for this formula using the technique described above, it is necessary to solve a formula which includes the division operator, and so we have not gained much. A solution to this problem is to store information of why the rewrite was applied, and use an *ad hoc* model computation procedure for each particular type of rewrite. In this example, the rewrites applied (if split two separate rewrites for clarity) is $x /_{\mathrm{u}} y \mapsto v, v <_{\mathrm{u}} z \mapsto P$. During model computation, we have the initial model $\{P = \top\}$, and the first model computation condition becomes $v <_{\mathrm{u}} z$ after simplification. Since we know that this was a propagation of unconstrained terms where both operands were unconstrained, we can choose any value for v and z that satisfies the atom. Let's say we choose $\{v = 0, z = 1\}$. The second model computation condition now becomes $x /_{\mathrm{u}} y = 0$ after simplification.

The reason for the rewrite was that both operands of the division were unconstrained so we are free to pick suitable values for them. Let's pick $\{x = 0, y = 1\}$. The final model therefore becomes $\{x = 0, y = 1, z = 1\}$. By realising why each rewrite of the formula was performed, we were able to compute a model without having to perform any complex reasoning.

For every type of rewrite, it is possible to devise an ad hoc procedure which solves the model computation condition efficiently. In the case of most rewrites such as those for substitution, pure literal elimination or the core rewriting technique, simple evaluation is enough to solve them. For propagation of unconstrained terms, we will show how they can be solved in some of the cases, the others are analogous.

$v + t \mapsto v'$ After first assigning arbitrary values to any unassigned variable in t, the model computation condition will become $v + c_1 = c_2$ which trivially simplifies into $v = c_2 - c_1$.

$v * c \mapsto v,\ c$ is odd The model computation condition is $v * c = c'$, which can be solved by noticing that this is the same as the solving the problem $ax \equiv b \pmod{n}$ and can be solved with standard methods by computing the least residual of $xb \mod n$. More details can be found in number theory textbooks, e.g. Yan [Yan02].

$v_1 /_{\mathrm{u}} v_2 \mapsto v$ Both operands are unconstrained, so we can choose $v_2 = 1$ and $v_1 = v$ as a solution.

$v_1 <_{\mathrm{u}} v_2 \mapsto p$ If p evaluates to true, we can pick $\{v_1 = 0, v_2 = 1\}$. Otherwise we can pick $\{v_1 = 0, v_2 = 0\}$.

Example 4.10

Given the formula $x = y + 2$, this can be simplified in several ways. We can realise that x is unconstrained and replace the equality with a fresh predicate, or we can apply the substitution $x \mapsto y + 2$. In the second case, the resulting formula $y + 2 = y + 2$ simplifies into $\top$. When computing a model we now have the model computation condition $x = y + 2$ to solve. With the ad hoc method we need to realise that since this was the result of a substitution, we can give any values to the variables in the right hand side, and then evaluate it to get a value for x. E.g. if we assign $y = 0$, we get the model $\{x = 2, y = 0\}$.
▮

4.10 Incrementality and backtrackability

To be useful in the MathSAT API, a preprocessing technique needs to be both incremental and backtrackable. For some techniques, these features are trivially supported. The local simplifications described in section 4.1 act locally on each subterm and subformula and naturally support both incrementality and backtrackability.

Substitution can also support both features with some modifications. One can either apply substitution locally on each asserted formula, or accumulate the substitutions found for each asserted formula and apply these also for every future asserted formula. Applying substitutions globally on all asserted formulae can cause problems with efficiency, since it may modify previously asserted formulae, and incrementality would be more difficult to achieve. But even just applying substitutions locally is problematic. If we first assert the formula

$$x > 3$$

and then assert the formula

$$x = 1$$

we can not simply perform the substitution $[x \mapsto 1]$ on the second formula, which would transform it into $\top$. This would make the conjunction of the two formulae satisfiable, rather than unsatisfiable. Instead, when discovering a new substitution, we must be careful to apply it without deleting the equality that was used to create it, and only apply it on the rest of the formula. So, for any substitution which eliminates a variable, the equality used to create the substitution must be preserved iff this variable has been asserted previously. When backtracking, this list of substitutions can be reset to the state it had at the last backtracking point.

For propagation of unconstrained terms, the situation is a little more complex. Whether a term is unconstrained or not is a global property depending on all asserted formulae. This means that a term t which is unconstrained after asserting a number of formulae may cease to be unconstrained in the future as further formulae are asserted. When that happens the original definition of this term must be inserted in the set of assertions again. This can be done by keeping track of the substitution which was performed $[t \mapsto v]$ where v was a fresh variable, and when a new formula is asserted which makes t no longer unconstrained, we can simply add the definition of the fresh variable $v = t$ to the set of asserted formulae.

Some of the other techniques also cause problems, such as disjunctive partitioning and the core rewrite technique. This limitation makes these techniques less useful in real-world applications, unless the application itself guarantees non-incremental usage. As we shall see in chapter 7 however, even simply solving several similar formulae benefits from an incremental solver, making incrementality a useful feature even in some non-incremental applications.

It should be noted that the necessary implementation work to achieve incrementality and backtrackability has not yet been performed in MathSAT.

4.11 Architecture

The bit-vector solver is still in an experimental stage, so the design goals are focused on configurability and simplicity rather than achieving the best possible performance. To achieve this, each preprocessor technique is a separate step, rather than interleaving steps into each other. The preprocessing is performed until none of these steps changes the formula.

Chapter 5

Approximation of formulae

Not all tranformations need be satisfiability preserving as was the case with the preprocessing techniques discussed in chapter 4. There is an important class of techniques, which are not. This class is often called *approximations*. An approximation has the characteristic that one of the possible results (satisfiable or unsatisfiable) are correct, but the other may not be.

Example 5.1

We have the formula

$$\phi \vee \psi$$

We can abstract this formula into ϕ. If this abstraction is satisfiable, we know that the original formula was also satisfiable. However, if ϕ is unsatisfiable, we still do not know whether the original formula is satisfiable or not. ▮

Approximation are commonly divided into two types: *under-approximations* and *over-approximations*.

Definition Given a formula ϕ an *under-approximation* $\underline{\phi}$ is a formula such that if $\underline{\phi}$ is satisfiable, so is ϕ.

Definition Given a formula ϕ an *over-approximation* $\overline{\phi}$ is a formula such that if $\overline{\phi}$ is unsatisfiable, so is ϕ.

The approximation used in example 5.1 is an example of an under-approximation. An example of an over-approximation can be found in example 5.2.

Example 5.2

If we have the formula $\phi \wedge \psi$ it is possible to approximate it with the formula ϕ. Should this formula be unsatisfiable, we know this holds also for the original formula, and therefore this is an over-approximation. ∎

The motivation for approximation techniques is that it is assumed that some parts of the formula are not relevant for demonstrating whether or not it is satisfiable, but there is no (simple) satisfiability-preserving rewriting technique able to simplify the formula in a way that eliminates that part of the formula. But as long as there exists a technique that computes an under- or over-approximation which does remove these difficult to reason with subformulae or subterms, we can still apply those to simplify the formula.

The downside is that if we make the incorrect choice of abstraction, the result from the solver does not tell us if the formula was satisfiable or not. In this case, we need to *refine* the approximation. A refinement produces a new formula, which hopefully has a better chance of producing the desirable result. Solving and refinement are performed iteratively, in the same way as abstraction refinement loops are used in verification. The general algorithm in found in algorithm 5.1. It starts by generating an initial approximation or the original formula, which is meant to be the coarsest approximation that will be tried. This formula is solved, and if the result is *admissible*, it is returned. An admissible result is one which

Algorithm 5.1: Approximation/refinement $\text{Solve}_{\text{AR}}(\phi)$

```
1 $\phi' \leftarrow$ initial approximation of $\phi$
2 **while** Solve($\phi'$) *not admissible* **do**
3     $\phi' \leftarrow$ refine($\phi'$)
4 **end**
5 **return** Solve($\phi'$)
```

holds also for the original formula, e.g., if the approximation was an under-approximation and the approximated formula was satisfiable, this result is admissible. If the result was not admissible, it is refined, which produces either a new formula and the algorithm iterates in this loop until an admissible result is produced. For termination there are two requirements on the components of this algorithm

1. If ϕ' is identical to ϕ, the result of Solve(ϕ') is always admissible.
2. Eventually, the refinement step will produce the original formula ϕ.

Example 5.3

Consider the formula

$$\bigvee_i \phi_i$$

Applying under-approximation, we can produce an initial approximation as ϕ_1. The result of solving the formula is then that either the approximation is satisfiable, or it is identical to the original formula. In the refinement step we can simply add one more disjunct ϕ_i to the approximation. Either we will be able to show the formula satisfiable with only a subset of disjuncts, or we will (eventually) solve the original formula. ∎

In this chapter, we will show three different approximations; One providing over-approximation for use in the lazy SMT schema, and two under-approximation techniques for use in preprocessing and in the theory solver respectively. We will also show how these can be trivially combined in a decision procedure by nesting of approximation refinement loops, and give some implementation details used in implementing these techniques in MathSAT.

Naturally, as with the preprocessing steps discussed in chapter 4 it is advantageous if the approximation techniques do not interfere with the ability to produce models of satisfiable formulae, so we will show how models can be computed when under-approximations are used. For the over-approximation technique discussed here, model computation is not an issue.

5.1 Over-approximation

Since over-approximation techniques do not require refinement when the approximation is unsatisfiable, it makes sense to attempt to apply over-approximation on cases where the formula at hand is believed to be unsatisfiable. One natural candidate is in a theory solver as used in the lazy SMT schema. In this case, the formula to be solved is a set of theory literals. $l_1, \ldots, l_N$. If for complete Boolean models the set is inconsistent (the conjunction of all literals $l_1 \wedge \cdots \wedge l_N$ is unsatisfiable, we compute a theory conflict set, perform conflict analysis in the boolean enumerator, backtrack, and continue searching. If it is consistent (the conjunction is satisfiable) we have found a model. That means that we can expect formulae to be unsatisfiable, and the satisfiable case is the exception. For this reason, it seems to be an ideal candidate for over-approximation.

In most cases, when a particular truth assignment is inconsistent, it is possible to compute a conflict set which is very small in relation to the number of literals on the truth assignment. Naturally, if we could somehow identify this subset from the start, a good over-approximation would be this exact subset. But this may not be possible in practise. It is also not unusual that it is possible to find an inconsistency that is "obvious", in the sense that it does not require complex reasoning to discover it.

We can also observe that some operators are in general more difficult to reason with than others, e.g., multiplication can be more difficult than addition or bit-wise operators. A simple strategy is therefore to initially only consider literals containing "simple" operators. Only if no inconsistency is detected among those literals are potentially more difficult-to-reason-with literals considered.

We choose the initial approximation as the empty set of literals. It may seem strange to choose an approximation that is guaranteed to not deliver an admissible answer, the reason for this choice is that EUF abstraction should be given an opportunity to show the formula unsatisfiable by itself when it is used. In cases where the bit-vector theory solver is called, the initial consistency check with an empty set of literals is cheap enough to not matter.

5.1.1 Refinement

Refining the over-approximation is straightforward, simply take more literals into account. There are many choices for how this could be done, and since the intuitive idea is to reason with "simple" atoms first the first step is to understand which are simple and which are not. First for all atoms a *penalty* is computed by accumulating penalties for all terms in each atoms as given by table 5.1. The penalties chosen in this work are meant to convey that some operators are potentially more difficult to reason with than other, but this is still only a very rough approximation of difficulty.

Table 5.1: Operator penalties

Operator	Penalty
Constants	0
Concatenation, selection	0
Sign/zero extension	0
Variable	1
Bit-wise operators	1
Rotation	1
Addition, subtraction	100
Shift	100
Multiplication, division	1000

Example 5.4

The atom $(x^{\langle 16\rangle} :: y^{\langle 16\rangle}) + 3^{\langle 32\rangle} <_{\mathrm{u}} \mathrm{zext}^{\langle 32\rangle}(x))$ has two variables (penalty $= 2$), a zero extension (penalty $= 0$), a concatenation (penalty $= 0$) and an addition (penalty $= 100$) giving a total penalty for the atom of 102. ∎

During refinement, we add a few more of the atoms with the lowest penalty which have not yet been added to the solver. To decide how many is "a few", all atoms are divided into *tiers* based on difficulty. The refinement procedure checks the current truth assignment, and locates the lowest tier containing atoms on the truth assignment which have not yet been added to the solver and adds all those atoms in that tier which occur in the truth assignment. In MathSAT, atoms have been somewhat arbitrarily divided into 4 tiers, with penalties in the intervals $[0, 100]$, $[101, 1000]$, $[1001, 10000]$ and lastly $[10001, \infty]$

Early termination An improvement of the basic refinement algorithm is to filter out those atoms occurring on the truth assignment for which the current theory solver model gives the correct truth value. Using this improvement it may be possible to discover that all literals on the truth assignment are satisfied and terminate the approximation/refinement loop early.

This is however not yet implemented in MathSAT.

5.2 Under-approximation

For under-approximation, we need to find another suitable case where it might be helpful. It is well known that arithmetic may be very difficult to reason with for SAT-based tools. But with automatically generated formulae coming from formal verification, it may well be that much of these arithmetic terms are not relevant for satisfiability.

So the idea here is to abstract away arithmetic in the hope that it is irrelevant. There are a multitude of possible ways of doing this, here we will iteratively approximate the formula, starting with those terms which seem to be most difficult to reason with, multiplications and divisions, and then moving on to additions/subtraction and so on.

The basic idea is to simply guess a value for a variable occurring in a term, in such a way that the terms becomes easier to deal with.

Example 5.5

Given the formula

$$a + b /_{\mathrm{u}} c <_{\mathrm{u}} 3 \wedge b * c = 2$$

If we were to guess that $c = 1$ the formula simplifies to

$$a + b <_{\mathrm{u}} 3 \wedge b = 2$$

which simplifies further to

$$a + 2 <_{\mathrm{u}} 3$$

which can be easily solved, e.g. with $a = 0$ ∎

In this work, we build on this example. We first rank operators according to "complexity": ranking division, remainder, multiplication higher that addition or subtraction, which in turn is ranked higher than bit-wise operators. Then we locate all variables occurring as operands in operators. Starting with the variables occurring in higher ranked operators, values are guessed for these variable in such a way as to remove the operator from the formula. E.g., for a divisor, 1 is a suitable value, for an addend 0 may be suitable.

In MathSAT these under-approximations are performed iteratively. First a candidate variable is identified, and it is replaced with a suitable value. Then all preprocessing steps that have been enabled by the user is performed on this approximation. This is done iteratively until either the formula simplifies to $\top$ or $\bot$, no more candidate variables can be found, or an upper limit on the number of iterations is reached. There are two reasons to perform the preprocessing steps after each under-approximation step. First, the under-approximation may generate more opportunities for simplification of the formula, and second, this increases the chance that we can detect that the current under-approximation is unsatisfiable without needing to solve the formula. This may reduce the number of unnecessary under-approximations significantly. The result in each iteration is stored on a stack of under-approximations, and this stack is initialised by pushing the original formula onto the stack. The current approximation is the top of this stack.

5.2.1 Refinement

Since during approximation one variable was assigned a variable at a time, and each intermediate formula was stored in a stack, refinement is as simple as removing the top of the stack and using the next formula. If the stack becomes empty, the original formula was unsatisfiable.

Since preprocessing has been performed on each formula in the stack, changes from one formula to the next can be more than simply replacing a variable with a constant. In the current implementation, each refinement is therefore solved in a separate solver rather than using a single solver incrementally. After solving each formula the theory conflict clauses are collected, and all clauses which are relevant are added to the new solver before solving. A relevant clause is one whose atoms occur in the current formula. In this way, at least some of the information gained in a previous iteration can be reused.

Early termination Since the under-approximation trigger further rewrites in preprocessing, early termination is not straight-forward. However, if the theory conflicts in a particular iteration would be enough to show unsatisfiability in the propositional abstraction of the original formula, then the solver will be able to detect this without making any call to the bit-vector theory solver due to the over-approximation in the theory solver which initially ignores all bit-vector atoms.

5.2.2 Under-approximation in theory solver

In the theory solver, a similar under-approximation of assigning variables to values technique can also be applied. A simple way of doing this is to add extra assumption to the SAT solver which describes the under-approximation that should be attempted. If we wish to approximate by guessing that some variable $v^{\langle n\rangle}$ is 0, and this variable has been bit-blasted into $(v_{n-1}, \ldots, v_0)$, we assume the literals $\neg v_{n-1}, \ldots, \neg_0$ in the SAT solver within the theory solver.

During refinement of this type of approximation, it is possible to check whether the approximation itself is part of the cause for unsatisfiability, or if the non-approximated set of literals would have been unsatisfiable as well. This can be done by checking either the unsatisfiable core of the formula in the sat solver, or as is done here by computing a top-level conflict in terms of the assumptions. If this conflict does not contain any of the assumptions that are part of the approximation, we can deduce that the problem was unsatisfiable. Otherwise, we can refine the approximation by removing all assumptions in the conflict that were part of the approximation, and solve again.

5.2.3 Model computation

Since both types of under-approximation assigns concrete values to variables (or bits of variables), computing models for a formula given a model for an under-approximated formula is trivial, we can simply extend the model with the values provided in the approximation step.

For the under-approximation in the bit-vector theory solver, this comes "for free" since the approximation is based on adding extra assumptions to the underlying SAT solver. Any model returned from the SAT solver can therefore be treated the same regardless of whether under-approximation was used.

For the under-approximation in preprocessing, the under-approximations of a variable v to a value c can be recorded as any other preprocessing rewrite $v \mapsto c$. The model computation algorithms described in section 4.9 will then work without modification. The only complication comes in refinement, when an approximation is refined some rewrites must be removed. This can be easily solved by keeping the rewrites in a stack and simply restore the stack a the state it was in before the last under-approximation was performed.

5.3 Combining multiple approximation/refinement loops

It is easy to see that under- and over-approximation can be combined, as long as the approximation/refinement loops are nested. Each approximation/refinement loop will take a formula as input, approximate and check the result. If the result is correct with respect to the approximation it can be returned, otherwise the approximation is refined and we solve again. This means that if the solver used is a decision procedure, then the approximation/refinement loop will also implement a decision procedure. Noticing this, it is easy to see that approximation/refinement loops can be nested by replacing the function used to solve the formula in each iteration with a function that implements an approximation/refinement loop. A

Algorithm 5.2: Approximation/refinement $\text{Solve}_{\text{AR}}(S, \phi)$

```
1 $\phi' \leftarrow$ initial approximation of $\phi$
2 while $S(\phi')$ not admissible do
3     $\phi' \leftarrow \text{refine}(\phi')$
4 end
5 return $S(\phi')$
```

sketch of this algorithm is shown in algorithm 5.2. This algorithm takes two inputs, the formula ϕ to solve and a decision procedure S which can solve formulae. This procedure can either be a normal decision procedure, or itself implement an approximation/refinement loop.

With early termination in the inner loop, the outer loop will also terminate as long as the approximations are of opposing types (under-approximation nested inside over-approximation, or vice versa). This is because an early termination result in this case will always be admissible in the outer loop, and that loop will also terminate. If two approximation loops of the same type are nested, an early termination of the inner loop will not automatically terminate the outer loop, it will only terminate if the result is admissible also in the outer loop.

This is how the under- and over-approximations in the bit-vector theory solver are combined. The under-approximation which assumes values for particular bits is nested inside the over-approximation which only considers a subset of the literals on the truth assignment. If the under-approximation loop terminates early, so will the over-approximation loop.

Chapter 6

Experimental evaluation

In this chapter we try to discover the effectiveness of the various techniques discussed in this thesis. There are several questions we would like to answer:

– Are all techniques useful?
– Is there some interaction between several different techniques, or are they independent of each other?
– Which techniques should be used, and which should not?
– What is the effect of model computation on execution time?

Even if we can't deliver a final answer to these questions, we will still make an attempt to give at least a partial answer of how well these techniques work in MathSAT. With the large number of techniques discussed, we will focus on an interesting subset of them, and provide only a brief overview of the efficiency of the rest.

All experiments in this chapter were carried out on machines with dual Intel Xeon E5430 CPUs running at 2.66 GHz using 16 GB of RAM running Linux.

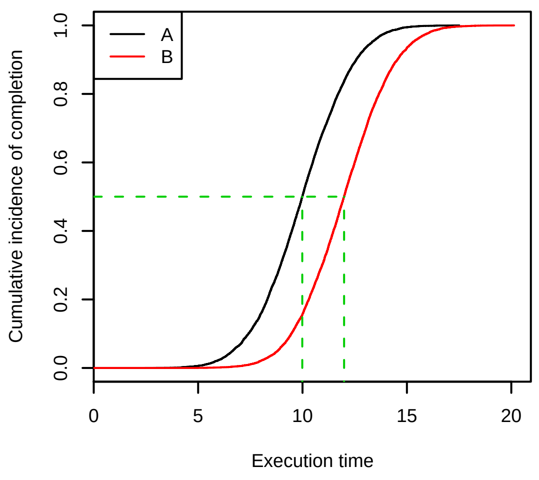


Figure 6.1: Example Cumulative Distribution Function plot

6.1 Cumulative distribution functions

Before we look at the experimental data, we introduce a graphical tool which can be used to provide an insight into the behaviour of an algorithm. A very useful type of plot is a Cumulative Distribution Function (CDF) plot, and they can be seen as a type of (Kaplan-Meier) survival plots. The CDF for some variable X can be defined as the probability that the variable is less than some value.

$$F(x) = P(X < x)$$

An example of two CDFs can be seen in figure 6.1. Here we have measured the execution time of two hypothetical algorithms on a set of instances, and computed the cumulative distribution function of the execution time. One advantage of CDF plots is that we can read off every percentile of the measured quantity in this type of plot. In this case we have marked the median (the 50th percentile), which is $F(x) = 0.5$ on the vertical axis. We can see that the median is around 10 for algorithm A, and around 12 for algorithm B. Looking at other percentiles, we can see that algorithm A is always better than B, and in this case we say that A *dominates* B and this is strong evidence that A outperforms B in general assuming the experiment was performed on a representative set of instances. Apart from being able to give an overview of the performance of an algorithm at a glance, it can also be used to compare more than two different algorithms, which is cumbersome with the more traditional scatter plots.

6.2 Effects of techniques

In this section we try to discover the effect of some of the techniques that have been presented in this thesis. We do this on a random selection of 100 real-world instances from the SMT-LIB. Since the instances from the SAGE tool [GLM07] outnumber all other instances by a wide margin, we first divided the instances into subsets and sampled these. We selected 20 instances from SAGE, 20 from Spear/Calysto, 20 from UCLID or related tools, and 40 from the other sources. On these we ran several tests with all combinations of the following techniques

– Encoding into SAT versus DPLL(T)
– Under-approximation in preprocessing
– Normalisation of bit-vector terms
– Substitution
– Propagation of unconstrained terms
– Pure literal elimination in preprocessing

A timeout of 600 second was used to keep computation time reasonable. The results clearly showed that three of the techniques have a significant impact on performance: Under-approximation, normalisation and substitution. An overview on the results can be seen in figure 6.2. Both normalisation and substitution does improve performance, while using under-approximation increases execution time. Using SAT appears to be an im-

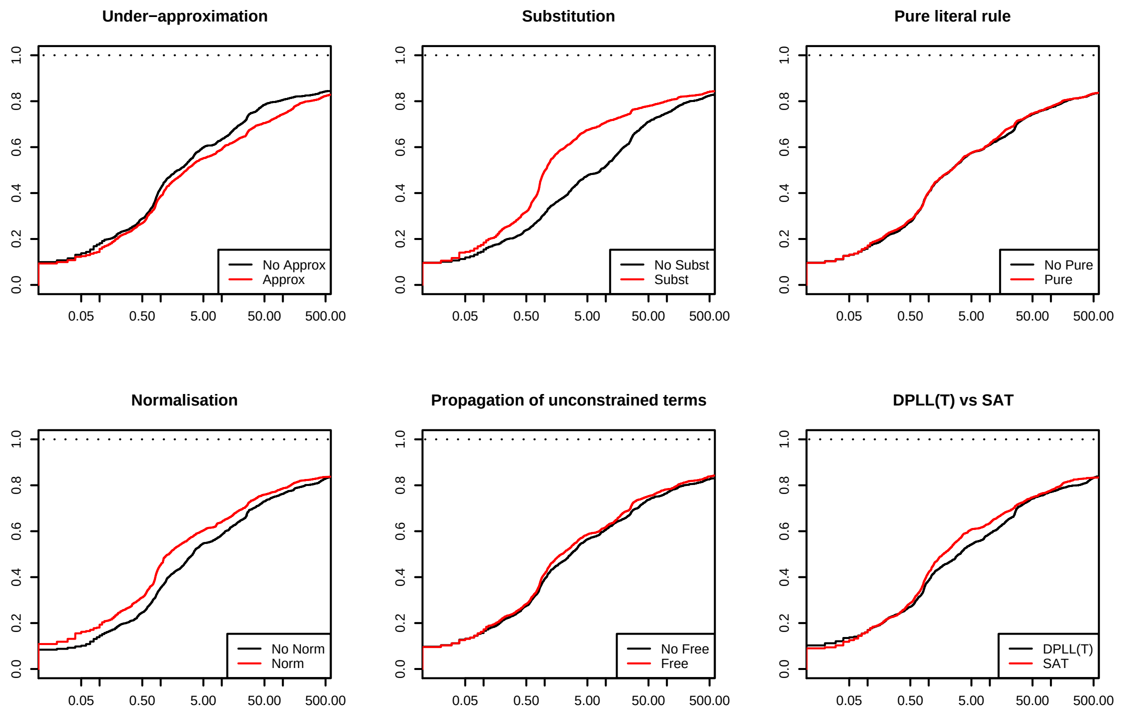


Figure 6.2: Effect of techniques

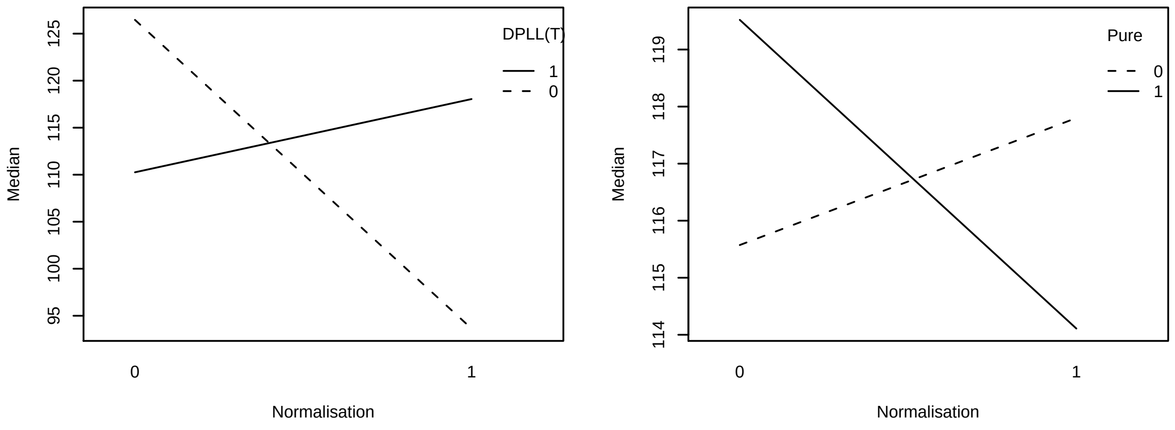


Figure 6.3: Pairwise interactions

provement over DPLL(T), but the effect is not as clear as for the other three techniques. Propagation of unconstrained terms might have an overall effect, but it is difficult to draw any conclusion. The pure literal rule has virtually no effect overall, so may not be very useful.

It is unlikely that the effect of a technique is independent of all other techniques. If that were the case, we could easily find the best configuration of techniques by simply testing one technique at a time, and choose whether to use it or not. It is far more likely that there is some interaction between different techniques, meaning the impact of one technique on performance depends to some extent on whether one or more other techniques are being used. When two techniques interact, we say there is a *pairwise* interaction between them. When n techniques interact, we say there is a n-way interaction. As an example, the choice between the lazy versus eager approach interacts pairwise with *all* other techniques tested in this experiment. Two examples of interaction can be seen in figure 6.3. The horizontal axis is the median execution time, and in the left figure we can see that normalisation hurts the median execution time when DPLL(T) is used, but helps performance when SAT is used. In the right figure, we can see that pure literal elimination although it didn't seem to have an overall effect still seems to interact with normalisation.

All in all, this makes finding the optimal set of technique a difficult problem. An automated approach like that presented in [HHLBS09] may help find a good configuration with a small manual effort.

6.3 SAT vs DPLL(T)

To compare encodings into SAT versus the lazy (DPLL(T)) approach, all SMT-LIB instances have been run with both techniques with a timeout of 1800 seconds. In both cases, all other preprocessing and approximation techniques were used, and the results can be seen in the scatter plot in figure 6.4. Failure to solve an instance is indicated by a red cross, and placed at an execution time $>$ 1800 seconds. The figure clearly shows that it is not easy to say that one technique is clearly better over the entire set of instances.

Looking purely at the real-world instances, encoding to SAT fails to solve 4 instances that can be solved by DPLL(T), whereas DPLL(T) fails to solve 17 instances that can be solved with SAT. Most of the latter instances are from Spear, specifically the wget set[1]. Here there are 16 instances which are easy to solve with an encoding into SAT, but which we fail to solve using DPLL(T). These instances are all satisfiable, and the boolean enumerator happens to generate a truth assignment that is very difficult to check consistency of, and we fail to do so in the time-limit. Had the boolean enumerator made different decisions, the resulting consistency checks in the bit-vector theory solver would have been trivial. This highlights a performance issue with DPLL(T). Once the boolean enumerator has made a decision there is no way for the theory solver to indicate that

[1]Located in QF_BV/spear/wget_v1.10.2 in the SMT-LIB

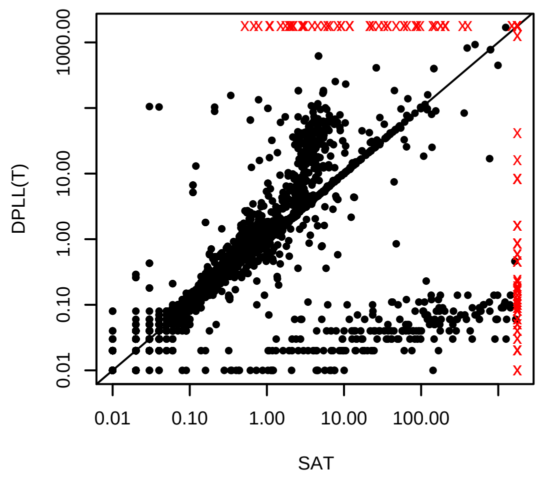


Figure 6.4: SAT versus DPLL(T)

this particular consistency check is "too difficult" and ask the top-level to restart and find another. In an encoding into SAT, this is taken care of by the restart policy of the SAT solver, but in DPLL(T) we are forced to check consistency of the current truth assignment. There are 848 instances where SAT uses less than half the time compared to DPLL(T), and 1036 instances where DPLL(T) uses less than half the time of SAT.

However looking purely at the real-world instances, the picture is more clear. A scatter plot and CDF is shown in figure 6.5. Here we have excluded the SAGE instances, both because they are so numerous as to dominate the figures, and because the performance is very similar on these instances with both techniques. We can clearly see that the encoding to SAT is advantageous, on this subset of the instances.

6.4 Under-approximation

The data in section 6.2 seemed to indicate that under-approximation is not a useful technique. But it is a technique which is targeted at a spe-

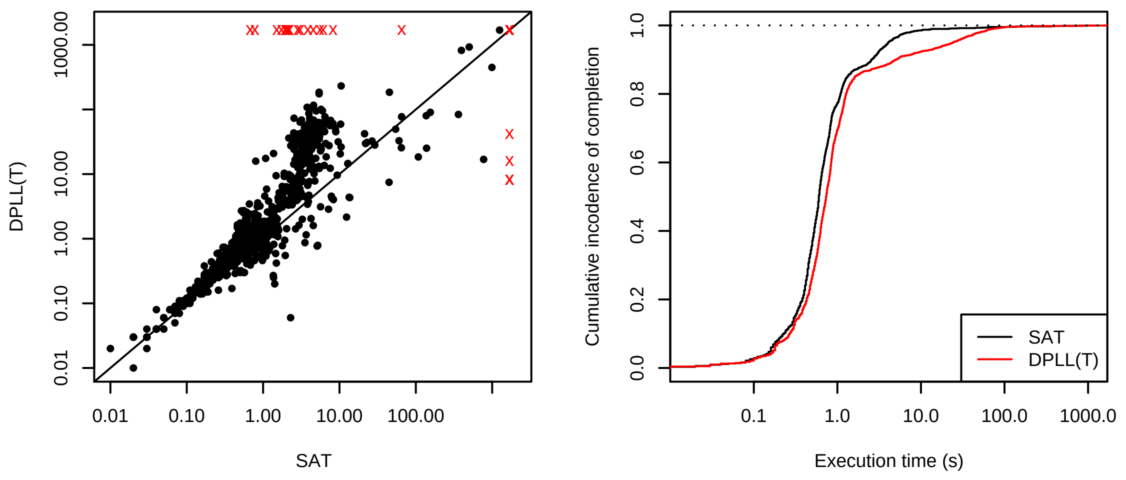


Figure 6.5: SAT versus DPLL(T) on real-world instances

cific case where we have potentially complicated terms which are not relevant for satisfiability, but are still not unconstrained. For this reason we have performed a more extensive experiment on real-world instances in the SMT-LIB, shown in figure 6.6, which includes all real-world instances except those from the SAGE tool. A timeout of 1800 seconds was used, and it seems clear that under-approximations does help performance. This set of instances are dominated by formulae generated by the Calysto static checking tool[2]. Those instances almost all contain division and multiplication operators, making them potentially difficult to reason with. However, these are almost all satisfiable, and under-approximation works very well in producing a significantly simpler formula which is trivial to solve. There are however several instances from other sources which are possible to solve without under-approximation, but which we fail to solve while using under-approximations. We can also see in the CDF that for longer execution times, not using under-approximation seems preferable indicating that an aggressive refinement strategy or a limit on the execution time for

[2]Located in QF_BV/spear in the SMT-LIB

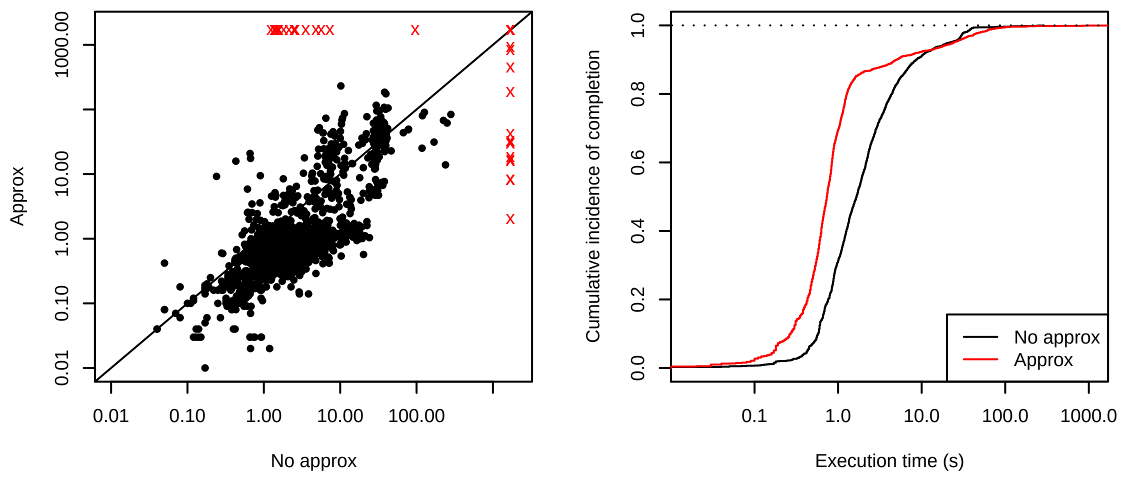
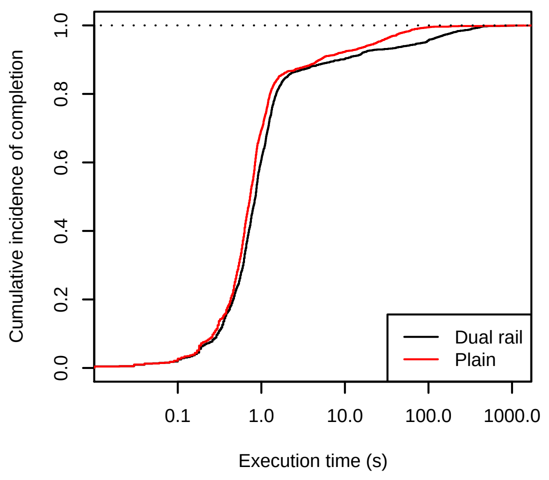


Figure 6.6: Effect of under-approximation on real-world instances

under-approximation may be fruitful.

6.5 Minimal model enumeration

Here we present some experiments using the dual rail based minimal model enumeration technique. We compare execution time and number of conflicts found on the real-world instances in the SMT-LIB again excluding SAGE, the results are shown in figure 6.7. With a timeout of 1800 seconds, we fail to solve 37 instances using DPLL(T), and with the dual rail encoding we fail to solve 31 instances. However, looking at the CDFs it seems clear that for the most case, dual rail suffers in performance. An explanation for this can be seen when looking at the number of conflicts, using dual rail almost always results in more conflicts.



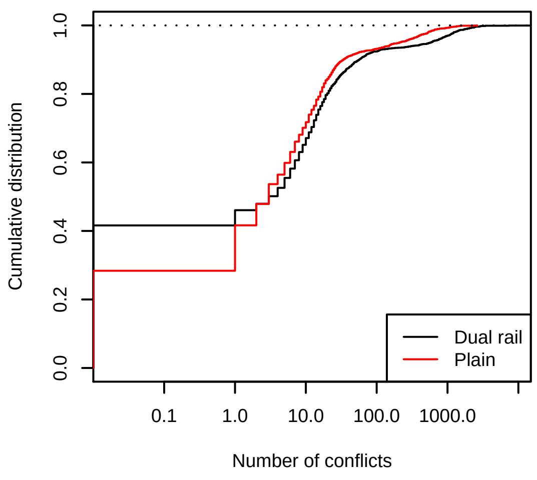


Figure 6.7: Effect of dual rail on real-world instances

6.6 Core rewriting and disjunctive partitioning

In the SMT-LIB benchmark library, there are a few instances in the core fragment of bit-vectors, none of which come from real-world applications. We will focus on the core fragment instances[3] contributed to the SMT-LIB by Bruttomesso, described in [BS09]. These are parametric and designed to show the effectiveness of rewriting techniques on the core fragment. We have tested three different techniques:

basic Uses all preprocessing techniques except the core rewriting technique of [CMR97] (also described in section 4.5) and disjunctive partitioning.

core Same as basic, but with the core rewriting technique

disj Same as core, but with disjunctive partitioning

The result of this experiment can be seen in figure 6.8. The basic variant fails to solve 141 instances, and is dominated by the other two techniques. Using the core rewriting technique, we fail to solve 76 instances, and finally using disjunctive partitioning we are able to solve all instances. However,

[3]Located in QF_BV/bruttomesso/core in the SMT-LIB

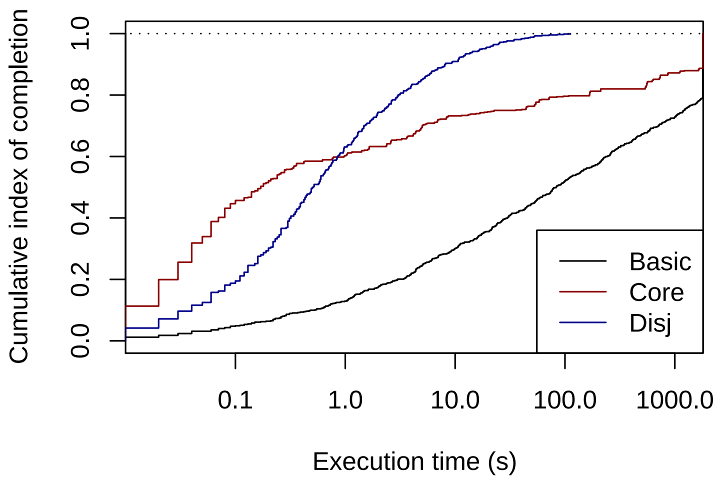


Figure 6.8: Effectiveness of rewriting in the core fragment

for most trivial instances, execution time is worse than just using the core rewriting technique. The reason for this is that when using disjunctive partitioning we are preprocessing all disjuncts independently, and this causes some extra overhead which is noticeable on the trivial instances.

6.7 Packet splitting

To show the benefit of the packet splitting technique, we will here use some instances again described in [BS09][4]. The results of this is shown in figure 6.9. On these instances, the effect of packet splitting is dramatic. This is because these instances, which are all unsatisfaible, can be solved using DPLL(T) and the EUF solver only. Without packet splitting, these instances require bit-vector reasoning, and the instances seem to be designed to show a performance problem for bit-vector solvers based around encoding into SAT. MathSAT is able to solve 31 out of the 64 instances if

[4]Located in QF_BV/bruttomesso/simple_processor in the SMT-LIB

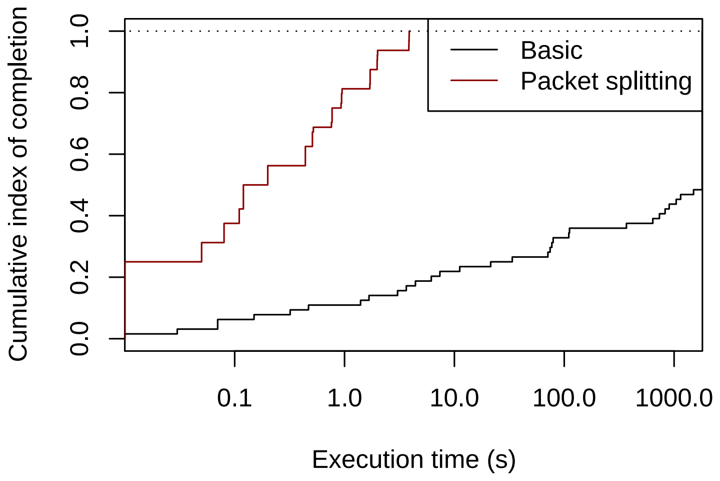


Figure 6.9: Effect of packet splitting on simple_processor instances

packet splitting is not used within 1800 seconds..

6.8 Difference propagation

For difference propagation, we will once again look at instances from [BS09][5]. These examples are somewhat extreme, when difference propagation is used, all instances become propositionally unsatisfiable. In general checking whether an arbitrary term is injective may be computationally expensive, so this technique probably has limited value. The results are shown in figure 6.10, and the difference is dramatic. This is due to the fact that for these instances, discovering that some functions are injective is the main difficulty in showing these instances to be unsatisfiable. It should be noted that even without this technique, all instances could still be solved within the time limit, the maximum time was 1255 seconds.

[5]Located in QF_BV/bruttomesso/lfsr in the SMT-LIB

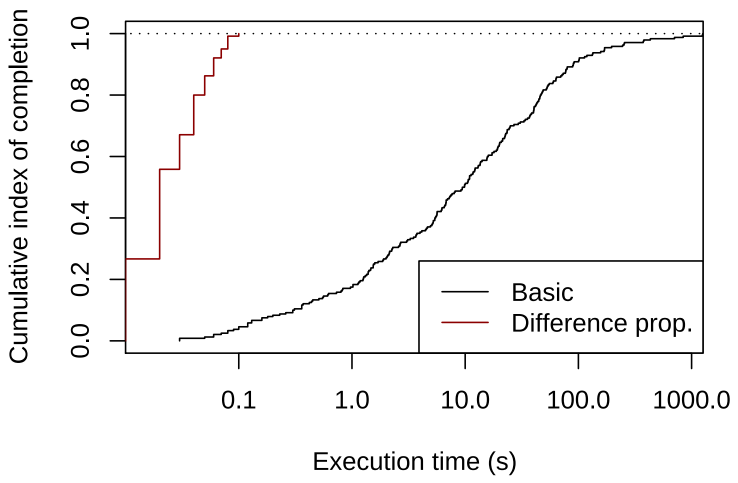


Figure 6.10: Effect of difference propagation on lfsr instances

6.9 Clustering

Performance of lazy clustering on real-world instances can be seen in figure 6.11. Performance is significantly worse, in large part due to the extra overhead of managing the clusters, which need to be merged repeatedly. Another problem is in excessive memory usage, which is significantly higher compared to not performing clustering. There are several cases where the solver run out of memory, the reason for this is as yet unclear.

Using the eager clustering technique, which clusters all atoms before search starts once and for all, we get the results shown in figure 6.12. The results are very similar to those found with lazy clustering, but eager clustering appears to be slightly better below 10 seconds and slightly worse above 10 seconds. This is likely due to less overhead in cluster management with the eager clustering technique, an advantage which disappears once the lazy approach has merged most of the possibly large initial number of clusters.

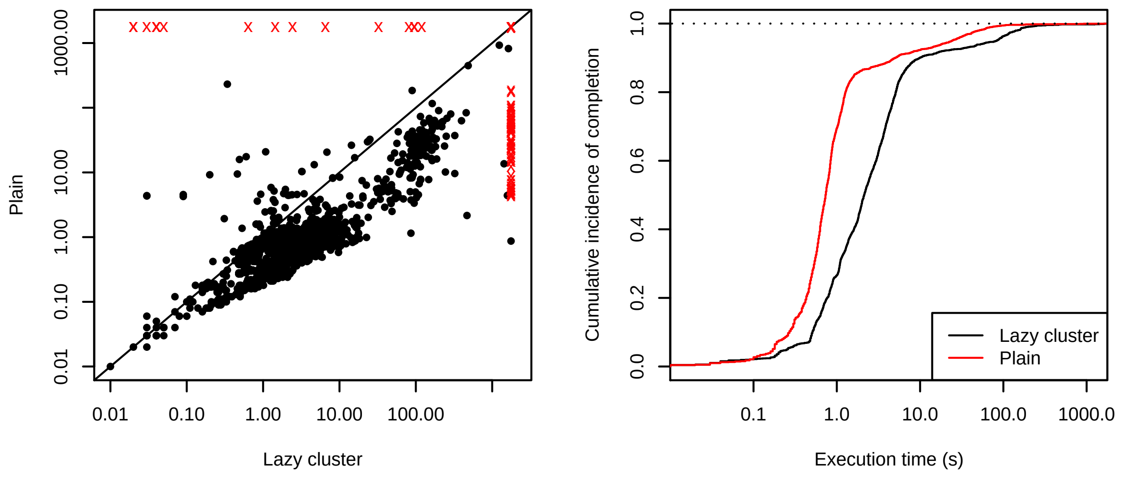


Figure 6.11: Lazy clustering

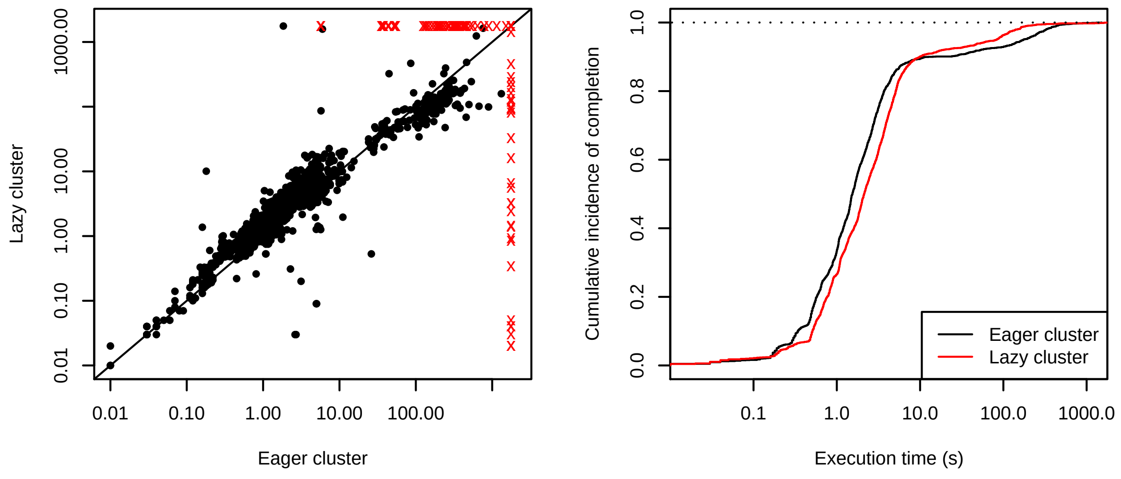


Figure 6.12: Eager clustering

6.10 Comparison with other solvers

A comparison with other state-of-the-art solvers is commonplace, but it also has its problems. In this case, many of the other solvers are closed source (as is currently MathSAT), and in some cases there is not much publicly available information regarding the techniques used. Comparing several completely different implementations, in some cases not knowing precisely the techniques applied in the different solvers can make drawing conclusions difficult. However, we shall make some attempt in this section.

For comparison with other solvers, we have chosen the following[6] SMT solvers

- Beaver [JLS09] SMT-COMP 2009 version
- Boolector [BB09a] version 1.2
- OpenSMT [BPST10] SMT-COMP 2009 version
- STP [GD07] SMT-COMP 2009 version
- Sword [WFG$^+$07] SMT-COMP 2009 version
- Yices [DdM06] SMT-COMP 2009 version
- Z3 [dMB08] version 2.3

All solvers have been run with a timeout of 1800 seconds, and a memory limit of 3.5 GB on the bit-vector instances in the SMT-LIB. An overview of the result is shown in the CDF in figure 6.13 The SMT-LIB contains a relatively large number of bit-vector instances from several different sources, so it may be difficult to gain much insight from this overview. What seems clear is that there is a large number of trivial instances, the median execution time for many solvers (Boolector, Yices and MathSAT) is less than 0.01 seconds. For all solvers the median execution time is less than 0.1 seconds. These trivial instances are either hand-crafted, or coming from the SAGE tool [GLM07][7].

[6]CVC3 was not used because of time constraints

[7]Located in QF_BV/sage in the SMT-LIB

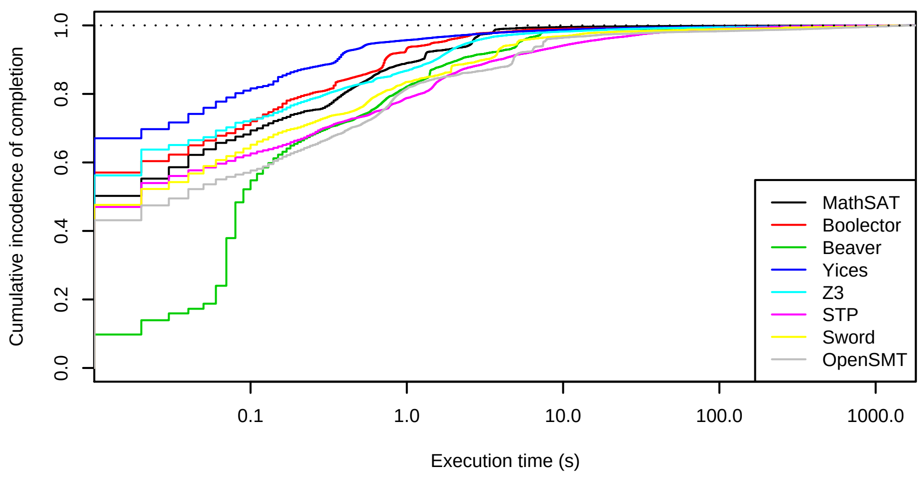


Figure 6.13: CDF of comparison with other solvers

To get a better picture, we will look at some of the sets of instances individually. The data in figure 6.13 includes all bit-vector instances in the SMT-LIB, but here I will focus on those which come from real-world applications of SMT solvers. We divide these into several subsets based on their origin as follows:

SAGE The instances coming from SAGE

Spear The instances generated by Calysto [BH08] and contributed by Domagoj Babić.

UCLID Various instances contributed from UCLID or related tools

Others All other "real-world" instances

Crafted All hand-crafted instances

The results on each subset are shown in figures 6.14 and 6.15, and an overview of the number of instances which could not be solved (either by timeout or exceeding the allowed memory) is shown in table 6.1. For the SAGE set, it seems clear that these instances are trivial. 4 of the solvers can solve every single one of the instances, and almost all instances with very short execution times. These come from an application which may

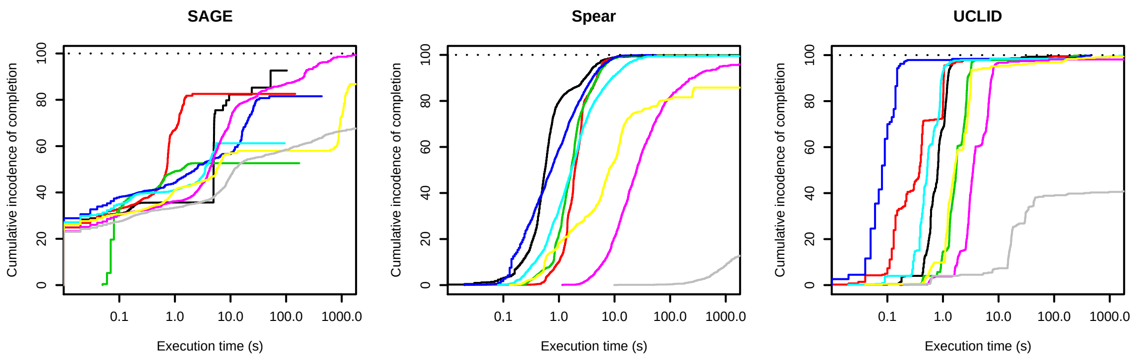


Figure 6.14: CDFs of comparison with other solvers

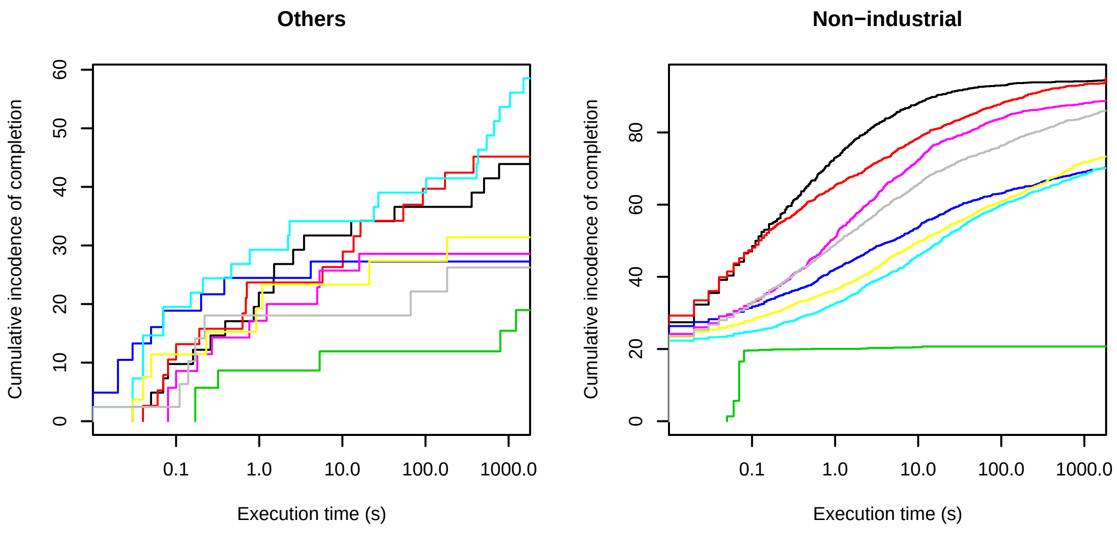


Figure 6.15: CDFs of comparison with other solvers

Table 6.1: Solver failures by set

Solver	SAGE	Spear	UCLID	Others	Non-ind.
Beaver	37	5	0	30	422
Boolector	0	0	1	21	87
MathSAT	0	1	0	23	86
OpenSMT	390	1465	251	34	214
STP	2	486	8	31	182
Sword	13	235	17	33	408
Yices	0	0	0	31	452
Z3	0	7	1	17	458

generate a large number of instances in a single run of the application, and all instances are trivial and related to each other. In this case there may be more effective ways of solving them than treating them as individual instances to solve as we do here. Chapter 7 shows another example of such an application as well as some ideas for what can be done. Separating these from the rest of the instances might give a clearer picture of performance in other cases, information which would otherwise be drowned by the SAGE instances which form the majority of the bit-vector instances in the SMT-LIB.

The lagging performance of OpenSMT can be attributed to the lack of preprocessing performed in the solver; This is the first version supporting bit-vectors, and has not yet been optimised. Similarly for STP and Sword, the large number of unsolved instances on Spear and UCLID might be caused by missing preprocessing techniques, although it is difficult to say.

In the SAGE subset, Beaver seems to be lagging in performance, both on the very simplest instances and by failing to solve 37 of them. The reason for this is unknown, perhaps there is some inefficiency that manifests itself on the simplest instance in this case. These formulae, although trivial can be non-trivial in size, and so an efficient implementation makes all the difference.

The problem area for most solvers seem to lie in hand-crafted instances. This is hardly surprising, since for the most part these have been designed to test the limits of the solvers. They are often instances of parametric problems, with one or more parameters which control the difficulty of the generated instances, and used to study how particular aspects of a solver scale or why particular solving techniques are in the contributors mind necessary.

More interesting is perhaps the instance in “Others” which mostly include instances which were made public very near to the 2009 competition

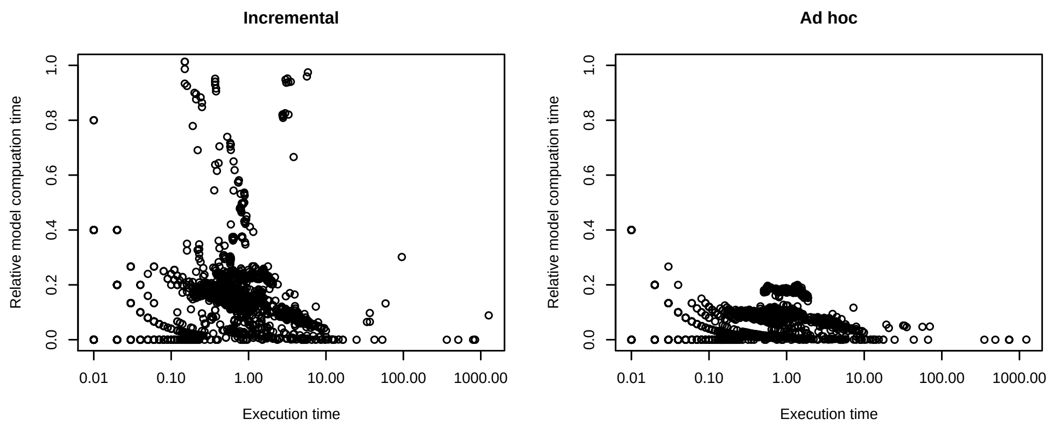


Figure 6.16: Overhead of model computation

giving solver authors little time to study them. They appear to be sufficiently different from other instances making several of them difficult to solve. Hopefully this doesn't indicate the techniques which work so well for the other older sets of instances in the SMT-LIB are really only useful for those specific instances. Indeed there is evidence that this is not the case. The instances in SAGE were also introduced very close to the competition, and so were several instances in the UCLID set. For these, the techniques used in many of the solvers appear to work quite well.

6.11 Model computation

To test how much overhead model computation generates when using all of the preprocessing and approximation techniques described in this thesis are used, we have measured this overhead on all SMT-LIB bit-vector instances. In figure 6.16, we can see all satisfiable instances solved with MathSAT using a 1800 second timeout. The figure shows the relative overhead compared to total execution time for both the incremental and the ad hoc model computation technique. The median overhead is 13% for the incremental technique and 2.9% for the ad hoc technique, the 3rd quartile is 20% and 9.3% respectively. We can also clearly see that the ad hoc technique seems to be more stable, even for larger execution times we can see outliers with significant overhead in some cases using the incremental technique, whereas the relative overhead of the ad hoc technique becomes smaller as execution times increase without apparent exception. The maximum time for model computation is 112 seconds using the incremental technique but only 3.4 seconds for the ad hoc technique. It should be noted that the current implementation of the ad hoc technique is an early prototype with no considerations for efficiency, a proper implementation can be expected to improve performance. Using the incremental technique, there are 12 instances where the solver either times out or runs out of memory during model computation.

For some of the hand-crafted instances by Brummayer and Biere[8] the advantage of the ad hoc technique for model computation is especially clear. These instances have been crafted explicitly to show the necessity for propagation of unconstrained terms, and without this technique they are very difficult to solve. Using propagation of unconstrained terms these instances become trivial, because the propagation effectively removes all non-trivial parts of the formulae. But once a model is needed, these parts must be taken into consideration during model computation. With the incremental technique of model computation, MathSAT runs out of memory after around 20 seconds during computation of the model, but with the ad hoc technique they can be solved and a model produced trivially, using less time than can be reliably measured (recorded as 0 seconds).

It should perhaps be noted that with under-approximation, 7 out of the 10 instances in that set can be trivially solved, simply because the under-approximation step happens to select suitable values for enough variables to make these instances trivial. But this might be attributable more to

[8] Located in QF_BV/brummayerbiere4 in the SMT-LIB

luck than anything else.

An alternative to the model computation techniques tested here is to simply avoid using preprocessing or solving techniques which make model computation less than straightforward. This has been the approach taken by MathSAT in the past, and the approach taken by some of the other solvers as well. As an example, when asking Boolector to compute a model, one can see from its output that it creates larger SAT problems for the underlying SAT solver PicoSAT, and that solving time increases. Which approach works best is instance-dependent, but it is clear that for instances where propagation of unconstrained terms is important, being able to compute models while still applying this technique is an advantage. Another important consideration is that by disabling techniques interfering with model computation, this has a negative impact also on unsatisfiable instances. With the ad hoc or incremental model computation techniques, there is no impact on unsatisfiable instances at all.

6.11.1 Other solvers

We will also take a brief look at model computation on other solvers. To keep computation time reasonable, we will only look at two of the other solvers, Boolector and Z3. The result for Boolector can be seen in figure 6.17, which shows Tukey's mean difference plots[9] for satisfiable and unsatisfiable instances respectively, plotting

$$(\frac{x+y}{2}, x-y)$$

where x is the execution time when not requesting a model, and y is the time when requesting a model. A positive differences means in this case that requesting a model causes some extra overhead in the solver, and a negative difference means formulae were solved with less execution time.

[9]Sometime also called Bland-Altman plots

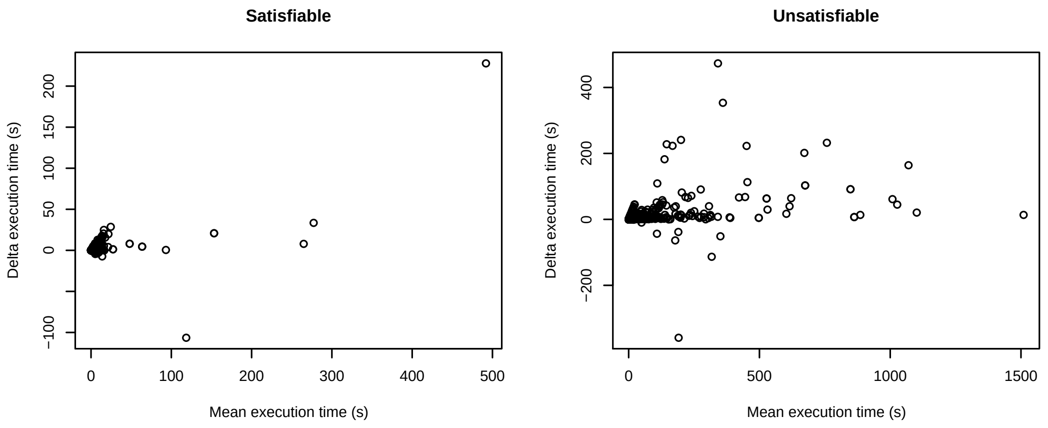


Figure 6.17: Model computation in Boolector

The experiments have been carried out by running the solver twice on each instance; Once with the flag `-m` to request a model. To get more accurate data, the solver should be run several times with each set of options, but this experiment will hopefully give some indication of the scale of the differences in execution time regardless. Since Boolector disables some techniques when a model is requested, some formulae can be solved in less time than otherwise, but in general a certain overhead for model computation is incurred. Since some techniques are disabled, we can also see that there is an effect on execution time for unsatisfiable and satisfiable instances alike. When a model is requested, the solver fails to solve 10 more satisfiable instances, but also solves one more unsatisfiable instance.

Using Z3, we get the results shown in figure 6.18. We can see that there is considerable variation in execution time, although in this case the solver solves the exact same number of instances in total. The solver does appear to modify which techniques are applied when a model is requested as can be seen by the varying performance on unsatisfiable instances.

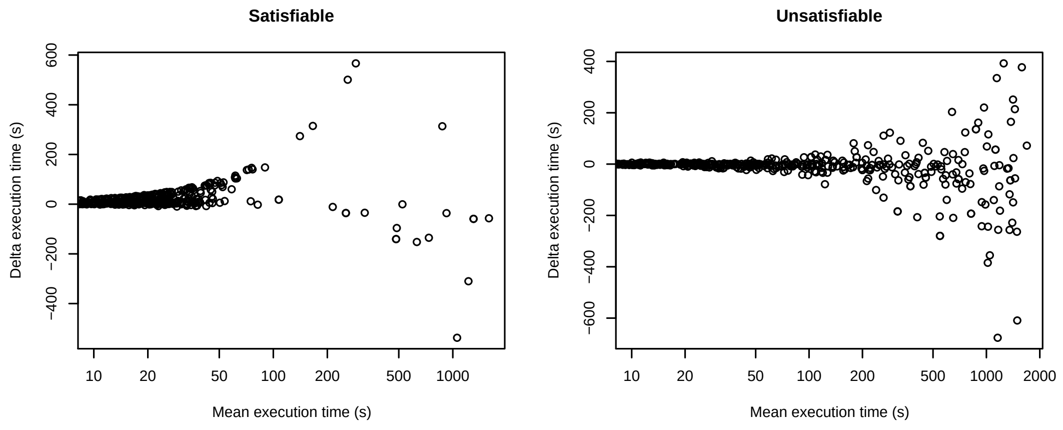


Figure 6.18: Model computation in Z3

6.12 Other techniques

Here we make some brief notes on the remaining techniques.

EUF layering Using the EUF solver does have an impact when using DPLL(T), but the effect is not of major importance. However, given that using the solver incurs a practically negligible overhead there seems to be little reason not to use it in general.

Over-approximation The over-approximation delivers a clear benefit, but unfortunately only on a fairly small number of instances.

Under-approximation in theory solver No variant has yet been found which delivers reasonable, let alone improved, performance.

Static learning The axioms that are currently instantiated in static learning are few in number and quite basic, and the impact on performance is small. It is possible that more powerful axioms could deliver a decisive performance gain, but it is unclear exactly what these axioms would be. Simply checking which conflict sets which are found during search and then adding these manually may lead to an endless number of axioms to be added as new formulae are solved, and this will soon become unmanageable. A more principled approach would clearly be necessary.

Chapter 7

An industrial case study

A modern Intel CPU may have over 700 instructions in the Instruction Set Architecture (ISA), some of them for backward compatibility with the very first X86 processors. Although the processor itself is a Complex Instruction Set Computer (CISC), the *microarchtiecture* (basically the implementation of the ISA) is what can be likened to a Reduced Instruction Set Computer (RISC).

The instructions in the ISA are translated into a smaller set of simpler instructions called *microinstructions* or *microperations* (sometimes called μops). The idea of using a simpler microarchitecture to implement a complex ISA was developed by Maurice Wilkes in the late 1940s, first published by Wilkes and Stringer [WS53] describing the approach taken with the EDSAC computer. Wilkes realised that the implementation of instructions were essentially a sequence of simpler operations that should be performed, and developed the idea of a microprogram. Each instruction in the ISA is translated into a small program in microcode. Later Wilkes enhanced the microcode instruction set with conditional branch instructions and this was used in the EDSAC 2, completed in 1958 [Wil92].

There are many reasons for a microcode-based CPU architecture, some of them may be

- It abstracts the ISA from the underlying microarchitecture. This makes it easier to change the microarchitecture while still supporting the same ISA maintaining backward compatibility
- It simplifies the microarchitecture. The decode and execute logic can be made simpler, this may help with complex instructions sets like the IA-64.

Most instructions in Intel processors correspond to a single microinstruction or at least a small number (at most 4) which can be translated directly into microcode, larger programs are stored in a microcode program memory called the Microcode ROM. Some of these programs may be surprisingly large, such as string move in the Pentium 4 which was reported in [HSU+01] to use thousands of microinstructions. Verification of these programs is a critical, but time-consuming process. To aid in the verification effort, a tool chain called MicroFormal has been developed at Intel starting in 2003 and under intensive research (in collaboration with academic partners) and development since. This system is used for several purposes:

- Generation of execution paths. These execution paths are used in traditional testing to ensure full path coverage, and to generate test cases which execute these paths, described in [AEMS06, AEO+08].
- Assertion-based verification. Microcode developers annotate their programs with assertions, and these can be verified to hold using MicroFormal.
- Verification of backwards compatibility, described in [AEF+05]. When a new generation CPUs are developed, they should be backwards compatible with older generations, although they may include more features.

At the heart of this set of tools is a system for symbolic execution of microcode, which is the part of the tool chain where we will concentrate.

In this chapter, we will start by giving a high-level overview of part of the MicroFormal tool set in order to give an understanding of the application necessary to understand the usage of decision procedures in this context. Then some of the techniques that are used to improve performance are described. Starting in section 7.3, the contribution using MathSAT in this application domain is discussed, and lastly there is an experimental evaluation section giving some evidence of how the proposed techniques perform.

7.1 Intermediate Representation Language

To simplify the symbolic execution engine, it does not work directly with microcode. Instead it works with an intermediate representation called Intermediate Representation Language, or IRL. This is a simple language with all features necessary to model microcode programs. Microcode programs are translated into IRL by a set of IRL templates, which define the translation from microcode instructions into a corresponding sequence of IRL instructions. This makes adapting the tool chain to a new microarchitecture simpler, all that needs to be written is a new set of templates describing how instructions are translated into IRL. Another benefit of using IRL is that it would be possible to handle other types of low-level software. Although the precise details of the language used in MicroFormal is not public, here we describe a simple language which will hopefully give an understanding of the main features of the language relevant for this work. It should be pointed out that none of the example programs in this chapter are real microcode programs.

The details of the IRL language have not been made public, but some features of it are known, e.g. see [AEF+05]. A grammar summarising the known details for this simple language is sketched in figure 7.1. This is meant to give the general idea of relevant language features, it is not meant as an accurate representation of a subset of the real language. In short the language has the following main features:

- All variables in IRL are of bit-vector of type. For logical values, single bits are used with 1 meaning true and 0 false.
- An instruction is either an assignment $V_1 := V_2$, a branching instruction or a terminating instruction.
- All instructions have a location, being the address in the microcode ROM where they are stored
- The operations over bit-vector terms are all of the operators in the SMT-LIB language, extended with bit versions of all relational operators, with the difference that they here are functions with a bit codomain.
- Branching instructions branch to a location indicating a specific instruction. This location can either be given by a constant (a direct branch) or a bit-vector variable (an indirect branch).
- Terminating instructions are either a normal termination or an exception. Exceptions are used to model abnormal termination of the microcode program.

Due to indirect branches, even computing a control flow graph (CFG) for an IRL program is a non-trivial task.

The correctness of the translation from actual microcode programs into IRL is crucial, but outside the scope of this high-level description of MicroFormal. We will also make many simplifications and skip over details that are not immediately relevant for the work presented.

```
⟨declaration⟩ ::= var ⟨variable-list⟩ :  BitVector[ ⟨width⟩ ] ;
⟨statement⟩   ::= ⟨location⟩ : ⟨instruction⟩ ;
⟨instruction⟩ ::= ⟨assign⟩ | ⟨branch⟩ | ⟨exception⟩ | ⟨exit⟩
⟨assign⟩      ::= ⟨variable⟩ := ⟨expression⟩
⟨expression⟩  ::= ⟨constant⟩ | ⟨variable⟩ | ⟨operator⟩ ( ⟨variable-list⟩ )
⟨branch⟩      ::= ⟨condition⟩ ? goto ⟨target⟩
⟨target⟩      ::= ⟨location⟩ | ⟨variable⟩
⟨exception⟩   ::= exception ⟨name⟩
⟨exit⟩        ::= exit
```

Figure 7.1: Intermediate representation language grammar

7.2 Symbolic execution of microcode

The MicroFormal symbolic execution engine is used to compute a set of *paths* through a program, where a path is a sequence of locations that the program can follow from start to finish. A path through the program for which there exists an assignment to input registers such that the execution follows that path is called *feasible*. A *partial path* is a path from the start to some non-exit location within the program. The problem solved by the symbolic execution engine is to find all paths from the starting location to one of the exit locations. Symbolic execution [Kin76] is a form of execution where all input (or initial values of variables) are symbolic. Take the following simple example, which swaps values in two bit-vector variables

```
x, y : BitVector[64];
l1: x := x + y;
l2: y := x - y;
l3: x := x - y;
l4: exit;
```

To execute this program symbolically, we start by giving the symbolic values x_0, y_0 to the variables `x` and `y`. For the first assignment `x := x + y` we create a new symbolic value x_1 and compute how it relates to the symbolic values of the variables in the right hand side of the assignment $x_1 \hat{=} x_0 + y_0$ and so on for all instructions in the program, accumulating the equations that define the symbolic values we have created.

`l1:`	`x := x + y`	$x_1 \hat{=} x_0 + y_0$
`l2:`	`y := x - y`	$x_1 \hat{=} x_0 + y_0,\ y_1 \hat{=} x_1 - y_0$
`l3:`	`x := x - y`	$x_1 \hat{=} x_0 + y_0,\ y_1 \hat{=} x_1 - y_0,\ x_2 \hat{=} x_1 - y_1$
`l4:`	`exit`	$x_1 \hat{=} x_0 + y_0,\ y_1 \hat{=} x_1 - y_0,\ x_2 \hat{=} x_1 - y_1$

By expanding the final definitions[1] we can see that the final values of the variables (x', y') depend on the initial given by the equations $x' = (x_0 + y_0) - x_0$ and $y' = (x_0 + y_0) - y_0$ which can be simplified to $x' = y_0$ and $y' = x_0$ respectively.

Apart from the current symbolic values for all variables in the program, during symbolic execution we also keep track of a *path condition* and the program location. The path condition is the conjunction of the conditions on the conditional branches along the current execution path, expressed in terms of the initial symbolic values. A more detailed description of how this may be performed is presented in [KaaV03].

Execution starts by executing the basic block (a non-branching sequence of instructions) starting at the beginning of the program to the first branch instruction. This partial path is marked as *open*. Then as long as there exists an open partial path π, all feasible branch targets continuing this path are computed by generating a sequence of *path feasibility conditions* which are sent to a decision procedure. A path feasibility condition is the path condition which would result when branching into a given branch tar-

[1]These definition are normally stored in their expanded form, the unexpanded form is shown here only for clarity

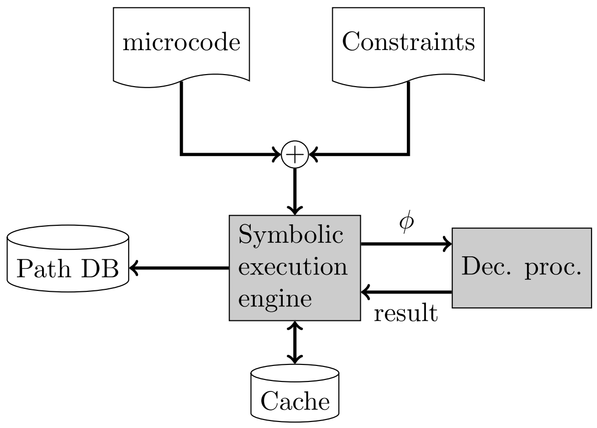


Figure 7.2: Overview of the MicroFormal symbolic execution engine

get. If this path condition is satisfiable, the target is feasible in the sense that there exists some input that would execute down the current path and branch to that target. For every feasible branch target, MicroFormal extends π with the basic block starting at that location into a new path π'. If π' reaches a terminating instruction, this path is stored in the path database. Otherwise it is marked as an open path and the execution continues. An overview of the symbolic execution engine in MicroFormal can be seen in figure 7.2.

A path feasibility condition for a partial path π is a formula which describes the possible branch targets symbolically in terms of the input variables combined with some query on the target, which is used to determine the possible values for the branch target. The details on the formulation of path feasiblity conditions are outside the scope of this thesis, here we will focus on the decision procedure used to solve these and other decision problems generated by MicroFormal.

From the point of view of the decision procedure, the symbolic execution engine feeds it a sequence of formulae one after another, and the result sent back for one formula affects the future paths taken by the symbolic execution engine and therefore also which formulae it receives in the future.

7.2.1 Some improvements to the basic symbolic execution algorithm

To improve performance of the symbolic execution, several techniques are used as described in [AEO+08]. Here we will briefly present three of them. One problem is the sheer size of the formulae sent to the decision procedure. In order to reduce the size of formulae, MicroFormal merges sets of partial paths ending up in the same location into a single path by introducing extra variables and conditional assignments. The details are explained in [AEO+08], but for our purposes the relevant effect this has is that it removes open partial paths which have so far been generated, and replaces them with a new merged path which is equivalent to but syntactically different from the previous paths.

Two other techniques that are used are based on *caching* and *SSAT*, briefly described below.

Caching of solver results The result of each solver call is stored in a cache shown in figure 7.2. This cache stores for every formula solved whether it is satisfiable or not, as well as the model for satisfiable formulae. If a formula α has been shown previously to be satisfiable, then any future formula $\alpha \vee \beta$ can be determined to be satisfiable without calling a solver. In the same way, if α has been shown to be unsatisfiable, any future occurrence of it as a subformula in future formulae can be replaced with $\perp$ as a simplification step.

In case this fails, it is possible to take a model stored in the cache and evaluate the current formula with it. In case it evaluates to true, there is no need to call the solver. It may also happen that the evaluation results in a new smaller formula due to some variable occurring in the formula which did not occur in the model. In this case it is possible to send this simplified formula to the solver, if it is satisfiable it was possible to extend the old model into a model for the current formula. The motivations for caching models is that if a path feasibility check for some partial path shows it to be feasible, there exists an extension to this paths. Therefore the model for this path feasibility check should be useful in the future.

SSAT In most cases, the symbolic execution engine generates a single formula which must be solved before execution can continue, because the satisfiability of this formula determines how the execution should proceed. But in some cases, it is possible to generate more than one formula, which it can predict must be solved regardless of their satisfiability. One technique used to improve performance of solving in these cases is to apply *Simultaneous SAT* (SSAT) introduced by Khasidashvili et al [KNPH06]. This technique is a modification of the standard DPLL algorithm which allows the user to solve multiple *proof objectives* for a single formula in CNF. The solver will solve all proof objectives and for each of them return their satisfiability and a model in cases of satisfiable proof objectives. The motivation behind this technique is twofold; First a single model may satisfy more than one proof objective, and second information learnt while solving one proof objective may be helpful in solving the others. Both of these assume that the proof objectives are closely related to each other, which is the case in this application.

7.3 Focus of this work

The focus of this case study is the decision procedure and its interaction with the symbolic execution engine. The aim is twofold:

- Improve execution time for each solver call
- Discover techniques which make more efficient use of the decision procedure.

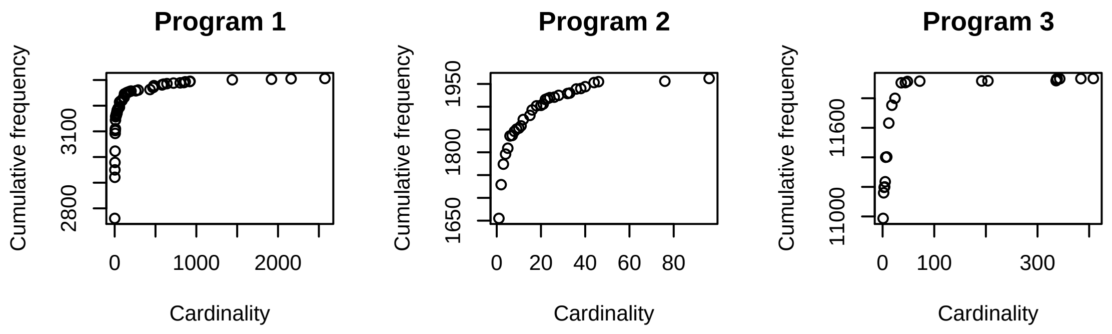


Figure 7.3: Cumulative frequency of set cardinality

For this reason, we will use a highly simplified view of MicroFormal as a system that captures the essential features of the tool and abstracts away those not relevant here. In short, we will view MicroFormal as a generator of formulae to be solved. From the solver point of view, the problem can be stated as solving a sequence of nonempty sets of formulae

$$\Phi_1, \Phi_2, \ldots, \Phi_N$$

where each Φ_i is a nonempty set of formulae. The sequence of formulae is not known a priori, meaning that the set Φ_{i+1} is not known until all formulae in the set Φ_i have been solved. Since all formulae in the sequence derive from the symbolic execution of the same microcode program, they will share the same set of variables. It will not be the case that one formula contain some variable v of one type, and another formula contain a variable with the same name, but a different type.

Most of the sets in the sequence will typically contain a single formula, but they can also in some cases contain large numbers of formulae, even thousands. Sample data for three programs is shown in figure 7.3, which shows the cumulative cardinality. For the three programs, the sequences contain between 84% and 92% sets with a single formula. To separate the two cases, we will call them singleton sets and non-singleton sets.

Definition A set with a single element will be called a *singleton*. Sets with more than one element will be called *non-singleton*.

In this work we will focus on three aspects of the usage of decision procedures in MicroFormal:

– The problem of solving singleton sets efficiently
– Improvements to the caching of solver results
– The problem of solving non-singleton sets efficiently

In the remainder of this chapter, we will consider various solutions to these three problems, provide an experimental evaluation, and finish by an evaluation against the incumbent solver Prover used in MicroFormal.

7.4 Reuse of learnt information

In MicroFormal, most sets in a sequence contain a single formula, and we need to solve this one formula to advance the search.

Each formula is usually very similar to the previous formula. This can be seen by measuring similarity for a number of medium to large sequences. Seeing each formula as a Directed Acyclic Graph (DAG) using perfect sharing (sometimes also called hash consing) we can compare the similarity of a pair of formulae $\langle \phi_1, \phi_2 \rangle$ by measuring the number of nodes in the DAG for ϕ_1 which do not occur in the DAG for ϕ_2. Given two formulae ϕ and ψ we compute the ratio of terms occurring in ϕ which do not occur in ϕ to the total number of terms in ϕ and vice versa. The similarity between the two will be taken as the minimal of the two ratios. We can see the result in figure 7.4. The figure shows measurements of similarity between each pair of consecutive formulae in each sequence.

Consecutive formulae appear to be highly similar, with a median similarity of 78%, 95% and 99% respectively, and this is something we would

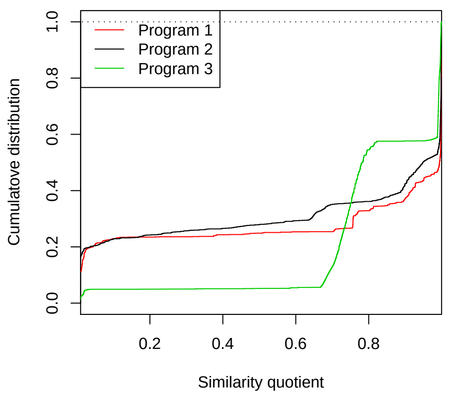


Figure 7.4: Pairwise formula similarity of instances

wish to take advantage of. The cases with very small similarity between formulae is almost always combined with at least one of the two formulae being very small. The approach we have taken is to reuse learnt information from the solving of one formula to help solving the next.

Modern solvers are often quite good at handling irrelevant information, the heuristics used in modern SAT solvers often manage to focus on the relevant parts of a formula, ignoring the rest. We will take advantage of this by retaining all information stored in the solver from one formula to the next. The basic algorithm when solving a sequence of individual formulae $\phi_1, \phi_2, \ldots$ is to first create one fresh predicate P_1, add the formula $P_1 \Leftrightarrow \phi_1$ and solve under the assumption of P_1 to discover if ϕ_1 is satisfiable. Then, we create another fresh predicate P_2 and add $P_2 \Leftrightarrow \phi_2$ to the solver and solve under the assumption of P_2. In the second iteration, the complete formula in the solver will be $(P_1 \Leftrightarrow \phi_1) \wedge (P_2 \Leftrightarrow \phi_2)$ and all learnt information from the solving of ϕ_1 is still available when solving ϕ_2.

Given the incremental interface of MathSAT, this is very simple to achieve and it is not necessary to introduce the fresh variables. If the bit-vector theory solver is set up to retain its state from one formula to the next (keeping all clauses and heuristic information), then we can solve each formula by pushing a backtrack point, solving, and then popping the backtrack point as shown in algorithm 7.1.

Algorithm 7.1: Solve reusing information

```
  Input: $\phi_1, \phi_2, \dots, \phi_N$
1 foreach $i \in [1..N]$ do
2     Push backtrack point
3     Assert($\phi_i$)
4     Solve
5     Pop backtrack point
6 end
```

Although the solver might be good at ignoring irrelevant information, eventually as the amount of irrelevant clauses grow these will have a negative impact on performance, and of course also on memory usage. Therefore it is important to at some point remove this information. The simplest possible approach would be to just throw away *all* information irrelevant or not, and then solve the next formula as if it is the first one encountered. The advantages of this is that it is very easy to implement and to use. The disadvantage is that we also throw away potentially useful information.

The main question with this approach of dealing with the accumulation of irrelevant information is, when to reset the solver? Several solutions suggest themselves:

– Use fixed reset frequency. Reset every k formulae.
– Reset based on subformula reuse. Measure how much the next formula is already known to the solver, how much of it is not previously known, and how much of the solver information is irrelevant.
– Use an adaptive strategy. Measure solver performance, and try to predict when degradation starts to occur. Reset before it becomes detrimental.

– Delete only irrelevant information from the solver, and keep the rest. This sounds like the best solution, but computing which information is irrelevant is not a simple problem. Just because it is not relevant for the current formula does not mean it will not become relevant again in the future.

Even in the cases where no learnt information is explicitly removed, the underlying solver is free to remove learnt clauses, as any standard SAT solver does. This can be more or less aggressive, and works regardless of how the solver is used. However, these techniques will not work on the original clauses generated from encoding of the formulae given to the solver, only the learnt clauses. In this application an aggressive heuristic for clause removal may be interesting, such as suggested in [AS09] and used in the glucose SAT solver.

7.5 Unsatisfiable cores for result caching

An alternative to speeding up the solving of each formula is to reduce the number of formulas that need to be solved in the first place. This approach has already been used successfully within MicroFormal by caching the result of each solver call. This means storing whether a particular formula is satisfiable or unsatisfiable, and if satisfiable also storing a model for it as decribed in section 7.2.1.

In particular if a previous formula ϕ is unsatisfiable and the current formula ψ contain ϕ as a subformula we can deduce that this formula is unsatisfiable, and replace all occurrences of it with $\bot$. In this way, it may be possible to deduce that the new formula is unsatisfiable without calling the solver.

A possible improvement to this is to store not just that a formula is unsatisfiable, but the *unsatisfiable core* of that formula. There are several ways of computing an unsatisfiable core, in this case the formulae are conjunctions of a large number of subformulae we will take as an unsatisfiable core a subset of the conjuncts of the formula which by themselves are unsatisfiable. Normally, an unsatisfiable core is defined on CNF formulae as a subset of the clauses which is unsatisfiable. In this case the formula is not in CNF, and producing an unsatisfiable core in terms of the CNF generated from the original formula is not directly usable in this case. Therefore we will use a more coarse-grained definition of unsatisfiable cores which is easier to use in this application

Definition Given a formula $\wedge_i \phi_i$ where $\mathbb{C} = \{\phi_i | i \in [1..N]\}$ is the set of conjuncts, an *unsatisfiable core* of the formula is a subset $\mathbb{C}' \subseteq \mathbb{C}$ such that $\wedge_{i \in \mathbb{C}'} \phi_i$ is unsatisfiable.

This can be efficiently computed as shown in a simple way by encoding the problem so that each conjunct is represented by a literal, and solve under the assumption of these literals. An unsatisfiable core can then be computed by performing conflict analysis on the final conflict in the solver taking care to produce a conflict set which only contain such literals. This feature is built-in into the MiniSat SAT solver [ES04] used in MathSAT, and its usage for unsat core extraction has been described in [CGS07, ANOR08]. The difference from those works is that our unsat cores are subsets of the conjuncts of the formula. These can be expected to be much fewer than the number of clauses, and so should deliver acceptable performance and avoid the bottlenecks reported in [ANOR08].

An unsatisfiable core can be seen as a reason for unsatisfiability of a formula, it is often the case that an unsatisfiable core of a formula is much smaller than the formula itself. This gives some hope that the unsatisfiable core will be more effective in deducing that future formulae are unsatisfiable than just the information that the entire formula was unsatisfiable.

If a formula contain the unsatisfiable core of a previous formula we can deduce that this is satisfiable without needing to solve it. We do this by simply check if the set of conjuncts in the formula contain any of the previously discovered unsatisfiable cores. This is both straightforward to implement and can be computed efficiently. It also gives good performance in practise in terms of the number of formulae which can be deduced as unsatisfiable without solving them as we shall see later in the experimental evaluation in section 7.8.

7.6 Non-singleton sets

In the cases where the current set of formulas contain more than one formula, we should try to take advantage of this in order to improve performance. For three medium-sized to large microcode programs the simulator generates sets of formulae with cardinalities as can be seen in figure 7.3. In total, there are 93 non-singleton sets with between 100 and 1000 instances, and 11 sets with over 1000 instances.

To take advantage of this, we would like to make the solver aware of all formulae beforehand. In this way we may be able to satisfy more than one formula at a time, and also reuse learnt information to discover that several formulae in the set are unsatisfiable. One way of achieving this is shown in a simple algorithm 7.2 we will call Multiple Similar Properties SAT (MSPSAT). Here we create one fresh predicate (boolean variable) p_i for each formula ϕ_i and give the solver the formula

$$\bigwedge_i p_i \leftrightarrow \phi_i$$

To solve ϕ_i, we solve under the assumption p_i. Should it be satisfiable under this assumption, we can easily check which of the other formulae are also satisfied by the same model by checking the truth assignment for the other fresh variables. The algorithm iteratively picks one unsolved formula as a goal and solves under the assumption of the corresponding fresh variable. If it is satisfiable it checks if any other unsolved formulae are satisfied by the same model and discharges all satisfiable formulae.

Algorithm 7.2: Guided MSPSAT

```
Input: $\phi_1, \phi_2, \ldots, \phi_N$
$P \leftarrow \emptyset$
$\phi \leftarrow \top$
foreach $i \in [1..N]$ do
    $p_i \leftarrow$ fresh predicate;
    $P \leftarrow P \cup \{p_i\}$;
    $\phi \leftarrow \phi \wedge (p_i \Leftrightarrow \Phi_i)$;
end
Sat $\leftarrow \emptyset$
Unsat $\leftarrow \emptyset$
while $P \neq \emptyset$ do
    $p_i \leftarrow$ some element in $P$
    if $\phi \wedge p_i$ satisfiable with model $\mu$ then
        Sat $\leftarrow$ Sat $\cup \{\phi_j \mid \mu \models \phi_j\}$
    else
        Unsat $\leftarrow$ Unsat $\cup\, \phi_i$
    end
    P $\leftarrow P \setminus$ (Sat $\cup$ Unsat)
end
return Sat, Unsat
```

7.6.1 An alternative MSPSAT algorithm

In the MSPSAT algorithm described above, in each iteration a candidate formula is picked to check satisfiability on. This may limit the heuristics of the solver however. An alternative is to keep track of all formulae not yet solved, and ask the solver to solve at least one of them, any one. This can be accomplished by giving the solver the disjunction of all formulae left to solve. The possible advantages of this is twofold; First, the solver is free to find a model for any of the formulae left to solve. Second, when all remaining formulae are unsatisfiable, it can be detected with a single solver call.

Algorithm 7.3: Unguided MSPSAT

```
Input: $\phi_1, \phi_2, \ldots, \phi_N$
1  Unsolved $\leftarrow \{\phi_i \mid i \in [1..N]\}$
2  Sat $\leftarrow \emptyset$, Unsat $\leftarrow \emptyset$
3  while Unsolved $\neq \emptyset$ do
4      $\phi \leftarrow \bigvee_{\phi_i \in \text{Unsolved}} \phi_i$
5      if $\phi$ satisfiable with model $\mu$ then
6          $S \leftarrow \{\phi_i \mid \phi_i \in \text{Unsolved}, \mu \models \phi_i\}$
7          Sat $\leftarrow$ Sat $\cup\, S$
8          Unsolved $\leftarrow$ Unsolved $\setminus S$
9      else
10         Unsat $\leftarrow$ Unsat $\cup$ Unsolved
11         Unsolved $\leftarrow \emptyset$
12     end
13 end
14 return Sat, Unsat
```

A disadvantage may be that the search for satisfiable formulae is unguided. This may cause a great deal of search, especially with a solver that assigns false on decision variables, as is usually done in MathSAT. A possible remedy may be to set up the decision heuristics of the solver so that it always picks true for the relevant atoms rather than false.

7.7 Basic parallelism

Another approach which comes to mind when there are a large number of formulae to solve is to solve them in parallel. Parallel computers are now commonplace, even on desktops, and this trend towards increasingly parallel computers can be expected to continue for some time on workstations and servers. Considering that in several cases, MicroFormal may provide the solver with hundreds or even thousands of formulae, attempting to take advantage of the growing number of cores in moderns computers may be an interesting avenue to pursue.

The approach we take is to treat the problem as a so-called *embarrassingly parallel* problem, i.e. a problem which can be divided into several sub-problems which have little or no interaction. Here, we divide the set into k subsets and solve each subset in parallel. Each subset can be solved on different cores on the same computer, or distributed on different computers in a cluster if k is large. If the computations were independent, we might expect performance to improve by a factor approaching k . But if we either reuse solver information or use MSPSAT, a linear speedup is not likely to be achievable. We lose some of the benefit of reuse, while gaining some by parallelism. Which outweighs the other is difficult to say, some experiments will be presented in section 7.8.3.

If we compare parallelism with solving each formula in the set, while reusing solver information and a particular reset interval, then it is clear that parallelism is advantageous when the number of formulae in the set is greater than the reset interval. If we have the reset interval r we can divide the set into “work packages” of r instances each, and solve the work packages in parallel on the available cores/CPUs.

7.8 Experimental evaluation

We now turn to an experimental evaluation of the techniques proposed in this chapter. Except where explicitly noted, all experiments were carried out on a machine with dual Intel Xeon E5430 CPUs running at 2.66 GHz using 32 GB of RAM running Linux.

Table 7.1: MicroFormal test sets

Program	Instances	Satisfiable	Unsatisfiable
Program 1	52933	44359	8574
Program 2	5468	4341	1127
Program 3	28962	13757	15205

Most of the experiments are run on instances coming from three nontrivial microcode programs. For these three, MicroFormal was instrumented to dump all instances to files in SMT-LIB format, and produce a log describing how these instances were created. In this thesis the programs will be called "program 1", "program 2" and "program 3", table 7.1 gives the number of formulae generated in each of these three MicroFormal runs. A test bench has then been created which can replay the solver calls in these three runs of MicroFormal, which makes it easy to experiment with different strategies and instrument the system to extract interesting information. In order to emulate the behaviour of MicroFormal, when solving a formula it is first loaded into memory in a separate data structure to avoid measuring the time taking for parsing formulae. From this data structure the MathSAT API is called, creating and solving formulae simulating the in-memory usage in MicroFormal as closely as possible without actually running MicroFormal.

Apart from the techniques described in this chapter, these experiments were performed with minimal preprocessing of formulae and translation into SAT rather than DPLL(T), since for the instances that are generated in MicroFormal, this seems to deliver better performance overall. For the instances taking the most execution time, more aggressive preprocessing techniques can be effective, but the total execution time is dominated by a large number of trivial instances, and the preprocessing normally used in MathSAT seems to be too expensive to be used here.

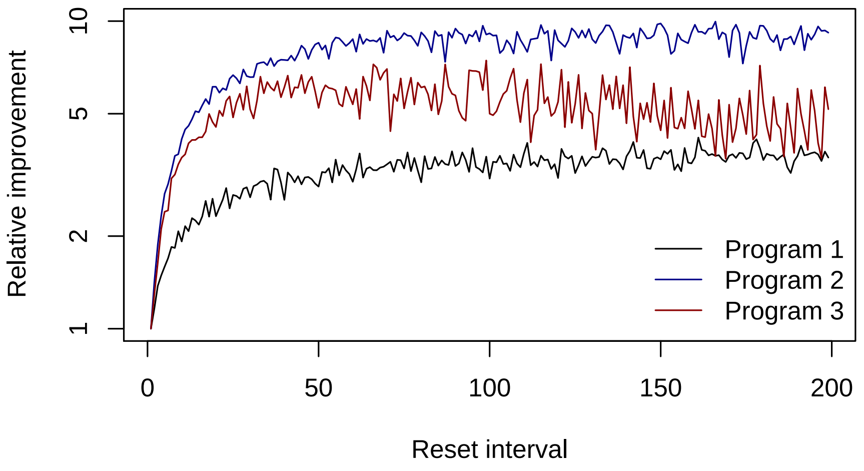


Figure 7.5: Effect of reset interval on singleton calls

7.8.1 Reuse of information

We start by investigating the effect of fixed reset strategies on singleton sets. For these experiments, we solve only singleton sets, skipping over the other calls completely. The result on the three programs are summarised in figure 7.5. It shows the relative improvement of reusing solver information compared to solving each formula in isolation. The horizontal axis shows the reset interval, that is how frequently all learnt information is thrown away. A reset inteval of 1 corresponds to solving each formula in isolation. From the figure, it is clear that there is a positive effect of reusing solver information. For program 1 the best improvement is a factor of 4 (at a reset interval of 161), and for program 2 the best improvement is a factor of almost 10 (at reset a interval of 169). Lastly for program 3 the best improvement is a factor of 7.4 (at a reset interval of 99).

We can also see that the exact reset frequency is not critical. For program 1 and program 2, there is only a minor difference between different reset intervals above 50. For program 3, the trend is similar but the data

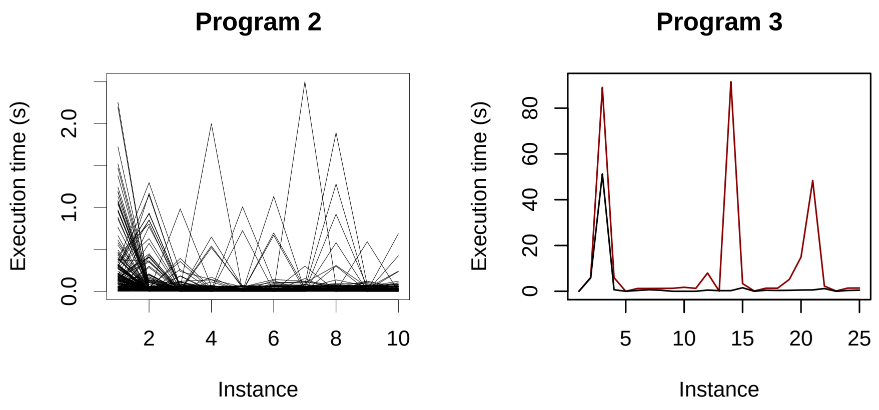


Figure 7.6: Effect of reset interval on individual singleton calls

appears to be more noisy. This is due to some outliers among the instances to be solved, which are both large and significantly different from any of the others. These cause significant overhead when these instances are retained in the solver and we attempt to solve fresh instances. Performance depends on being able to divest the solver of this irrelevant information as soon as possible, but with a fixed reset interval how quickly this happens is largely due to chance. To avoid this, we will choose a reset interval of 25 for future experiments, which although shorter than what is indicated as the optimal, should on the other hand handle such outliers better. With this reset interval, the improvement for these three programs is a factor of 2.7, 6.7 and 4.9 respectively.

To see some details of the effect of reusing solver information, we will study the individual solver calls for program 2. In figure 7.6, we show the execution time for all individual instances. Instance 1 is all instances immediately after reset, instance 2, is the next instance and so on. In this case a reset interval of 10 was chosen, to keep the figures a little more readable. The results for longer reset intervals is similar. Given that reusing information works so well, it might be expected that all but the very first instance after reset will be trivial. This is not quite the

Table 7.2: Fixed vs optimal reset strategy. Execution times in seconds

Set	Freq. 25	Freq. best	Optimal
Program 1	275.4	179.2	153.1
Program 2	65.6	44.0	40.8
Program 3	1724.0	1125.0	–

case, but the other solver calls are noticeably faster. The reason it is not always the case is that a particular subsequence from reset until the next reset may by chance start with trivial instances, and for the first slightly harder instance solving it is harder than the rest. After that one instance, the solver will typically have learnt the relevant information to be able to solver the remaining instances more quickly.

In the same figure we can see the main cause for the "noisy" results of program 3 in figure 7.5. The figure shows a sequence of singleton sets from a reset point and 25 sets onwards. Although reusing information helps for other singleton sets, for this particular subsequence it does not. The performance penalty also varies greatly depending on where the resets are performed in relation to the problematic instances.

To get an idea of how efficient a simple fixed reset strategy is, it can be compared with an "optimal" strategy. An optimal strategy is a sequence of reset intervals indicating where resets should take place. To keep the time needed to compute these reset strategies reasonable, we have limited the maximal reset interval in any such strategy to 200 instances, without such a limit computing the optimal strategy will be prohibitively time-consuming. The result is summarised in table 7.2. For comparison, we also list the execution times of running with two fixed reset frequencies, namely resetting every 25 instances and also the best reset fixed frequency found for each program. The difference between the best fixed reset frequency, and the optimal reset strategy is not very large, indicating that using a fixed frequency may be a suitable heuristic. For program 3, computing the optimal reset strategy turned out to be too resource intensive to complete in a reasonable time despite the limit on maximal reset interval. Neither lower limits on the maximal reset interval nor manual effort failed to find a strategy with an execution time less than 1100 seconds.

To check if reuse of solver information is usable outside of MicroFormal, the technique has also been applied to the instances[2] in SMT-LIB coming from the SAGE tool [GLM07]. Out of 12 sets of instances, a fixed reset strategy of resetting every 25 instances helped in all but two sets. In one of the two, execution time was comparable (332 versus 334 seconds). In the other reusing solver information used 65 seconds versus 11 seconds for solving each instance individually. The added time is taken up in two instances which take considerably more time than the rest. Full results for these sets of instances can be found in figure 7.7, where total execution time (in seconds) for each set of instances is reported. Although the improvement is not as large as for the three microcode programs seen earlier, there is still a fairly clear improvement, and, indeed this improvement is statistically significant ($p = 0.016$).

7.8.2 Non-singleton sets

For the cases where MicroFormal generates multiple formulae to solve there are several choices, we will look at a few of them as listed below:

1. Solve them in the same way as single formulae. There might not after all be any need to treat these instances any different from any other.
2. Solve them as with single formulae, but with an infinite reset interval. The motivation is that similarity can be expected to be better within each set than between singleton instances since all instances in a set have been generated at a specific point in symbolic execution.

[2]Available in the SMT-LIB in QF_BV/sage

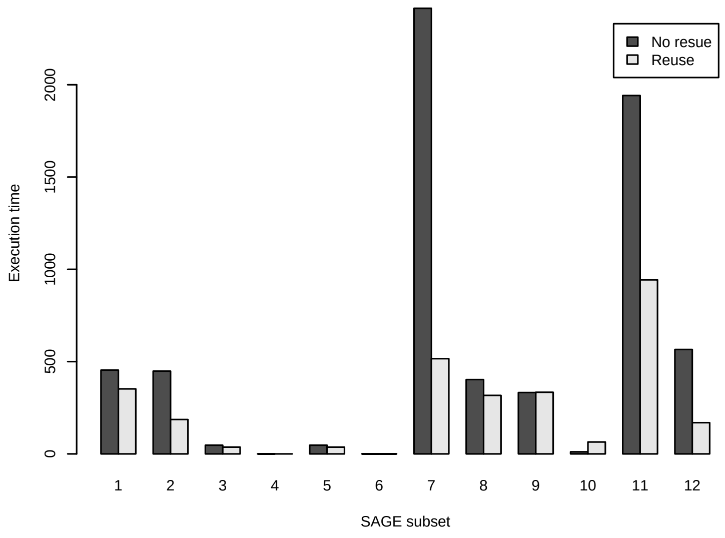


Figure 7.7: Effect of reusing solver information on SAGE instances. Execution times in seconds

3. Solve them with Guided MSPSAT.
4. Solve them with Unguided MSPSAT.
5. Use parallelism, dividing non-singleton sets into a number of subsets and solving each in parallel using one of the above techniques.

As a baseline, let's look at the performance when treating each instance as a singleton, disregarding that more than one instance is known a priori. The results are shown in the first row in table 7.3. Using the two MSP-SAT algorithms, we get the results in the two last rows of the same table. We can see a significant improvement over solving each formula individually. For comparison, we also include the execution time when solving all instances reusing solver information using a reset interval of 25, and also when resetting only in between sets of instances. We can see that using a reset interval of 25 gives worse performance than using the Guided MSPSAT algorithm, so there seems to be some value in treating these sets in a special way. For these three programs at least there does however not seem to be an advantage with MSPSAT when compared to using a

Table 7.3: Performance of the MSPSAT algorithms

Method	Program 1	Program 2	Program 3
No reuse	104459.86	1722.31	55539.64
Reset (25)	9104.31	217.13	5434.52
Reset in-between	4485.51	243.91	2694.61
Guided MSPSAT	6064.98	278.00	2826.98
Unguided MSPSAT	>200000.00	292.39	9824.27

separate solver instance for non-singleton sets which is reset in-between every set. Indeed, the latter technique has a small advantage over the others. Unguided MSPSAT appears to work poorly, and a closer look reveals that for large sets, the solver performs very poorly. This is not visible in program 2, because this example doesn't contain really large sets, all have cardinality ≤ 96.

In a sequence of formulae, there can be many sets of low cardinality, and in these cases treating them as if they were singleton solver calls may be useful. As we have seen, reuse of solver information can be very benficial, and for very small sets the penalty of initially solving without learnt information from previous formulae may dominate total solving time for the set. A simple heuristic may be to check the cardinality of the set, and if it is below some threshold, treat each formulae in the set as a singleton set. An experiment with varying thresholds can be found in figure 7.8, where total execution time is plotted as a function of the threshold. As can be seen there seems to be some improvement, but there is also quite a bit of noise making it difficult to quantify the gain. A smoothed line (using local fitting) is added to make the trend more clear.

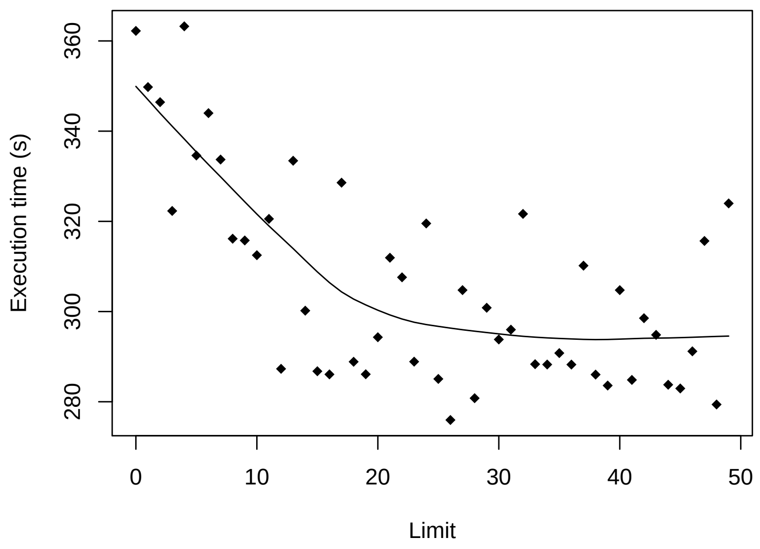


Figure 7.8: Effect of MSPSAT threshold on program 2

7.8.3 Parallelism

Using parallelism to solve non-singleton sets appears at first glance to be very promising way of improving performance. Non-singleton sets can contain hundreds or even thousands of instances, so the problem is trivially parallelisable. But since instances are highly similar one to the other, simply dividing a non-singleton set into a number of subsets and solving them independently will interfere will the advantages of reusing learnt information in solving. An example can be seen in figure 7.9 where one single non-singleton set containing 580 instances is solved in parallel, using between 1 and 8 cores on the same machine. The figure shows the wall-clock execution time taken to solve all instances, the speedup versus using a single core, and the Karp-Flatt metric [KF90], defined as

$$\frac{\frac{1}{s} - \frac{1}{p}}{1 - \frac{1}{p}}$$

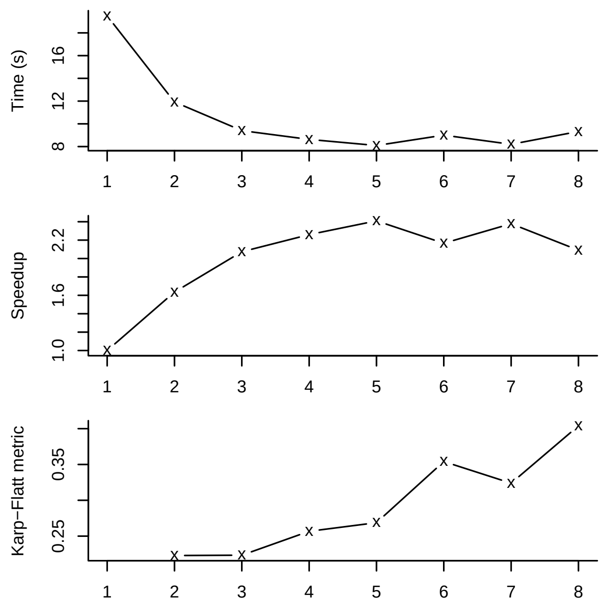


Figure 7.9: Effect of parallelism on a simple non-singleton set

where s is the speedup and p the number of cores used. An advantage of this metric is that since the fraction will remain constant if the speedup is ideal, it is easy to spot any issues with scaling. If the fraction increases, extra overhead is introduced as the number of cores is increased. Although using 2 cores shows a nice performance improvement, above 3 cores the improvement is much less than might be expected, and it quickly seems to stagnate at a speedup of around a factor of 2. This is due to the similarity of formulae and that once one have been solved, solving each of the rest is significantly cheaper. Dividing the set into several subsets means we need to incur the cost of solving the "first" instance many times. So it seems that for the instances generated in MicroFormal, the problem of solving non-singleton instances is not as trivially parallelisable as it would at first appear. The benefit of reusing solver information appear to outweigh to some extent the benefit of solving formulae in parallel.

Table 7.4: Hit-rate of unsatisfiable core caching

Program	Singletons	Hit-rate	Total	Hit-rate
Program 1	1059	64.3%	8574	90.3%
Program 2	534	50.9%	1127	69.7%
Program 3	3283	16.2%	15205	79.0%

7.8.4 Unsatisfiable cores

To test the effectiveness of unsatisfiable core caching, we focus on singleton solver calls for simplicity. For each of the three programs, we collect the sequence of unsatisfiable singleton instances. Those are then run in sequence, collecting each unsatisfiable core. If the cores that have been computed in the past can be used to deduce that the current formula is unsatisfiable, that is seen as a cache hit in a hypothetical unsatisfiable core cache and skipped. The hit-rate for the three programs can be seen in table 7.4. Two measurements have been made, both for only singleton sets, and for all instances in each sequence. The table shows the number of unsatisfiable instances in each case, as well as the potential hit-rate, the amount of instances which would not need to be solved. We can see that for singletons sets, there can be a considerable reduction in number of unsatisfiable instances. Taking all instances into account, the hit-rate is even more impressive. However, the hits which are found in non-singleton sets are typically caused by another instance within each set. Because of this the potential gain may be reduced somewhat.

In table 7.5, we can see the effect of removing the singleton sets which can be detected to be unsatisfiable would have on performance. For each program, three execution times have been measured, all with a reset interval of 25 for singletons. “Original” is the time when all instances are used, for “Singletons” all singleton instances where a previous unsatisfiable core from a singleton instance is enough to show unsatisfiability have been

Table 7.5: Performance on singleton sets when unsat core hits are removed

Program	Original	Singleton	Total
Program 1	6049.2	5987.4	5914.6
Program 2	332.3	333.2	323.1
Program 3	4657.9	4892.8	4693.6

removed. Lastly for "Total" all unsatisfiable instances where any previous unsatisfiable core could be used to deduce unsatisfiability have been removed. Although there is an effect on total execution time for all solver calls, the effect seems to be surprisingly small. Given the large hit-rate of caching unsatisfiable cores, a better improvement might have been expected. An explanation for this is that when reusing solver information, the unsatisfiable core is likely to be retained by the solver if it was discovered in the recent past, after the last reset. And it is often the case that the solver "horizon" includes enough learnt information to easily discover unsatisfiability. The case of program 3 deserves some comment. Here, performance actually *degrades* (even if only slightly) when some instances are removed. This is highly surprising; We might think that solving less instances should require less time, not more. The explanation can be found in the results on reuse of solver information shown in figure 7.5. The measured execution time shows significant noise when varying reset interval, and when some instances are removed we end up resetting the solver at more unfortunate times during execution. This highlights the disadvantage of a static reset strategy, which although it does provide a clear benefit compared to not reusing solver information, is not optimal.

It should be noted that what is measured here is just the effect on solving time, how the execution time of the rest of MicroFormal is affected is not known as those experiments have not yet been performed.

Table 7.6: Ample performance summary (execution times in seconds)

Solver	Type	Median	Mean	Stddev
Prover	Singleton	1072.14	2887.13	5973.29
	Non-singleton	389.01	2264.52	4432.13
	Ample	2412.00	6282.90	10316.34
MathSAT	Singleton	98.48	289.05	704.25
	Non-singleton	233.25	975.24	1751.98
	Ample	997.00	2183.03	2842.62

7.8.5 Ample experiments

As a final experiment the impact of MathSAT on the *Ample* tool is evaluated. Ample (which stands for Automatic Microcode Path Logic Extraction) is a tool in MicroFormal used for then generation of execution paths for dynamic testing, and this will be used for experimental evaluation in this section. For this evaluation 32 different microcode programs have been selected to be representative of small, medium, and large programs. For each, Ample is run with the Prover SAT solver, and with MathSAT. In MathSAT, reusing of solver information was used with a fixed reset frequency of 25, and for non-singleton sets Guided MSPSAT was used. For Prover, singleton sets were solved individually, and non-singleton sets were solved using the SSAT algorithm. The tool was run on machines with Intel Xeon 5160 CPUs running at 3 GHz and 32GB RAM running Linux, and the execution times of solver calls, other processing, total execution time and memory usage was measured. In these experiments, in no case was memory usage an issue. For this experiment, unsatisfiable core caching is not used. This is simply because the feature requires some changes to Ample which had not been implemented at the time these experiments were performed.

The results are summarised in table 7.6, where the median, mean, and

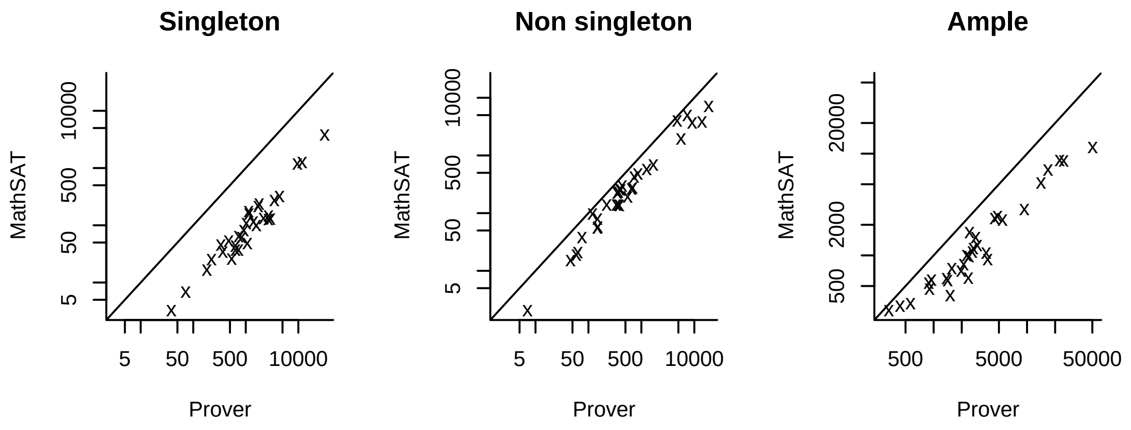


Figure 7.10: Ample execution performance

standard deviation for the total execution time of singleton instances, MSPSAT/SSAT as well as total execution time is presented. For every program, the performance of MathSAT is better than that of Prover, and for total execution time the improvement is at worst a factor of 1.17, at best a factor of 4.43, and overall the improvement is a factor of 2.88. Not surprisingly, the improvement is statistically significant ($p = 9 \cdot 10^{-9}$). As the experiments on non-singleton sets showed, simply reusing solver information, resetting the solver in-between each set may improve performance further. At the time of writing, this has not been tried on these 32 microcode programs.

It should be noted that the difference on non-singleton sets are not necessarily due to the different algorithms (MSPSAT versus SSAT) being used, since two completely different solvers are used for the comparison.

Chapter 8

Extensions to SMT

There are many real-world problems which deal with resource consumption. Some examples may be

– Placing and routing in VLSI design [LMS06]. Placing deals with placing blocks on a die in the optimal way, reducing die area and maximising routability. Routing deals with laying out wires between blocks reducing delay or interference between wires.
– Configuration design [MF89, WS97]. The configuration design problem is loosely defined as assembling a system built with components which have interdependencies such that the overall system is feasible and optimal according to some criteria.
– Another application area may be planning under resource constraints [Koe98], where it is desirable to not only find some plan, but find a plan which minimises resource usage.

In this chapter we will discuss some extensions of SMT which can allow us to solve these and similar problems. We start with some preliminaries peculiar to this chapter, then introduce a new theory, the theory of costs, and then describe how optimisation of cost functions can be carried out. After this we show how the theory can be used to extend the problems of Max-SAT and Pseudo-Boolean Optimisation into SMT, finishing up with some notes on implementation issues, and an experimental evaluation of the approach[1].

8.1 Preliminaries

The optimisation problem can be stated as minimising an objective function $f(x)$ over some formula ϕ

$$\begin{aligned} &\min f(x) \\ &\text{subject to } \ \phi \end{aligned}$$

The corresponding maximisation problem $\max f(x)$ can be restated as the minimisation problem $\min -f(x)$, so here we will focus on the minimisation problem only.

Informally, the objective function is a function that given a truth assignment over the predicates in the formula computes an integer value. In order to tie integer values to predicates, we need a mapping from truth values to integer numbers. One such mapping is the characteristic function.

Definition A *characteristic function* $\mathbf{1}_A : A \to \{0, 1\}$ for a set A is a function such that

$$\mathbf{1}_A(x) = \begin{cases} 1 & x \in A \\ 0 & x \notin A \end{cases}$$

This allows us to introduce a mapping from literals to the set $\{0, 1\}$. A polynomial objective function over a set of literals $\mathbb{L} = \{l_1, l_2, \ldots\}$ can then be stated as a function over the model μ

$$f(l_1, l_2, \ldots) = \sum_i c_i \prod_j \mathbf{1}_\mu(l_j)$$

Any polynomial objective function can be linearised, that is restated as an equivalent linear function, by introducing fresh predicates. For any product

[1]The work presented here will also occur in [CFG+10]

$\mathbf{1}_\mu(l_1) \cdot \mathbf{1}_\mu(l_2)$ we can introduce a fresh predicate p, conjunct $p \Leftrightarrow l_1 \wedge l_2$ to the formula and replace the product with $\mathbf{1}_\mu(p)$. In a similar way, we can convert all linear objective functions into functions where all literals l_i are positive, that is are predicates by introducing a fresh predicate p for every negative literal $\neg p_i$ and conjunct $p \Leftrightarrow \neg p_i$ to the formula and replace $\neg p_i$ with p in the objective function. Further, we can scale the function so that all constants are positive. If there exists a negative weight $-c$, we can simply add c to all weights. After eliminating all negative weights in this way, we can remove the products having zero weights since they do not affect the objective function. Without loss of generality, we will therefore only consider linear objective functions, with positive literals and positive weights.

Definition A linear *pseudo-boolean function* is a function for a model μ over a set of predicates $\mathbb{P} = \{x_1, x_2, \ldots\}$ to a integer number

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n} c_i \mathbf{1}_\mu(x_i)$$

where $c_1, c_2, \ldots$ are positive integers.

An alternative way which is common in the literature of expressing such functions when boolean variables have the domain $\{0, 1\}$ is $\sum_i c_i x_i$. We choose the earlier description so that it fits better with the formalism in the other chapters.

8.2 A theory of costs

In this work we will encapsulate the constructs necessary for describing objective functions and constraints over such functions in a theory, which we will call the theory of *costs*, and which allows for highly efficient reasoning on such functions and constraints. The theory of costs consists of

– A set of variables $\mathcal{V}_C$,
– The set of positive integers $\mathbb{Z}^+$
– A ternary predicate symbol such that if $v \in \mathcal{V}_C$ and $c \in \mathbb{Z}^+$, $d \in \mathbb{Z}^+$, then $\mathsf{incur}(v, c, d)$ is a predicate
– A binary predicate $\lessdot$ such that if $v \in \mathcal{V}_C$ and $b \in \mathbb{Z}^+$ then $v \lessdot b$ is a predicate.

Informally, $\mathsf{incur}(v, c, d)$ incurs the cost c on the variable v, and $v \lessdot b$ gives an upper bound b for the variable. The total incurred cost on a variable is the sum of all $\mathsf{incur}(v, c_i, d_i)$ predicates currently assigned to true. This means the semantics can not be defined in the traditional compositional way, instead we must define the semantics given a truth assignment seen as a set of literals

$$\mu = \bigcup_{v \in \mathcal{V}_C} \mu_v$$

where each μ_v is a set of literals concerning the variable $v \in \mathcal{V}_C$

$$\begin{array}{rcl} \mu_v & = & \{\mathsf{incur}(v, c_1^1, d_1^1), \ldots, \mathsf{incur}(v, c_n^1, d_n^1)\} \\ & \cup & \{\neg\mathsf{incur}(v, c_1^2, d_1^2), \ldots, \neg\mathsf{incur}(v, c_m^2, d_m^2)\} \\ & \cup & \{v \lessdot b_2^1, \ldots v \lessdot b_o^1\} \\ & \cup & \{\neg(v \lessdot b_2^2), \ldots \neg(v \lessdot b_p^2)\} \end{array}$$

where $b_i^j, c_i^j \in \mathbb{Z}^+$ and $d_i^j \in \mathbb{Z}^+$. We can give semantic to each subset μ_v individually since the subsets do not share any variables. The incurred cost on the variable v given the truth assignment μ_v can now be defined as

$$\mathsf{incurred}(v, \mu_v) = \sum \{c \mid \mathsf{incur}(v, c, d) \in \mu_v\}$$

The set μ_v is inconsistent iff there exists a $(v \lessdot b) \in \mu_v$ such that $b \leq \mathsf{incurred}(v, \mu_v)$. If there exists at least one $(v \lessdot b) \in \mu_v$, then we can define the *residual* of v as the smallest difference $\mathsf{incurred}(v, \mu_v) - b$. The current truth assignment μ would then be inconsistent if there exists a residual less than 1.

It may seem odd that the incur predicate is ternary, since we do not use the third argument in the semantics. A binary predicate incur(v, c) would seem to be sufficient. The reason for the predicate being ternary rather than binary is in order to be able to incur the same cost on a variable v multiple times, as demonstrated in example 8.1.

Example 8.1

If the incur predicate would be binary, the formula incur$(v, 3) \wedge$ incur$(v, 3)$ is equivalent to incur$(v, 3)$ and incurs a cost 3 on the cost variable v. It is not possible to incur the same cost twice in this case. Since we have ternary incur predicates, we can use unique third arguments to incur the same cost twice as in incur$(v, 3, 1) \wedge$incur$(v, 3, 2)$ which incurs 6 on the cost variable. ∎

8.2.1 Computing minimal conflict sets

Finding minimal conflict sets is according to Aloul et al [ARSM07] an instance of the knapsack problem and therefore NP-complete, but in their text what constitutes minimal can be interpreted in two different ways. In our case, the minimal conflict set computation problem can be stated equivalently as follows: Assuming we have a set $C \subset \mathbb{Z}^+$ and some bound $b \in \mathbb{Z}^+$, the problem is to solve

$$\begin{array}{rl} \min & |C'| \\ s.t. & C' \subseteq C \wedge \sum C' \geq b \end{array}$$

A basic greedy algorithm taking the largest elements in C will produce a minimal subset, and it is easy to see why. If there existed a smaller subset C'' whose sum also exceeded the bound, then we can replace the smallest element in C'' with the largest element in $C\setminus C''$ and end up with a set of the same cardinality which also violates the bound. This can be repeated until $C \setminus C''$ doesn't contain any element larger than the smallest element in C'', which is the same set produced by the greedy algorithm. Computing the conflict set which exceeds the bound with the smallest margin, however, is equivalent to the minimisation knapsack problem (an overview of knapsack problems can be found in [KPP04]) and a more difficult problem. Similarly, computing all minimal subsets is also more complicated, but here will limit ourselves to finding a single minimal conflict set.

In our case, computing minimal conflict sets is therefore possible in $O(n \log n)$, assuming C is not being kept sorted. If it is, computing conflict sets would be linear in the number of elements of the conflict set.

8.2.2 Deduction

Deducing truth values for unassigned literals is straightforward. If there exists an unassigned predicate $\mathsf{incur}(v, c, d)$ for some variable v and the residual under the current truth assignment is r, then we can deduce $\neg\mathsf{incur}(v, c, d)$ iff $r < c$.

8.3 Optimisation

Using the theory of costs, we can encode an objective function

$$f(l_1, , \dots, l_n) = \sum_{i=1}^{n} c_i \mathbf{1}_\mu(l_i)$$

as a small formula in the theory of costs by creating a fresh cost variable v and create the formula

$$\bigwedge_{i=1}^{n} \mathsf{incur}(v, c_i, i) \Leftrightarrow l_i$$

Solving the optimisation problem in the theory of costs can now be done by conjuncting the encoding of the objective function to the formula, and use the simple algorithm outlined in algorithm 8.1. This algorithm relies on

Algorithm 8.1: Basic linear search based optimisation

```
Input: Formula $\phi$ with a cost variable $v$
1  if $\phi$ satisfiable with cost $b$ then
2      opt $\leftarrow b$
3      $\phi \leftarrow \phi \wedge v \lessdot b$;
4      while $\phi$ satisfiable with cost $b'$ do
5          opt $\leftarrow b'$
6          $\phi \leftarrow \phi \wedge v \lessdot b'$
7      end
8      return opt
9  end
10 return unsatisfiable
```

an off-line refinement of the best known bound, and can be implemented outside the solver rather than making modifications to the solver internals. Since we incrementally add new conjuncts to the formula until it becomes satisfiable it is trivial to use an incremental solver, reusing all learnt information. A simple alternative is to do this on-line, i.e. to let the cost theory solver add this constraint during search when a model has been found. Then the SMT solver can do conflict analysis, backtrack and continue searching. This is however not yet supported in MathSAT.

8.3.1 Dichotomic search

If a formula contains a set of incur predicates $\{\mathsf{incur}(c, c_1, d_1), \ldots \mathsf{incur}(c, c_N, d_N)\}$, we can conclude that the optimal cost must be in the interval $[0, \sum_{i=1}^{n} c_i]$ if the formula is satisfiable. With the linear search algorithm, we can in the worst case iterate $\sum_{i=1}^{n} c_i$ (or 2^n) times in the loop before finding the optimum. An alternative would be to "short-circuit" this search by guessing tighter bounds, which may reduce the number of formulae to solve in order to find the optimum.

This type of search algorithm is often called *dichotomic search*, and is outlined in algorithm 8.2. Instead of simply searching for a cost lower than the lowest cost found so far, we keep track of the interval $[l, u]$ in which we know the optimum to lie, and we guess that it is possible to short-circuit the search by picking some cost b in the interval, and check if the formula conjuncted with $v \lessdot b$ is satisfiable. If our guess was incorrect the formula is unsatisfiable, and we can tighten the interval to $[b, u]$. If the formula was satisfiable, we can tighten the interval to $[l, b]$. We iterate guessing and tightening the known interval like this until the optimal has been found, i.e when the interval contains a single value.

Algorithm 8.2: Dichotomic search based optimisation

```
Input: Formula $\phi$ with cost variable $v$
if $\phi$ satisfiable with cost $b$ then
    $\langle l, u \rangle \leftarrow \langle 0, b \rangle$
    while $l \neq u$ do
        $r \leftarrow$ some value in $(l, u]$
        if $\phi \wedge v \lessdot r$ satisfiable with cost $b$ then
            $u \leftarrow b$
        else
            $l \leftarrow r$
        end
    end
    return $l$
end
return unsatisfiable
```

Using the general dichotomic search algorithm outlined here, it is possible to implement several different search strategies. We can observe that the linear search algorithm can be seen as a special case of the dichotomic search algorithm where we always pick the largest cost in the interval where we know the optimal lies. It should also be noted that this algorithm can easily be implemented in an incremental solver reusing all information learnt from solving in all previous iterations. In each iteration we solve the same formula, assuming some additional atom to be true.

In the current implementation, we use *binary* search. This is one variant of dichotomic search where the guessed bound in line 5 is always the midpoint in the interval.

8.3.2 Lower bounds on the cost

If we can compute a lower bound for the cost, we can use this to prune search. Given that we have an upper bound u which we are trying to satisfy, and the current partial interpretation gives us a cost of b. Should we know a lower bound on the cost l on the cost for any complete model, then we can use this in two situations

1. If the total cost of unassigned incur predicates are less than $b - l$, we have a conflict
2. If one or more of the unassigned incur predicates are needed to expand the current interpretation to a model that satisfies the lower bound, then they can be deduced.

In order to support this, we make the cost theory take advantage of negative $\lessdot$ literals. A negated $v \lessdot b$ means that the cost must be at least b.

One possible source of lower bounds comes from the dichotomic search algorithm. Should we find that the formula $\phi \wedge v \lessdot b$ is unsatisfiable we can conclude that $\phi \wedge \neg(v \lessdot b)$ has the same optimum as ϕ itself. We can update the formula in this way on line 10 in the algorithm.

8.4 Pseudo-boolean constraints

The theory of costs can be used for more than simply encoding an objective function, one example of this is encoding Pseudo-Boolean constraints. A Pseudo-Boolean constraint is a constraint on $0-1$ variables

$$\sum_i a_i x_i \leq b$$

The Pseudo-Boolean Optimisation problem is the problem of optimising an objective function over a set of Pseudo-Boolean constraints. Recently there have been proposed [BBR09] polynomial translations from pseudo-boolean constraint to CNF, but a pseudo-boolean constraint remains a more compact representation and may be more efficient to reason with. The theory of costs can be used to represent pseudo-boolean constraint in SMT, by using one cost variable v for each pseudo-boolean constraint and describe it as a simple formula over incur predicates $\bigwedge_i x_i \Leftrightarrow \mathsf{incur}(v, a_i)$ conjuncted with a bound $v \lessdot b$.

The Pseudo-Boolean satisfiability problem consists of deciding satisfiability for a conjunction of pseudo-boolean constraints, and the Pseudo-Boolean optimisation problem can be stated as

$$\begin{array}{l} \min f(x) \\ \text{subject to } \ \bigwedge_i \sum_j a_{ij} x_{ij} \leq b_i \end{array}$$

With the theory of costs we can represent this formula efficiently in MathSAT by encoding each pseudo-boolean constraint as outlined above and the objective function as we have seen in the previous section.

Proposition 8.4.1 *The problem of satisfiability in* SMT(Cost) *is equivalent to the pseudo-boolean satisfiability problem*

Proof One possible proof is a constructive proof. We need to show that any pseudo-boolean constraint can be translated in polynomial time into a formula in SMT(Cost), and that any SMT(Cost) formula can be translated in polynomial time into a set of pseudo-boolean constraints. We start with the pseudo-boolean to SMT(Cost) translation

It is enough to show how a single pseudo-boolean constraint can be translated since all constraints can be translated independently from one another. Given a pseudo-boolean constraint

$$C \hat{=} \sum_i c_i l_i \geq c$$

we create a fresh cost variable v and create the formula $E \hat{=} \bigwedge_i l_i \Leftrightarrow \mathsf{incur}(v, c_i, i)$ corresponding to the linear function and $B \hat{=} \neg v \lessdot c$ for the lower bound. For a pseudo-boolean formula with a set of constraints $C_1, C_2, \ldots, C_N$ we then create the corresponding equisatisfiable SMT(Cost) formula $\bigwedge_i E_i \wedge B_i$.

For the other direction we first separate the propositional part of the formula from the theory part by replacing all incur predicates with a fresh variable p_i and keeping the incur predicates separate in the formula

$$I \hat{=} \bigwedge_i p_i \Leftrightarrow \mathsf{incur}(v, c_i, d_i)$$

and in the same way replacing all $\lessdot$ predicates with fresh variables q_i and keeping the $\lessdot$ predicates separate

$$B \hat{=} \bigwedge_i q_i \Leftrightarrow \lessdot(v, b_i)$$

For every cost variable v which has more than one $\lessdot$, we create one fresh cost variable w and for all equivalences $p_i \Leftrightarrow \mathsf{incur}(v, c_i, d_i)$ in I we add new equivalences $p_i \Leftrightarrow \mathsf{incur}(w, c_i, d_i$ and replace one of the $v \lessdot b$ predicates with $w \lessdot b$ in B.

For each cost variable v, we now have a conjunction of incur predicates

$$\bigwedge_i P_i \Leftrightarrow \mathsf{incur}(v, c_i, d_i)$$

and a single $\lessdot$ predicate

$$q \Leftrightarrow v \lessdot c$$

This can now be expressed as two pseudo-boolean constraints, one for the case of the upper bound

$$\sum_i c_i p_i - c\neg q < c$$

and one for the case of the lower bound

$$\sum_i c_i p_i + cq \geq c$$

Translating the purely propositional part of the formula is straightforward, by first converting it into CNF and then translate each clause $\{l_1, \ldots, l_N\}$ into one pseudo-boolean constraint $\sum_i l_i \geq 1$. ■

8.4.1 Encoding Pseudo-Boolean constraints into SAT

For purely propositional pseudo-boolean problems, we use a simple heuristic to decide whether or not to encode a particular pseudo-boolean constraint into SAT. We based this on the work done in MiniSat+ [ES06], if the constraint can be expressed with a small number of clauses, it is translated into CNF. This is particularly useful for pseudo-boolean benchmark instances which often contain clauses $\{l_1, \ldots, l_n\}$ expressed as pseudo-boolean constraints $\sum_i l_i \geq 1$. Handling those in the theory of costs may cause unnecessary overhead.

8.5 Max-SMT

Since SMT(Cost) is equivalent to the pseudo-boolean problem, it is interesting to examine what properties and techniques lift from the pseudo-boolean case into SMT. One interesting feature is Max-SAT. Given an unsatisfiable SAT problem in CNF, the Max-SAT problem is discovering a maximal subset of clauses which are satisfiable. This problem can be easily stated in SMT(Cost) by creating one cost variable v and converting every clause

$$\{l_1, l_2, \ldots, l_N\}$$

where $l_1, l_2, \ldots, l_N$ are literals into a clause

$$\{l_1, l_2, \ldots, l_N, \neg\mathsf{incur}(v, 1, d)\}$$

where d is a unique integer. We shall call this optimisation problem Max-SMT. There are several extensions of the basic problem:

– Partial Max-SMT
– Weighted partial Max-SMT

In partial Max-SMT, there are some clauses that must be satisfied, so called *hard* clauses, and others that may be violated, so called *soft* clauses. Hard clauses can simply be modelled as the original clause. In weighted partial Max-SMT, each clause has a cost, and not satisfying it incurs that cost.

In MathSAT we can generalise this into arbitrary constraints or formulae. A hard formula is simply a conjunct of the formula, and a soft formula is modelled as an equivalence $\mathsf{incur}(v, 1, d) \Leftrightarrow \phi$. A weighted soft formula can be modelled in the same way with the corresponding weight as a cost.

8.6 Implementation issues

It is easy to make most operations in the theory solver $O(1)$. For every cost variable we keep track of the currently incurred cost c , and we also keep all unassigned incur predicates on each variable sorted by cost in a doubly linked list. We also keep a map from each literal to their position in the list to keep look up in the list in constant time. When a new incur literal is assigned, we remove the corresponding element from the linked list and update the currently incurred cost. Since literals are retracted in the reverse order in which they are added, we know where the element should be place in the linked list, so retracting a literal can also be done in constant time.

Checking for consistency is best performed by keeping two stacks of the assigned $\lessdot$ literals for each variable, one for upper bounds and one for lower bounds. The current bounds $[b_l, b_u)$ are reflected by the tops of these stacks. When a positive literal $v\lessdot$ is added, this is pushed on the stack of upper bounds iff it is tighter than the current bound, and negative literals are handled correspondingly. If the current upper bound $b_u \leq c$, we have found a conflict, and a minimal conflict set is found by greedy search on the positive incur literals with the largest costs. Deduction can also be done in constant time. Since we keep all unassigned predicates $\mathsf{incur}(v, c_i, d_i)$ on cost, we can deduce $\neg\mathsf{incur}(v, c_n, d_n)$ iff $c_n \geq c - b_u$ where we only need to look at the last predicate in the sorted list. Deductions based on lower bounds can be implemented analogously. In this way deduction will be linear in the number of deduced literals.

8.7 Experimental evaluation

All experiments in this chapter were carried out on machines with dual Intel Xeon E5430 CPUs running at 2.66 GHz using 16 GB of RAM running Linux. The experiments were run using a time limit of 300 seconds, and a memory limit of 2 GB.

Since MathSAT extended with the theory of costs is able to support the standard Max-SAT and Pseudo-Boolean Optimisation problems, we will evaluate performance on such problems against some dedicated solvers for these problems. This will hopefully demonstrate any inefficiencies in the basic idea wrt. optimisation and costs without mixing in other theories. Using SMT formulae, it may be difficult to separate the performance of the cost solver from the performance of other theory solvers. The current implementation is still very basic and does not include any heuristics or special tricks that may improve performance.

To get an idea of the performance of our theory solver we therefore first evaluate MathSAT on Max-SAT and Pseudo-Boolean Optimisation problems, using dedicated solvers for these problems as comparison. Then we evaluate the performance on Max-SMT, using the Yices solver for comparison.

8.7.1 Max-SAT

For Max-SAT, we have compared performance with several dedicated Max-SAT solvers which competed in the 2009 Max-SAT Evaluation, and with Yices as it is also able to solve the Max-SAT problem. The solvers are

- Clone [PD07a]
- MsUncore 2 [MMP09]
- SAT4J [SAT]
- Yices [DdM06]

We have chosen 100 industrial instances randomly from each of the Max-SAT and Partial Max-SAT categories in the last Max-SAT Evaluation 2009, and all 80 industrial weighted partial Max-SAT instances from the same source. In these categories MsUncore placed first in pure Max-SAT, and third in weighted Max-SAT. SAT4J placed first in weighted partial Max-SAT. For MathSAT, we use both linear and binary search, and the results are summarised in table 8.1.

Table 8.1: Performance on Max-SAT problems.

Category	Solver	Optimum	Sat	Time	Mean	Median
Max-SAT	MsUncore	83	0	2191.17	26.40	6.94
	Yices	56	0	1919.79	34.28	8.16
	SAT4J	30	50	1039.07	34.64	12.54
	MathSAT-binary	16	71	1017.87	63.62	20.41
	Clone	15	0	2561.06	170.74	129.06
	MathSAT-linear	5	82	466.91	93.38	72.05
Partial Max-SAT	Yices	71	0	1643.60	23.15	0.23
	SAT4J	67	31	1943.81	29.01	1.48
	MathSAT-binary	55	43	248.00	4.51	0.07
	MathSAT-linear	53	45	611.52	11.54	0.10
	MsUncore	46	0	353.84	7.69	0.20
	Clone	44	29	1743.54	39.63	6.59
Weighted partial Max-SAT	MathSAT-binary	80	0	110.49	1.38	1.23
	SAT4J	80	0	271.86	3.40	3.26
	MsUncore	80	0	579.20	7.24	7.09
	MathSAT-linear	79	1	1104.10	13.97	8.95
	Clone	0	0	0.00	N/A	N/A

The table shows for each solver the number of instances where it found an optimal solution, the number of instances it found a non-optimal solution, and for the instances where it found some solution it also shows the median, mean and total execution time. For each category the rows are ordered from "best" to "worst".

As expected, MathSAT performs poorly on pure Max-SAT instances. This is due to the size of the theory conflicts, which are unable to effectively prune search. For the other two problem categories, these problems include a significant number of hard clauses, and so the approach we have taken seems to perform much better.

We can however also see that MathSAT manages to find some solution more often than any other solver, even if the optimum is not always found.

Table 8.2: Performance on Pseudo-Boolean problems.

Category	Solver	Optimum	Unsat	Sat	Time	Mean	Median
OPT-SMALLINT	SCIP	98	8	62	3078.88	29.04	3.49
	Bsolo	88	7	110	1754.31	18.46	0.43
	PBClasp	67	7	127	869.66	11.75	0.05
	MathSAT-linear	63	7	132	1699.69	24.28	0.21
	MathSAT-binary	63	7	132	2119.07	30.27	0.22
	SAT4J	59	6	127	1149.96	17.69	1.34
OPT-BIGINT	MathSAT-binary	52	13	45	2373.35	36.51	15.54
	MathSAT-linear	48	13	49	1610.04	26.39	13.40
	SAT4J	19	18	51	759.15	20.51	3.55

8.7.2 Pseudo-Boolean Optimisation

The following dedicated solvers for Pseudo-Boolean optimisation which were taking part in the 2009 Pseudo-Boolean Evaluation was used for comparison:

- Bsolo [MM05a]
- PBClasp [PBc]
- SAT4J
- SCIP [BHP09]

SCIP was the winner of the OPT-SMALLINT category, and SAT4J was the winner of the OPT-BIGINT category. We test performance both on OPT-SMALLINT, where all coefficients are smaller than 2^{20}, as well as on OPT-BIGINT where at least one coefficient is larger than 2^{20}. Unfortunately, SAT4J was the only other solver capable of handling arbitrarily large coefficients. We selected 189 industrial instances from OPT-SMALLINT, and 224 industrial instances from OPT-BIGINT which were publicly available in 2009. The results are summarised in table 8.2. For SMALLINT, MathSAT lags far behind the 2009 winner SCIP, closer to PBClasp which was in third place when it comes to finding optimal solutions, or determining that an instance is unsatisfiable. However, MathSAT finds *some* solution for 202 instances, which is only outdone by Bsolo with 205 instances.

In the OPT-BIGINT category, MathSAT clearly outperforms SAT4J, but since this is the only other solver which support arbitrarily large coefficents, it is difficult to draw conclusions from this. Especially when also noting that SAT4J appears to lag behind the other solvers on OPT-SMALLINT.

8.7.3 Max-SMT

We have also tested performance of optimisation of formulae using some other theory, in this case linear real arithmetic. In this experiment, we have produced some randomly generated weighted Max-SMT formulae, by combining several formulae from the SMT-LIB with random weights on each clause. We have also generated partial weighted Max-SMT formulae by assigning random weight to a subset of the theory atoms on unsatisfiable formulae.

For comparison we use Yices, which supports Max-SMT. Barcelogic also supports Max-SMT [BNO+08, NO06]. Unfortunately we were not able to obtain an optimised version of Barcelogic supporting Max-SMT in time, so here we will only use Yices for comparison.

The results are summarised in table 8.3. The table shows for how many instances the optimum was found, for how many instances each solver was alone in finding the optimum, the total execution time, as well as the mean and median of the execution time. We can see that MathSAT with binary search appears to outperform linear search, as well as Yices.

Table 8.3: Performance on Max-SMT problems.

Category	Solver	Optimum	Unique	Time	Mean	Median
Weighted	MathSAT-binary	56	6	4886.59	87.26	68.38
Max-SMT	Yices	47	3	5260.67	111.92	86.21
	MathSAT-linear	23	0	4777.45	207.71	251.00
Weighted partial	MathSAT-binary	206	1	1462.98	7.10	2.45
Max-SMT	MathSAT-linear	206	1	2228.39	10.81	4.02
	Yices	195	0	3559.53	18.25	3.19

Chapter 9

Related work

The topic of bit-vector reasoning has received a lot of attention in recent years, and there is a rich corpora of related work. In this chapter, we will try to give an overview of the field, dividing the chapter into one section for each subtopic.

9.1 Solving

There are many works based on encoding into SAT, which uses a possibly modified SAT solver in some way to reason with bit-vector formulae.

In [WSK04], Wedler et al. proposes a modified DPLL procedure is proposed with ad hoc support for arithmetic. In this work, the solver uses a modified decision heuristic and conflict analysis, in order to take advantage of the known structure of addition networks in the original bit-vector formula.

In [NB04], Novikov and Brinkmann describe a modular SAT procedure aimed at RTL verification. Instead of bit-blasting the entire RTL design, blocks may be modeled using ad hoc procedures within the SAT procedure. Assigning values to inputs or outputs of this block causes the ad hoc procedure to propagate value to other externally visible signals in the block.

In [CKS05] Cook et al. propose an eager encoding into SAT with the Cogent solver which is applied for program verification. The solver encodes bit-vector formulae in a straightforward way by standard bit-blasting.

In [WFG+07], Wille et al. describe an extension to the DPLL procedure aimed at solving bit-vector formulae efficiently similar to Novikov and Brinkmann. Apart from ad hoc procedures for modules like multiplication, Wille et al. also use a specialised decision heuristic based on the type of module.

9.1.1 Bit-blasting

Babić proposes in [Bab08] to use strength reduction to reduce the size of the generated CNF. This is a technique from compiler optimisation which can translate multiplication and division by constants into potentially more efficient shifts and additions. He also proposes to base bit-blasting on *gate-minimal* circuits which should result in a smaller number of generated variables and clauses.

Jha et al. proposes in [JLS09] to use a relational encoding for division and remainder, which they claim delivers superior performance.

9.1.2 Encoding into LIA

Brinkmann and Drechsler proposed [BD02] a translation of a fragment of bit-vector constraints into linear arithmetic. Zeng et al [ZKC01] follows a similar approach. Later, Bozzano et al. proposed an extension in [BBC+06] into SMT formulae over the theory of bit-vectors by encoding into a SMT formula over linear integer arithmetic rather than create a large linear arithmetic problem for the entire formula. Independently Kroening [Kro05] proposed a similar translation into SMT over linear arithmetic together with a solver for linear arithmetic based on the Omega test [Pug91].

The approach of Bozzano et al was later discovered to suffer scaling problems when applied to more complex real-world formulae, and in [BCF$^+$07] Bruttomesso et al. proposed a lazy encoding of bit-vector literals into linear arithmetic. In this work the translation into linear arithmetic was not performed up-front, but inside a theory solver, and only when after simplification at the bit-vector level had been performed. The lazy approach was also paired with a number of inference rules which were applied in a theory solver in an attempt to find many of the “simpler” theory conflicts without having to resort to an encoding into linear arithmetic.

9.1.3 Modular arithmetic

The p-adic method was first proposed in [Mal03] and extended to non-linear arithmetic by Babić and Musuvathi in [BM05]. As reported by Babić in [Bab08], the approach by Babić and Musuvathi may not be scalable to more complex bit-vector problems.

9.2 Preprocessing

Here we will give a brief overview on the related work dealing with the main preprocessing techniques.

9.2.1 Simplification

Barrett et al. [BDL98] proposed simplifications which produces a canonical form for a fragment of the bit-vector theory as defined in this thesis.

In [GBD05] Ganesh et al. proposed a set or simplification rules which can be applied in polynomial time.

Bruttomesso et al. [BCF$^+$07] proposed a small set of simplification rules which were hardcoded into the solver. These were local, and could be performed in linear time.

Babić [Bab08] performs simplification using a term rewriting engine, using approximately 160 rules. According to the author the engine is not recursive, which seems to indicated that it is not applied to a fix-point, but very little detail is provided on how the engine works. Our idea of using term rewriting was developed independently of this work.

Several works have proposed simplifications on bit-blasted formulae before solving. Brummayer and Biere [BB06] proposes a set a rewrite rules on and-inverter graphs (AIGs). Another approach was proposed by Eén et al. [EMS07] based on DAG-aware minimisation and structure technology mapping.

9.2.2 Substitution

Performing substitutions to eliminate variables has been proposed in several earlier works. With this substitution, if we have a formula of the form $v = t \wedge \phi$ where v does not occur in t, then this formula can be replaced with $\phi[v \mapsto t]$. This has been proposed by Ganesh et al. [GBD05] were it is called propagation of equalities as well as Bruttomesso et al. [BCF$^+$07] where it is called variable elimination, and Jha et al [JLS09] which call it equality propagation. Both Bruttomesso et al. and Jha et al. also perform substitutions on formulas on the form $p \wedge \phi$ which can be replaced with $\phi[p \mapsto \top]$.

Term substitution is also performed in Boolector as described by Brummayer [Bru09] although there are not many details which kinds of substitutions are performed.

9.2.3 Propagation of unconstrained terms

Propagation of unconstrained terms was first proposed for SMT solving by Bruttomesso et al. in [BCF+07].

Brummayer independently proposed the technique in [Bru09]. A more limited version is also used in Beaver [JLS09], where if all subterms are unconstrained, or in their terms "don't cares", the term itself is also a don't care.

None of these works describe how models can be computed for satisfiable formulae, or how one can propagate unconstrained formulae as well as terms.

9.3 Extension of EUF

Bruttomesso and Sharygina proposed in [BS09] an extension of EUF which is based on a lazy version of the core rewrite technique of Cyrluk et al [CMR97].

9.4 Minimal or reduced model enumeration

Several approaches have been proposed for reducing the number of literals sent to the theory solvers. de Moura and Bjørner [dMB07b] proposed what they call *relevancy propagation*. In this technique the current truth assignment created by the SAT solver is checked on the original formula. If a particular literal in the current truth assignment can not affect satisfiability of the original formula, it is not communicated to the theory solvers.

Another technique has been proposed by Sebastiani [Seb07] called *ghost filtering*. In this technique if a particular variable only occurs in already satisfied clauses, it can be ignored when selecting a new decision variable by the SAT solver. The advantage of this technique is that it does not require a particular encoding of the formula of access to the original formula, the downside is that it requires some changes to the decision heuristic of the SAT solver.

Minimal model generation by a dual rail encoding was used by Roorda and Claessen [RC06], although the paper doesn't mention how the dual rail encoding achieves minimality of models. It was also used for SMT in [BCF+07], although the details were left out because of space constraints.

9.5 Approximation

Approximation/refinement schemes was first proposed for SMT solving by Bryant et al. in [BKO+07]. In this work, under-approximations limited the domain of bit-vector variables (reducing the number of bits and then performing a sign extension back to the original width). Refinement was done by increasing the domain. Over-approximations were done by replacing subformulae with fresh propositional variables.

He and Hsiao [HH08] use under-approximation in a way similar to Bryant et al., but instead of just performing sign extension they also attempt to use zero-extension.

Brummayer and Biere [BB09b] propose a generalisation of the sign- and zero-extension by He and Hsiao by partitioning bit-vectors and assigning all bits in some partitions the same value. They also show how this can be combined with over-approximation in a simple loop together with early termination for admissible results.

9.6 Reusing learnt information

Whittemore et al. [WKS01] describe reusing of learnt clauses in the SATIRE SAT solver. This is an incremental SAT solver which allows the user to retract clauses and add new ones before searching again. To implement this the solver keeps track of the dependencies between learnt clauses and original clauses. If a clause is retracted, all clauses which have been learnt using this clause are also removed. Silva and Sakallah [SS97] proposed a technique for reusing clauses from one formula to the next in automatic test pattern generation (ATPG) for circuits. In this application a SAT solver is used to try to generate stimuli that exposes a particular fault. They notice that some learnt clauses are independent of the current target fault instead depending only on the circuit being studied, and could be reused from one SAT problem to the next. This happens if a learnt clause is derived solely from clauses originating in the circuit. Strichman [Sht01] noticed that in the context of Bounded Model Checking (BMC), certain clauses could be reused from one unrolling to the next.

Eén and Sörensson showed in [ES03] how learnt clauses could be reused when doing k-induction. This relies on the idea that in this application we are monotonically adding non-unit clauses to the solver, and all unit-clauses can be used as assumptions rather than adding them permanently to the solver.

In [GD05] Große and Drechsler propose to reuse clauses learnt while solving one formula when solving another iff they can be derived from the intersection of the clauses in the two formulae.

Babić and Hu proposed some simple heuristics to decide if a fact is reusable or not in [BH09], which allow for reuse of learnt unit clauses.

The only work which considers the idea of reusing all information is the work by Eén and Sörensson, which is targeted for the case of k-induction where all non-unit clauses in one formula will occur also in the next. For general solving of similar formulae which are not extensions of one another, all previous work concentrate on techniques to compute the relevant parts of the learnt clauses and reuse only those.

9.7 Simultaneous SAT

Khasidashvili et al. [KNPH06] introduced a technique for solving a set of related formulae using an algorithm they call Simultaneous SAT (SSAT). Given a formula in CNF and a set of *proof objectives* being literals in this formula, their algorithm is a modification of a normal DPLL-like algorithm. They always keep a particular proof objective as the current goal to satisfy, the *currently watched proof objective*. At any decision this literal is chosen unless it has already been given a truth value. When the solver finds a model, it checks all other proof objectives and records all that have been satisfied by the model. Then a new currently watched proof objective is chosen among those which has not yet been solved. This is repeated until all proof objectives have been solved. The SSAT algorithm can be seen as a special case of reusing learnt information when all formulae to be solved are known in advance.

In contrast to the SSAT algorithm, the MSPSAT algorithm presented in this work doesn't require any modifications of the underlying solver. Indeed it would be possible to implement using the MathSAT API rather than modifying any part of the solver.

9.8 Unsat core extraction

Zhang and Malik showed in [ZM03] how to use the proof of unsatisfiability from a proof-producing SAT solver to compute an unsatisfiable core. In this work a core is the set of clauses which occur in the proof of unsatisfiability.

The idea of using a set of assumptions representing subformulae and computing an unsatisfiable core by computing a final conflict in terms of these assumptions was first published by Griggio et al. in [CGS07] and also by Asin et al. in [ANOR08]. The technique seems to have been previously known in the field though.

The technique is also used in the Yices SMT solver, although the details are unpublished, and it is implemented in some SAT solvers, such as MiniSat.

9.9 Optimisation

The problem of optimisation in SMT was first introduced by Nieuwenhuis and Oliveras [NO06] with a focus on the Max-SMT problem. This work introduced a theory which could encode a single objective function, and optimisation was handled by monotonically strengthening the theory. This strengthening is used to express an increasingly strict upper bound on the objective function, meaning that the only possible search algorithm in this theory is linear search. In contrast our approach supports several objective functions making it possible to encode pseudo-boolean constraints efficiently, as well as several different optimisation algorithms.

Max-SMT is also supported in the Yices solver, but no published details are available on the underlying techniques used to implement this feature.

In the field of Pseudo-Boolean Optimisation, the work by Aloul et al. [ARSM07] may be the most similar. This uses a generalisation of the unit-propagation rule to propagate literals from Pseudo-Boolean constraints, in a way similar to how truth assignments to incur predicates can be deduced in our theory of costs. They use a “sliding” upper bound to implement optimisation, which also seems to rule out the possibility of performing dichotomic search on the objective function.

Chapter 10

Conclusions

We have seen several preprocessing and approximation techniques, and experimental data shows that for real-world formulae, preprocessing is often far more important than which precise techniques are used to actually solve a formula.

For solving formulae, the experimental evidence has shown that a basic lazy scheme suffers from some drawback in comparison to a translation into SAT, but that it also can help boost performance significantly on some instances. Cases where the lazy scheme is advantageous seem to be where layering is helpful, using abstractions of the theory which can be efficiently decided. A possible application area where this might occur frequently is in equivalence checking or related fields. The two instances of Burch-Dill style [BD94] verification of processor pipelines included in the SMT-LIB are examples of this.

We have also seen techniques which allow model computation while all preprocessing techniques are used, which makes these techniques useful in practise and the performance of the solver more stable and predictable with respect solving without computing models.

With the theory of costs proposed in this thesis, we show that it is possible to support optimisation, Max-SMT and the compactness of Pseudo-

Boolean constraints with very minor changes to any SMT solver following the lazy schema. This is achieved by a new theory, rather than invasive modifications of the solver.

Overall, there seems to be support for concluding that MathSAT with the theory of costs delivers good performance, and is largely comparable to state of the art techniques in dedicated Max-SAT and Pseudo-Boolean solvers.

10.1 MicroFormal

In this industrial case study, we have seen that it is not enough to have an efficient solver, how this solver is used can have a big impact on performance. We have seen that reusing learnt information from solving previous formulae can be very useful, and that in some cases it is possible to achieve good performance without resorting to more complex techniques for reusing information that have been proposed in the past.

MathSAT has now been successfully integrated into MicroFormal, and it delivers significantly improved performance over the SAT-based solver previously used. A version of the tool set with MathSAT integrated has been made available to users within Intel with MathSAT available as a command-line option. This version has been successfully used for verification of a next generation microarchitecture. According to our partners at Intel, in the future, MathSAT will be made the default decision procedure in MicroFormal.

10.2 Future work

The work presented in this thesis has given us the impression that so far, we are only scratching the surface of what may be possible with bit-vector reasoning. For this reason the future work section may seem to be a bit extensive, but hopefully useful to the reader. It is not by any means a comprehensive list of possibilities for future directions, just a small sampling of ideas that may be interesting.

10.2.1 Stochastic local search

The success of the under-approximations used in this thesis seems to indicate that often guessing values for variables may be a successful strategy. This leads to the idea of using stochastic local search (SLS) for SMT. Some attempt in this direction has been done in [GST09], but in that work local search was only performed on the propositional abstraction of the formula, essentially replacing the DPLL solver with a local search algorithm. Theory consistency was then checked using conventional theory solvers. But if the results on under-approximation is an indication, performing local search on the variables in the formula rather than the propositional abstraction may be fruitful. The disadvantage of this is that the local search algorithm need to be purpose-built for each theory.

In fact, performing some preliminary exploratory experiments with SLS was the initial motivation for what became the under-approximation described in section 5.2.

10.2.2 Model computation

Although we have shown that it is possible to compute models while still retaining all preprocessing techniques described in this thesis, the current implementation which allows this is a preliminary prototype which still shows unnecessary computational overhead. To evaluate the full potential of the technique described, an optimised version needs to be developed and tested in cases where large numbers of rewrites are performed on formulae in preprocessing.

10.2.3 Adaptability of solver

When solving bit-vector formulae using DPLL(T), MathSAT currently uses both the EUF solver and the bit-vector solver, always calling the cheaper EUF solver first. Only if it fails to find any inconsistency is the bit-vector solver called. In [BCF+07], a more complicated design was shown with more layers of increasingly more powerful solvers. This seems advantageous in principle since it increases the chances of avoiding a potentially expensive call to the final bit-vector solver, but on the other hand it also increases the complexity of the system. Determining what all these layers should contain is not a simple problem, and there is a risk that such an architecture requires continuous tinkering with the layers to achieve good performance as time goes by.

An alternative might be an adaptive theory solver which is capable of learning from previous experience in solving formulae. Since we would like to discover theory conflicts more cheaply than by a call to the bit-vector solver, we could try to analyse the conflicts found by the theory solver and attempt to avoid having to call the theory solver in a similar situation in the future.

As an example, if we find the bit-vector theory conflict

$$\begin{aligned}
&\neg(x^{\langle 64\rangle} <_{\mathrm{u}} 172)\\
&x^{\langle 64\rangle} <_{\mathrm{u}} \mathrm{sext}^{\langle 64\rangle}(24) + \mathrm{sext}^{\langle 32\rangle}(y^{\langle 32\rangle})\\
&\neg(z^{\langle 64\rangle} <_{\mathrm{u}} 176)\\
&\neg(z <_{\mathrm{u}} \mathrm{sext}^{\langle 64\rangle}(24) + \mathrm{sext}^{\langle 32\rangle}(y^{\langle 32\rangle}))\\
&z^{\langle 64\rangle} <_{\mathrm{u}} 172
\end{aligned}$$

Can we learn something more general from this conflict which can be reused later? To start with, we can notice that this conflict is not minimal,

$\neg(z^{\langle 64\rangle} <_{\mathrm{u}} 176)$ can be removed from the set. Now we can start trying to create a "generalised" conflict.

$$\begin{array}{l} \neg(x^{\langle n\rangle} <_{\mathrm{u}} v^{\langle n\rangle}) \\ x^{\langle n\rangle} <_{\mathrm{u}} y^{\langle n\rangle} \\ \neg(z^{\langle n\rangle} <_{\mathrm{u}} y^{\langle n\rangle}) \\ z^{\langle n\rangle} <_{\mathrm{u}} v^{\langle n\rangle} \end{array}$$

Finally we can universally quantify over all variables to create a non-ground axiom. Several steps are necessary to produce a generalised conflict from a conflict learnt during search:

– Minimise the conflict set. This can be done with several different techniques such as those used to compute unsatisfiable cores, or minimisation of infeasible subsets in linear programming e.g. following the approaches in [CD91, AS08].
– Replacing irrelevant terms with fresh variables. This can be done using the delta-debugging technique described by Brummayer and Biere in [BB09c].
– Generalise from fixed sized bit-vectors to non-fixed sized bit-vector terms where possible. Non-fixed size bit-vector formulae are not decidable, so this step may not be possible to automate fully. A possible approach could be to use a theorem prover with support for the theory of non-fixed width bit-vectors and perform generalisation in cases this theorem prover can prove is correct in a reasonable time.
– Add universal quantifiers on all variables to make the conflict set into a non-ground axiom.

With a framework which is able to perform efficient matching of this generalised conflict set against the current truth assignment, we would be able to find any similar conflict cheaply in the future. The advantage of this is that it doesn't depend on incrementality of any other technique of reusing learnt information withing the same execution of the tool, but can be used to speed up completely separate run of the solver. If we run the solver twice on the same formula, learning generalised conflicts after the first run, we can expect that execution time will be improved in the second run.

An improvement would be to integrate this into the EUF solver, where it would be possible to check for conflicts on the congruence closure rather than just the set of literals in the current truth assignment. In this case the matching of generalised conflicts is similar to the problem of E-matching which have been described in earlier works [dMB07a, MŁK08]

Another possibility for adaptivity of the solver would be to use machine learning techniques to adapt solver heuristics based on previous experience.

10.2.4 DPLL(T) or the lazy schema

We have seen in the experimental evaluation in chapter 6 that in the lazy schema there are several issues with performance.

– For satisfiable instances, the solver may visit a large number of truth assignments before finding a consistent truth assignment. Guiding the solver towards the right truth assignment may help improve performance in these cases, and there are several ways in which this could be accomplished such as by deduction.
– Sometimes the boolean enumerator makes decisions which forces the theory solver to solve very difficult consistency problems. These may be difficult enough that the solver times out trying to solve these, although if the boolean enumerator had only made the "right" decisions, the formula would have been trivial to solve. When translating into SAT, the solver uses restarts to get out of unproductive parts of the search space, but in the lazy scheme the theory solver is forced to solve this truth assignment before search can continue. A possible so-

lution to this would be to allow the theory solver to "give up" and ask the top-level to attempt to find another truth assignment if possible,

– [BPST10] introduces theory solver suggestions, literals which can not be deduced but are consistent with the internal state of a theory solver. No data on the effectiveness of such a technique is yet available, but it may help top guide the top-level SAT solver towards satisfiable truth assignments, finding models quicker.

10.2.5 MicroFormal

Although some improvements have been made to MicroFormal in this case study, the time taken to solve formulae is still considerable compared to the rest of the work of the symbolic execution engine, on average over half the execution time is spent in solving formulae. Therefore, it would be interesting to look for further ways of reducing the time taken to solve instances as well as reducing the number of instances that need to be solved. Listed below are a few possibilities which may be interesting to investigate.

Improvements on non-singleton sets Currently, the MSPSAT algorithm is being used in MicroFormal for solving non-singleton sets of instances. However, the experimental results show that with a suitable reset strategy, solving them in the same way as singleton sets can achieve better performance. Switching to this would give better performance until a more efficient version of MSPSAT can be developed.

Better models Since MicroFormal is currently capable of storing models for previous formulae, and use these in a model caching scheme to either avoid future solver calls, or significantly reduce the complexity of future calls, it makes sense to attempt to adapt the models returned from the

Table 10.1: Model caching potential

Program	Instances	Models
Program 1	1702	187
Program 2	1121	142
Program 3	7704	1656

solver to maximise the utility of this feature. A "good" model is in this case one which models (or can be extended to model) as many future formulae as possible, therefore minimal (or near minimal) models may be interesting.

As an indication of the potential of better models, table 10.1 shows how many unique models are needed to model all satisfiable singleton instances. This data was computed with a basic greedy search trying to satisfy as many instances as possible with the same model, it is possible that the potential for model caching is greater than the results indicate. There seems to be some significant potential in discovering better models to avoid future solver calls, but it is unclear how this potential can be harnessed.

Heuristics for resets The reset strategy used in this work is a simple strategy with a fixed reset frequency. Although it has been shown to deliver a significant performance improvement, it is still vulnerable to outliers in the sequence of instances. It would be interesting to discover heuristics capable of detecting when irrelevant information stored in the solver is likely to negatively affect performance, and build an adaptive reset strategy around such a heuristic. This should allow for longer reset intervals in the cases where no outliers exists, and further improve performance. Given that the chosen reset interval was conservative because of the risk of encountering outliers, it might be possible to achieve performance closer to the optimal. Looking closer at the singletons in program 3, the sequence which had

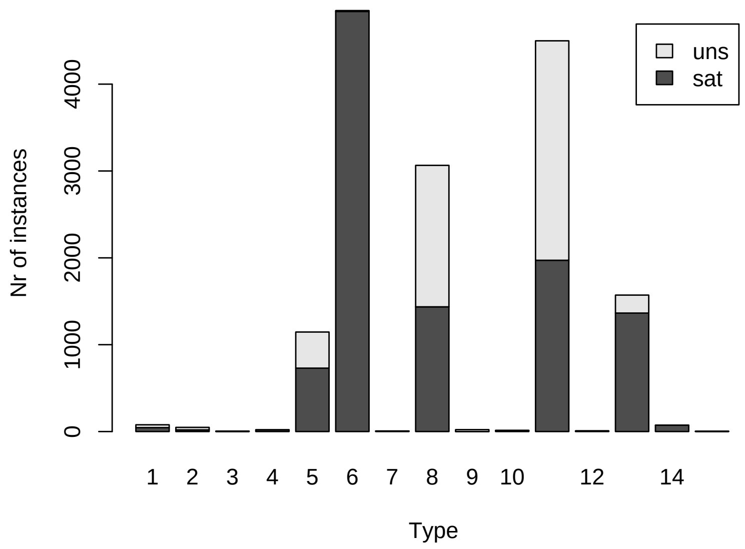


Figure 10.1: Kinds of instances

several outliers, the optimal reset strategy seems to prefer to reset immediately after resets. Developing a heuristic around this observation may be fruitful. Another interesting avenue would be to experiment with techniques for removing irrelevant information in the solver while still keeping what is still relevant. A challenge is to predict which information will not become relevant again in the near future.

Heuristics based on instance origin Although the symbolic execution engine in MicroFormal only computes paths, it is possible to separate instances generated by it by their origin, meaning from where inside the tool the instances where generated. It might be possible to discover some pattern in the origin of instance dividing instances into different kinds which might be solved in different ways. Figure 10.1 shows the number of singletons of different origins in the three program used for experiments in section 7. MicroFormal has been instrumented to provide a hint on what part of the engine created each instance, and in these cases there were a total of 15 different types. It seems that certain types (6 in particular) are very likely to be satisfiable; It may be possible to use such observations to develop heuristics that improves performance.

A hybrid concrete/symbolic execution engine One technique which can quickly discover sets of paths in a program is fuzz testing. It might be possible to combine fuzzing with symbolic execution by starting with generating a number of paths with fuzzing, and then extending this set using symbolic execution. The two methods can be interleaved by a technique similar to [GLM07] where given a specific path computed by concrete execution new paths can be computed by picking the branching points of this path and generating a path feasibility condition to determine if any other branch is possible. If a new partial path is discovered, this can be completed by concrete execution of that path, and fuzzing may be applied to discover many other similar paths taking the same branch. Judicial use of fuzzing and concrete execution may in the best case be able to significantly reduce the number of formulae that need to be solved, and taking a closer look at this possibility may be a fruitful avenue of research.

Other possibilities There are many other possibilities for future improvement. Among them are, apart from faster performance for pure solving, the following:

- Support for uninterpreted functions. MicroFormal abstracts some parts with uninterpreted functions, but currently those are eliminated using Ackermann's expansion by MicroFormal itself. Passing the original formula on to the solver may improve performance.
- Parallelism. Although parallelising non-singleton sets was shown to have some scaling issues, there are other opportunities for parallelism

in MicroFormal. An example would be performing the symbolic execution in parallel exploring several paths simultaneously.

10.2.6 The theory of costs

The theory presented here is so far very basic, and many improvements and extensions are possible.

Heuristics for dichotomic search Apart from the basic linear or binary search many other strategies are possible.

- We could pick a bound other than the midpoint, say $b = \frac{2(u-l)}{3}$ to reduce the chance of incorrectly guessing a bound below the optimum.
- We could mix linear search with dichotomic search. We could run a few iterations of linear search interspersed with a few iterations of picking some lower bound.
- An interesting alternative might be an adaptive splitting strategy were we choose to split on

  $$b = \lceil k(u-l) \rceil$$

  were $k \in (0, 1]$ and is updated in each iteration based on how the search progresses. If the previous bound was satisfiable, we might increase k. If it was unsatisfiable, we might decrease k.

Estimating lower bounds In Pseudo-Boolean Optimisation, several techniques of estimating lower bounds have been proposed e.g. [MM05b]. It may be interesting to investigate how well this or similar work lifts to the theory of costs.

Cost order There are many possible extensions to the basic theory. One in particular is to add an ordering between cost variables. If v_1, v_2 are cost variables, then $v_1 \lessdot v_2$ is a predicate imposing a strict order on these two variables. This would allow modelling of relations between different cost functions, which might be interesting. So far, we have however no application of this extension in mind.

Bibliography

[ABC$^+$02] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Korniłowicz, and Roberto Sebastiani. Integrating boolean and mathematical solving: Foundations, basic algorithms, and requirements. In *Artificial Intelligence, Automated Reasoning, and Symbolic Computation*, pages 157–192. 2002.

[AEF$^+$05] Tamarah Arons, Elad Elster, Limor Fix, Sela Mador-Haim, Michael Mishaeli, Jonathan Shalev, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Lenore D. Zuck. Formal verification of backward compatibility of microcode. In *Computer Aided Verification*, pages 185–198. 2005.

[AEMS06] Tamarah Arons, Elad Elster, Terry Murphy, and Eli Singerman. Embedded Software Validation: Applying Formal Techniques for Coverage and Test Generation. *Microprocessor Test and Verification, International Workshop on*, 0:45–51, 2006.

[AEO$^+$08] Tamarah Arons, Elad Elster, Shlomit Ozer, Jonathan Shalev, and Eli Singerman. Efficient symbolic simulation of low level software. In *Proceedings of the conference on Design, automation and test in Europe*, pages 825–830, Munich, Germany, 2008. ACM.

[ANOR08] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Efficient generation of unsatisfiability proofs and cores in SAT. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 16–30. 2008.

[ARSM07] Fadi A. Aloul, Arathi Ramani, Karem A. Sakallah, and Igor L. Markov. Solution and optimization of systems of Pseudo-Boolean constraints. *IEEE Trans. Comput.*, 56(10):1415–1424, 2007.

[AS08] Mustafa K. Atlihan and Linus Schrage. Generalized filtering algorithms for infeasibility analysis. *Comput. Oper. Res.*, 35(5):1446–1464, 2008.

[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st international joint conference on Artifical intelligence*, pages 399–404, Pasadena, California, USA, 2009. Morgan Kaufmann Publishers Inc.

[Bab08] Domagoj Babić. *Exploiting Structure for Scalable Software Verification.* PhD thesis, University of British Columbia, Vancouver, Canada, 2008.

[BB06] Robert Brummayer and Armin Biere. Local Two-Level And-Inverter Graph Minimization without Blowup. *2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'06)*, October 2006.

[BB09a] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for Bit-Vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer-Verlag, 2009.

[BB09b] Robert Brummayer and Armin Biere. Effective Bit-Width and Under-Approximation. In *Computer Aided Systems The-*

ory - EUROCAST 2009, volume 5717 of *Lecture Notes in Computer Science*, pages 304–311. Springer-Verlag, 2009.

[BB09c] Robert Brummayer and Armin Biere. Fuzzing and delta-debugging SMT solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, pages 1–5, Montreal, Canada, 2009. ACM.

[BBC$^+$05a] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *Tools and Algorithms for the Construction and Analysis of Systems*, number 3440 in Lecture Notes in Computer Science, pages 317–333. Springer Berlin / Heidelberg, 2005.

[BBC$^+$05b] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. MathSAT: Tight Integration of SAT and Mathematical Decision Procedures. *Journal of Automated Reasoning*, 35(1):265–293, October 2005.

[BBC$^+$06] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Ziyad Hanna, Zurab Khasidashvili, Amit Palti, and Roberto Sebastiani. Encoding RTL constructs for MathSAT: a preliminary report. *Electronic Notes in Theoretical Computer Science*, 144(2):3–14, 2006.

[BBR09] Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New encodings of Pseudo-Boolean constraints into CNF. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, pages 181–194, Swansea, UK, 2009. Springer-Verlag.

[BCF$^+$07] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. A Lazy and Layered SMT(BV) Solver for Hard Industrial Verification Problems. In *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 547–560. Springer-Verlag, 2007.

[BCF$^+$08] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT solver. In *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 299–303. Springer-Verlag, 2008.

[BD94] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In *Proceedings of the 6th International Conference on Computer Aided Verification*, pages 68–80. Springer-Verlag, 1994.

[BD02] Raik Brinkmann and Rolf Drechsler. RTL-Datapath verification using integer linear programming. In *Proceedings of the 2002 Asia and South Pacific Design Automation Conference*, page 741. IEEE Computer Society, 2002.

[BDL98] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35th annual Design Automation Conference*, pages 522–527, San Francisco, California, United States, 1998. ACM.

[BH08] Domagoj Babic and Alan J. Hu. Calysto: scalable and precise extended static checking. In *Proceedings of the 30th international conference on Software engineering*, pages 211–220, Leipzig, Germany, 2008. ACM.

[BH09] Domagoj Babić and Alan Hu. Approximating the safely reusable set of learned facts. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(4):325–338, October 2009.

[BHP09] Timo Berthold, Stefan Heinz, and Marc E Pfetsch. Solving Pseudo-Boolean Problems with SCIP. Technical Report 08-12, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 2009.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[Bie08] Armin Biere. PicoSAT Essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):75–97, 2008.

[BKO$^+$07] Randal Bryant, Daniel Kroening, Joël Ouaknine, Sanjit Seshia, Ofer Strichman, and Bryan Brady. Deciding Bit-Vector arithmetic with abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *Lecture Notes in Computer Science*, pages 358–372. Springer-Verlag, 2007.

[BM05] Domagoj Babić and Madanlal Musuvathi. Modular arithmetic decision procedure. Technical Report TR-2005-114, Microsoft Research Redmond, 2005.

[BMS$^+$96] Ricky W Butler, Paul S Miner, Mandayam K Srivas, Dave A Greve, and Steven P Miller. A bitvectors library for PVS. Technical report, NASA Langley Technical Report Server, 1996.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[BNO$^+$08] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. The barcelogic SMT solver. In *Computer Aided Verification*, pages 294–298. 2008.

[BPST10] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OpenSMT Solver. In *16th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*. To be published, 2010.

[Bru09] Robert Brummayer. *Efficient SMT Solving for Bit-Vectors and the Extensional Theory of Arrays*. PhD thesis, Johannes Kepler University Linz, November 2009.

[BS09] Roberto Bruttomesso and Natasha Sharygina. A Scalable Decision Procedure for Fixed-Width Bit-Vectors. In ICCAD 2009, to appear, 2009.

[CD91] John W. Chinneck and Erik W. Dravnieks. Locating minimal infeasible constraint sets in linear programs. *INFORMS Journal on Computing*, 3(2):157–168, 1991.

[CFG$^+$10] Alessandro Cimatti, Anders Franzén, Alberto Griggio, Roberto Sebastiani, and Cristian Stenico. Satisfiability Modulo the Theory of Costs: Foundations and Applications. In *16th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*. To be published, 2010.

[CGS07] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. A simple and flexible way of computing small unsatisfiable cores in SAT modulo theories. In *Theory and Applications of Satisfiability Testing – SAT 2007*, pages 334–339. 2007.

[CKS05] Byron Cook, Daniel Kroening, and Natasha Sharygina. Cogent: Accurate theorem proving for program verification. In *Computer Aided Verification*, pages 296–300. 2005.

[CMR96] David Cyrluk, Oliver Möller, and Harald Rueß. An Efficient Decision Procedure for a Theory of Fixed-Sized Bitvectors. Ulmer Informatik-Berichte 96-8, Universität Ulm, Fakultät für Informatik, 1996.

[CMR97] David Cyrluk, Oliver Möller, and Harald Rueß. An efficient decision procedure for the theory of fixed-sized bit-vectors. In *Computer Aided Verification*, pages 60–71. Springer Berlin / Heidelberg, 1997.

[Coc70] John Cocke. Global common subexpression elimination. *SIGPLAN Not.*, 5(7):20–24, 1970.

[Daw09] Jeremy Dawson. Isabelle theories for machine words. *Electronic Notes in Theoretical Computer Science*, 250(1):55–70, September 2009.

[DdM06] Bruno Dutertre and Leonardo de Moura. A fast Linear-Arithmetic solver for DPLL(T). In *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[dMB07a] Leonardo de Moura and Nikolaj Bjørner. Efficient E-Matching for SMT solvers. In *Automated Deduction – CADE-21*, pages 183–198. 2007.

[dMB07b] Leonardo de Moura and Nikolaj Bjørner. Relevancy propagation. Technical note, Microsoft Research Redmond, October 2007.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. 2008.

[EMS07] Niklas Een, Alan Mishchenko, and Niklas Sörensson. Applying logic synthesis for speeding up SAT. In *Theory and*

Applications of Satisfiability Testing – SAT 2007, pages 272–286. 2007.

[ES03] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[ES04] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, pages 333–336. 2004.

[ES06] Niklas Eén and Niklas Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

[GBD05] Vijay Ganesh, Sergey Berezin, and David L. Dill. A decision procedure for fixed-width bit-vectors. Technical Report CSTR 2007-06, Stanford Computer Science Department, 2005.

[GD05] Daniel Große and Rolf Drechsler. Acceleration of SAT-Based iterative property checking. In *Correct Hardware Design and Verification Methods*, pages 349–353. 2005.

[GD07] Vijay Ganesh and David Dill. A decision procedure for Bit-Vectors and arrays. In *Computer Aided Verification*, pages 519–531. 2007.

[GLM07] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated Whitebox Fuzz Testing. Technical Report MSR-TR-2007-58, Microsoft Research Redmond, Redmond, WA, May 2007.

[Got76] Eiichi Goto. Monocopy and Associative Algorithms in an Extended Lisp. Technical Report 74-03, Information Science Laboratory, University of Tokyo, 1976.

[Gri09] Alberto Griggio. *An Effective SMT Engine for Formal Verification*. PhD thesis, DISI – University of Trento, 2009.

[GST09] Alberto Griggio, Roberto Sebastiani, and Silvia Tomasi. Stochastic local search for smt: a preliminary report. Satisfiability Modulo Theories Workshop, 2009.

[HH08] Nannan He and Michael S. Hsiao. A new testability guided abstraction to solving bit-vector formula. In *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, pages 39–45, Princeton, New Jersey, 2008. ACM.

[HHLBS09] Frank Hutter, Holger Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research (JAIR)*, 36:267–306, 2009.

[HSU$^+$01] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Desktop Platforms Group, and Intel Corp. The Microarchitecture of the Pentium®4 Processor. *Intel Technology Journal*, 1:2001, 2001.

[JLS09] Susmit Jha, Rhishikesh Limaye, and Sanjit Seshia. Beaver: Engineering an efficient SMT solver for Bit-Vector arithmetic. In *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 668–674. Springer-Verlag, 2009.

[KaaV03] Sarfraz Khurshid, Corina Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. 2003.

[KF90] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Commun. ACM*, 33(5):539–543, 1990.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[KNPH06] Zurab Khasidashvili, Alexander Nadel, Amit Palti, and Ziyad Hanna. Simultaneous SAT-Based model checking of safety properties. In *Hardware and Software, Verification and Testing*, volume 3875 of *Lecture Notes in Computer Science*, pages 56–75. Springer-Verlag, 2006.

[Koe98] Jana Koehler. Planning under Resource Constraints. In *European Conference on AI (ECAI)*, pages 489–493, 1998.

[KPP04] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer-Verlag, Berlin, Germany, 2004.

[Kro05] Daniel Kroening. Linear Arithmetic with Bit-Vectors using Omega and SAT. Technical Report 483, ETH Zürich, Computer Systems Institute, 2005.

[LMS06] Luciano Lavagno, Grant Martin, and Louis Scheffer. *Electronic Design Automation for Integrated Circuits Handbook - 2 Volume Set*. CRC, April 2006.

[Mal03] G. I. Malaschonok. Solution of systems of linear equations by the p-Adic method. *Programming and Computer Software*, 29(2):59–71, March 2003.

[MF89] Sanjay Mittal and Felix Frayman. Towards a generic model of configuraton tasks. In *Proceedings of the 11th international joint conference on Artificial intelligence - Volume 2*, pages 1395–1401, Detroit, Michigan, 1989. Morgan Kaufmann Publishers Inc.

[MŁK08] Michał Moskal, Jakub Łopuszański, and Joseph R. Kiniry. E-matching for fun and profit. *Electronic Notes in Theoretical Computer Science*, 198(2):19–35, May 2008.

[MM05a] Vasco M. Manquinho and Joao Marques-Silva. Effective lower bounding techniques for Pseudo-Boolean optimization. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 2*, pages 660–665. IEEE Computer Society, 2005.

[MM05b] Vasco M. Manquinho and Joao Marques-Silva. Effective lower bounding techniques for Pseudo-Boolean optimization. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 2*, pages 660–665. IEEE Computer Society, 2005.

[MMP09] Vasco Manquinho, Joao Marques-Silva, and Jordi Planes. Algorithms for weighted boolean optimization. In *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 495–508. 2009.

[NB04] Yakov Novikov and Raik Brinkmann. Foundations of Hierarchical SAT-solving. In *Int'l Workshop on Boolean Problems*, pages 103–141, 2004.

[NO06] Robert Nieuwenhuis and Albert Oliveras. On SAT modulo theories and optimization problems. In *Theory and Applications of Satisfiability Testing - SAT 2006*, pages 156–169. 2006.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.

[PBc] PBclasp. `http://potassco.sourceforge.net/labs.html`.

[PD07a] Knot Pipatsrisawat and Adnan Darwiche. Clone: Solving weighted Max-SAT in a reduced search space. In *AI 2007: Advances in Artificial Intelligence*, pages 223–233. 2007.

[PD07b] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Theory and Applications of Satisfiability Testing – SAT 2007*, pages 294–299. 2007.

[Pla81] David A. Plaisted. Theorem proving with abstraction. *Artificial Intelligence*, 16(1):47–108, March 1981.

[Pug91] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13, Albuquerque, New Mexico, United States, 1991. ACM.

[RC06] Jan-Willem Roorda and Koen Claessen. SAT-Based assistance in abstraction refinement for symbolic trajectory evaluation. In *Computer Aided Verification*, pages 175–189. Springer-Verlag, 2006.

[RV01] Alan Robinson and Andrei Voronkov, editors. *Handbook of automated reasoning*. Elsevier Science Publishers B. V., 2001.

[SAT] SAT4J. `http://www.sat4j.org/`.

[SE08] Niklas Sörensson and Niklas Eén. MiniSat 2.1 and MiniSat++ 1.0 – Sat Race 2009 Editions. Sat Race 2008 Competition solver description, 2008.

[Seb07] Roberto Sebastiani. Lazy Satisability Modulo Theories. *JSAT*, 3(3-4):141–224, 2007.

[Sht01] Ofer Shtrichman. Pruning techniques for the SAT-Based bounded model checking problem. In *Correct Hardware Design and Verification Methods*, pages 58–70. 2001.

[SS97] Joao P. Marques Silva and Karem A. Sakallah. Robust search algorithms for test pattern generation. In *Proceedings of the*

27th International Symposium on Fault-Tolerant Computing (FTCS '97), page 152. IEEE Computer Society, 1997.

[Tse68] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic*, pages 115–125, 1968.

[War02] Henry S. Warren. *Hacker's Delight.* Addison-Wesley Professional, 2002.

[WFG$^+$07] Robert Wille, Görschwin Fey, Daniel Grobe, Stephan Eggersglub, and Rolf Drechsler. SWORD: a SAT like prover using word level information. In *Very Large Scale Integration, 2007. VLSI - SoC 2007. IFIP International Conference on*, pages 88–93, 2007.

[Wil92] M.V. Wilkes. EDSAC 2. *Annals of the History of Computing, IEEE*, 14(4):49–56, 1992.

[WKS01] Jesse Whittemore, Joonyoung Kim, and Karem Sakallah. SATIRE: a new incremental satisfiability engine. In *Proceedings of the 38th annual Design Automation Conference*, pages 542–545, Las Vegas, Nevada, United States, 2001. ACM.

[WS53] M. V. Wilkes and J. B. Stringer. Micro-Programming and the design of the control circuits in an electronic digital computer. *Mathematical Proceedings of the Cambridge Philosophical Society*, 49(02):230–238, 1953.

[WS97] Bob Wielinga and Guus Schreiber. Configuration-Design problem solving. *IEEE Expert: Intelligent Systems and Their Applications*, 12(2):49–56, 1997.

[WSK04] Markus Wedler, Dominik Stoffel, and Wolfgang Kunz. Arithmetic reasoning in DPLL-Based SAT solving. In *Proceedings of the conference on Design, automation and test in Europe - Volume 1*, page 10030. IEEE Computer Society, 2004.

[Yan02] Song Y. Yan. *Number Theory for Computing.* Springer-Verlag, 2nd edition, 2002.

[YM06] Yinlei Yu and Sharad Malik. Lemma learning in SMT on linear constraints. In *Theory and Applications of Satisfiability Testing - SAT 2006*, pages 142–155. 2006.

[ZKC01] Z. Zeng, P. Kalla, and M. Ciesielski. LPSAT: a unified approach to RTL satisfiability. In *Proceedings of the conference on Design, automation and test in Europe*, pages 398–402, Munich, Germany, 2001. IEEE Press.

[ZM03] Lintao Zhang and Sharad Malik. Extracting Small Unsatisfiable Cores from Unsatisfiable Boolean Formulas. Conference on Theory and Applications of Satisfiability Testing, 2003.