DECISION PROCEDURES
FOR
BIT-VECTORS, ARRAYS AND INTEGERS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Vijay Ganesh

September 2007

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Professor David L. Dill) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Professor Dawson Engler)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Dr. Natarajan Shankar)

Approved for the University Committee on Graduate Studies.

Abstract

Decision procedures, also referred to as satisfiability procedures or constraint solvers, that can check satisfiability of formulas over mathematical theories such as Boolean logic, real and integer arithmetic are increasingly being used in varied areas of computer science like formal verification, program analysis, and artificial intelligence. There are two primary reasons for this trend. First, many decision problems in computer science are easily translated into satisfiability problems in some mathematical theory. Second, in the last decade considerable progress has been made in the design and implementation of practical and efficient decision procedures. The improvement has been so dramatic that for many problems, translation to the satisfiability problem over some logic, followed by invocation of a decision procedure for that logic is often better than special-purpose algorithms. Also, for certain applications, decision procedures have proved to be an enabling technology. For example, the recent rapid adoption of decision procedures in automated bug-finding tools has enabled these tools to find many hard to detect bugs in real-world software, deemed infeasible a few years ago. As applications cover new ground, they generate larger, more complex formulas, and demand greater efficiency from the decision procedures they employ in deciding these formulas. This constant demand for more efficient, robust and practical decision procedures forms the motivation for the work presented here.

This dissertation discusses the design and implementation of novel, practical, robust and efficient decision procedures for the satisfiability problem of the theories of bit-vectors, arrays, and mixed real and integer linear arithmetic. The algorithms discussed here can demonstrably handle very large formulas obtained from real-world applications.

More specifically, this thesis makes three contributions. First, a new efficient decision

procedure for the theory of bit-vectors and arrays, called STP, is presented. The architecture of the STP tool is SAT-based, i.e., the input formula is efficiently translated into a Boolean formula after significant preprocessing, and this Boolean formula is then fed to a decision procedure for Boolean logic, often known as a SAT solver. The preprocessing steps consist of a solver for linear bit-vector arithmetic, and two algorithms based on the abstraction-refinement paradigm to handle very large arrays. The solver algorithm is online, based on a solve-and-substitute method, and can solve for whole bit-vectors or parts thereof. Online means that the solver can accept and process new input in an incremental fashion, without having to re-process all the previously received inputs. Algorithms employing the abstraction-refinement paradigm abstract away parts of the input in the hope that the resulting abstracted formula will be easier to decide, and that the answer will be correct. In the event the resulting answer is not correct, the algorithm suitably refines the abstracted formula to obtain a less abstract formula, and then tries to decide it. This process of refinement is repeated until the correct answer is obtained. The mix of aforementioned algorithms and the SAT-based approach used in STP has proved very effective in deciding extremely large formulas over the theory of bit-vector and arrays generated from real-world applications like bug-finding, formal verification, program analysis, and security analysis.

Second, a mixed real and integer linear arithmetic decision procedure is presented, that has been implemented as part of the CVC tool [SBD02]. The decision procedure is online and proof-producing. A decision procedure is called proof-producing if it can generate a mathematical proof of its work. Proof-producing decision procedures have the advantage that their work can be checked by external proof checkers. Also, these characteristics are very important in making the decision procedure efficient in the context of combining decision procedures of various mathematical theories to obtain a decision procedure for the union of these theories.

Third, a new decision procedure for the quantifier-free fragment of Presburger arithmetic is presented. The decision procedure is based on translating the satisfiability problem of this fragment into the language emptiness problem of a finite state automaton, which in turn is translated into a model-checking problem. This approach takes advantage of efficient Binary Decision Diagram (or BDD) implementations in modern symbolic model

checkers. BDDs are data structures that can efficiently represent finite state automata. Finally, various decision procedures for quantifier-free Presburger arithmetic are compared in a systematic fashion.

Acknowledgments

I am indebted to a great many people, without whose support I could not have completed this dissertation. First and foremost, I would like to thank my adviser, Professor David L. Dill. Besides being an excellent mentor and adviser, he supported me through some very difficult periods during my stay at Stanford. Dave's approach to solving problems, with an insistence on simplicity, elegance and effectiveness has had a lasting impact on me. Even when things seemed very difficult to me, he encouraged me to soldier on, and provided perspective based on his experience.

I would next like to thank Professor Dawson Engler. The EXE project, lead by Prof. Engler, provided the context in which a large portion of my thesis work was developed. I also learnt many a good lesson by simply observing Dawson go about solving hard problems in a practical way. Dr. Sergey Berezin is another person who helped make my stay at Stanford an excellent learning experience, for which I am very thankful to him. Also, I had a fantastic time engaging Sergey in discussions about all kinds of topics outside of work. I am grateful to Dr. Natarajan Shankar, with whom my interactions have always been pleasant and a great source of inspiration. I am continually impressed by the ease with which he navigates both theoretical and practical aspects of various problems from different domains. I would like to sincerely thank the members of my thesis defense committee, Professors Christopher Jacobs, Michael Genesereth, Dawson Engler, David L. Dill, and Dr. Henny Sipma, for agreeing to be on the committee at a rather short notice.

I have been lucky to have some wonderful peers who have supported me during my years at Stanford. In particular, I would like to thank Jacob Chang, Asheesh Khare, Cristian Cadar, Ted Kremenek, Debasis Sahoo, Arnab Roy and others for their warm friendship.

No acknowledgment is complete without the expression of gratitude towards one's family. I am deeply indebted to my loving mother, who has been and continues to be my greatest inspiration. Her confidence in me has kept me going through some very difficult times. I learnt from her the value of perseverance and honesty. I am also very grateful to my lovely wife Natalia, whose support has been critical in helping me complete this work. I would like to express my gratitude towards my late elder brother Chidambaram Raju, who inspired me to be inquisitive. A debt of gratitude goes the rest of my family. Finally, I dedicate this dissertation to my mother.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Decision procedures, also referred to as satisfiability procedures or constraint solvers, that can check *satisfiability* of formulas over mathematical theories such as Boolean logic, real and integer arithmetic, bit-vectors and arrays are increasingly being used in varied areas of computer science like formal verification, program analysis, and artificial intelligence. There are two primary reasons for this trend. First, many decision problems in computer science are easily translated into satisfiability problems in some mathematical logic theory. Second, in the last decade considerable progress has been made in the design and implementation of practical and efficient decision procedures. The improvement has been so dramatic that for many problems, translation to the satisfiability problem over some logic, followed by invocation of a decision procedure for that logic is often better than special-purpose algorithms. For example, decision procedure based algorithms have been proposed for problems in planning from artificial intelligence [KS92, KMS96, Kau06, Wel99], model checking from formal verification [BCCZ99, ADK$^+$05, PBG05], program analysis [Aik06], static bug detection [XA07], and other areas [BHZ06]. In all these examples, the algorithms are based on the idea of translating the problem effectively into the satisfiability problem over Boolean logic, followed by invocation of efficient decision procedure for Boolean logic [ES03, MMZ$^+$01, MSS99, BHZ06].

In the case of certain applications, decision procedures have proved to be an enabling technology, i.e., the applications are now able to tackle problems that were considered almost impossible to solve automatically before. For example, the recent rapid adoption of decision procedures in automated bug-finding tools has enabled these tools to find many hard to detect bugs in real-world software [CGP$^+$06, CGP$^+$07, BHL$^+$07]. Another application where decision procedures have become enablers is security analysis or malware analysis [NBFS06].

As applications cover new ground, they generate larger, more complex formulas, and demand greater efficiency from the decision procedures they employ in deciding these formulas. This constant demand for more efficient, robust and practical decision procedures forms the motivation for the work presented here.

1.1 Problem Statement

The problem under consideration in this dissertation is the design of efficient, practical and robust decision procedures which can decide the satisfiability problem of the quantifier-free theories of bit-vectors [Möl98, BKO$^+$07, BDL98], arrays [SBDL01, BMS06a], and mixed real and integer linear arithmetic [Pug91]. The satisfiability problem for each of these theories is known to be NP-complete. It is considered unlikely that uniformly efficient solutions for NP-complete problems can be found, and consequently it becomes very important that the proposed algorithms be practical and application driven. Implementation, data structure and architectural issues of these decision procedures are as important as the algorithms themselves, and any good solution must address them. Efficacy of the algorithms and architectural choices presented must be compared vis-a-vis other current competing tools on real-world applications. All these issues, i.e., the design, implementation and architectural issues associated with the construction of efficient and practical decision procedures for the above mentioned theories are addressed in this dissertation.

It is often very desirable to modularly combine decision procedures of different theories to obtain a decision procedure for the combined theory. Such combinations impose many unique efficiency constraints on the design of the individual decision procedures. The problem of making individual decision procedures efficient in a combination setting is also addressed in this thesis. Also, the question of designing proof-producing decision procedures is discussed.

1.2 The Choice of Mathematical Theories

The mathematical theories whose decision procedures are discussed here were chosen primarily due to the nature of the applications. The most important applications currently are automated analyses of computer programs, whether they be static or dynamic, for security or bug-finding, from hardware or software domain. Automated program analyses generate formulas from the expressions occurring in computer programs. Expressions occurring in programs are usually fixed-length vectors of bits, arrays, or can be simulated as terms over real and integer arithmetic. Also, it is often the case that these formulas do not have quantifiers in them. Hence, the focus here is on efficient decision procedures for theories such as the quantifier-free theory of bit-vectors and arrays, linear real and integer arithmetic.

1.3 Definitions

Before discussing the definitions of the satisfiability problem or decision procedures for various logical theories, a brief overview of first-order logic is provided here. The following overview is borrowed mostly from the well-known mathematical logic textbook by H. B. Enderton [End72].

1.3.1 First-Order Logic

First-order logic [End72, Hod93, Hug93] is a formal mathematical language whose sentences have precise meaning. The alphabet of this language includes two distinct kinds of symbols, logical symbols and parameters:

1. Logical Symbols

   - Parentheses: (,)
   - Quantifiers: $\forall$ (for all), $\exists$ (there exists)
   - Boolean Connectives: $\neg$ (not), $\wedge$ (and), $\vee$ (or)
   - Constant formulas: *true*, *false*
   - Equality: =

2. Non-logical Symbols

    - Variables
    - Function Symbols: Each function symbol has an associated arity, a non-negative integer that indicates the number of arguments required by the function (Functions with 0 arity are also called constants)
    - Relation Symbols: Each relation symbol also has an associated arity

The alphabet of this language is used to construct *terms* and *formulas*. Variables are usually represented by letters x, y, z.... It is always assumed that there are fixed rules, called *well-formedness* rules, which govern the construction of terms and formulas.

1. A term is recursively defined as

    - a variable
    - function application over terms

2. A formula is recursively defined as

    - *true*, *false*
    - equality between terms or application of an n-ary relation symbol to terms
    - If θ is a formula, then so is $\neg\theta$
    - If θ and ψ are formulas, then so are $\theta \wedge \psi$ and $\theta \vee \psi$

A formula constructed out of an application of a relation symbol over terms is called an *atomic formula* (The formulas *true* and *false* are also atomic). Atomic formulas and their negations are referred to as *literals*. Formulas constructed out of only literals and Boolean connectives are called *quantifier-free formulas*. A quantifier-free formula can be visualized as a tree whose leaves are literals, and whose internal nodes are the Boolean connectives. The non-leaf structure of the tree is often referred to as the *Boolean structure* of the formula.

Often quantifier-free formulas are written as $\theta(\bar{x})$, where $\bar{x}$ represents the list of variables occurring in the quantifier-free formula. Such variables are called free variables. If

$\theta(\bar{x})$ denotes a quantifier-free formula, then $\forall \bar{x} \theta(\bar{x})$ and $\exists \bar{x} \theta(\bar{x})$ denote quantified formulas. The variables $\bar{x}$ in $\forall \bar{x} \theta(\bar{x})$ are referred to as *bound* variables. Often, formulas can have some variables that are bound and others that are free. For example, the variables $\bar{y}$ in $\forall \bar{x} \theta(\bar{x}, \bar{y})$ are free variables, where $\theta(\bar{x}, \bar{y})$ is a quantifier-free formula. A *sentence* is a formula with no free variables.

1.3.2 Theories and Models

A *theory* is a set of first-order sentences. In this thesis, it is always assumed that a theory is *deductively closed* under a set of *axioms*. Informally, an axiom is a sentence that is self-evident. Also, a set of sentences is said to be deductively closed if every sentence in the set is *deduced* from the axioms that belong to that set (The reader is referred to Enderton's textbook [End72] for a more formal definition of deduction).

The *signature* of a theory is the set of function and relation symbols appearing in its sentences. A *structure* M over signature Σ is a tuple consisting of the following:

1. A set called the *domain* of M, often written as $dom(M)$
2. A mapping from each n-ary function symbol f in Σ to f^M, an n-ary function from $(dom(M))^n$ to $dom(M)$
3. A mapping from each n-ary predicate symbol p in Σ to p^M, an n-ary relation over the set $dom(M)$

For a given structure, M, and formula $\theta(\bar{x})$, a *variable assignment* ρ is a function which assigns to each variable in the formula an element of M. If, under a given variable assignment, the formula evaluates to true, we say that M and ρ *satisfy* $\theta(\bar{x})$ (written as $M \models_\rho \theta(\bar{x})$), and ρ is called a satisfying assignment. Such a structure is called a *model* of the formula $\theta(\bar{x})$. A formula $\theta(\bar{x})$ is said to be *satisfiable* if there exists some structure M and a satisfying assignment ρ such that $M \models_\rho \theta(\bar{x})$. If Γ is a set of formulas and θ is a formula, then $\Gamma \models \theta$ denotes that, for every structure and variable assignment satisfying each formula in Γ, the same structure and variable assignment satisfy θ.

Propositional logic (also called sentential logic or Boolean logic) is a logic whose formulas are constructed out of variables and Boolean connectives such as $\neg$, $\wedge$ and $\vee$ according to *well-formedness* rules. The variables (also called Boolean variables) can take values *true* or *false*.

1.3.3 Decision Procedures and The Satisfiability Problem

A *decision procedure* is an algorithm or computer program which accepts a formula in some logic or theory, and, in finite time, always correctly returns a value indicating whether the input formula is satisfiable or not. If such an algorithm exists for a certain theory, then that theory is said to be *decidable*. The problem of determining if a formula in some logic or theory is satisfiable or not, is referred to as the *satisfiability problem* for that logic or theory. Hence, we also say that a decision procedure for a theory solves its satisfiability problem.

Soundness and *completeness* are two very important properties required of decision procedures. Informally, soundness means that every step of the procedure is mathematically *correct*, and completeness means that the procedure will eventually produce all *facts* necessary to draw the correct conclusion. We say that a decision procedure is *sound* if whenever the decision procedure terminates and returns unsatisfiable on an input, then that input is indeed unsatisfiable. On the other hand, the decision procedure is said to be complete, if whenever the decision procedure terminates and returns satisfiable on an input, then that input is indeed satisfiable. It is also important to prove for each decision procedure that it terminates on all inputs.

The problem of determining *validity* of a formula is the dual of the satisfiability problem. A formula is said to be *valid* if it is true in all models. Otherwise, the formula is said to be *invalid*. Procedures that determine validity are often called automated prover or validity checkers. If a formula is true in all models, then its negation must be unsatisfiable. Hence, the validity of a formula that belongs to a decidable theory is often determined by first negating it, and then invoking a decision procedure on the negated formula.

Decision procedures for various mathematical theories have been studied by logicians since the early part of the 20th century [End72]. Their primary focus has been to determine if a certain mathematical theory is decidable or not. In computer science decision procedures were researched since the inception of the field. However, unlike in mathematical logic the focus has been more on efficiency of decision procedures and their applications. The most important decision procedure studied in computer science is the one for Boolean logic, known as Boolean SAT solvers or simply as SAT solvers [ES03, MMZ^{+}01, MSS99, BHZ06]. In the last decade, considerable progress has been made in the efficiency of Boolean SAT solvers thanks not only to improvements in algorithms and data structures, but also due to the increased performance of microprocessor and memory systems in modern computers.

In recent decades considerable attention has been paid to the problem of modularly combining decision procedures of certain theories to obtain a decision for the union of those theories [NO79, Sho84]. Such a combination of decision procedure will be referred to simply as the *combination*. The combination algorithm or implementation into which the individual decision procedures are embedded will be referred to as the *combination framework*. Two properties of an individual decision procedure important in the context of a combination are the *online* and *proof-producing* properties. Online (also called incremental) means that the decision procedure can accept and process new input in an incremental fashion, without having to re-process all the previously received inputs. A decision procedure is called proof-producing if it can generate a mathematical proof of its work.

1.4 Contributions At a Glance

The contributions of this thesis are:

1. A new efficient decision procedure for the theory of bit-vectors and arrays, called STP, is presented [GD07a, CGP^{+}06, CGP^{+}07] (Chapter 4). The architecture of the STP tool is SAT-based, i.e., the input formula is efficiently translated into an equivalent Boolean formula after significant preprocessing, and this Boolean formula is then fed to a SAT solver. The preprocessing steps consist of a solver for linear bit-vector arithmetic (also known as linear modular arithmetic or residue arithmetic), and two algorithms based on the *abstraction-refinement* paradigm to handle very large arrays. The solver algorithm is online, based on a solve-and-substitute method, and can

solve for whole bit-vectors or parts thereof. Online (also called incremental) means that the solver can accept and process new input in an incremental fashion, without having to re-process all the previously received inputs. Algorithms employing the abstraction-refinement paradigm abstract away parts of the input in the hope that the resulting abstracted formula will be easier to decide, and that the answer will be correct. In the event the resulting answer is not correct, the algorithm suitably refines the abstracted formula to obtain a less abstract formula, and then tries to decide it. This process of refinement is repeated until the correct answer is obtained. The mix of aforementioned algorithms and the SAT-based approach used in STP has proved very effective in deciding extremely large formulas over the theory of bit-vector and arrays generated from real-world applications like bug-finding, formal verification, program analysis, and security analysis.

2. A mixed real and integer linear arithmetic decision procedure is presented [BGD03, BGD02] based on the Omega-test [Pug91], that has been modified such that it can be efficient in the context of a combination framework [NO79, TH96, Sho84, Bar03] (Chapter 3). The particular combination under discussion is the CVC tool [SBD02]. The decision procedure is proof-producing. Such decision procedures have the important advantage that their work can be checked by external proof checkers. Also, the decision procedure is online. These two characteristics enable the decision procedure to be efficient in the context of a combination. Also, this decision procedure supports a predicate called the *int* predicate, in addition to the standard theory of mixed real and integer linear arithmetic.

3. A new decision procedure for the quantifier-free fragment of Presburger arithmetic is presented [GBD02] (Chapter 2). The decision procedure is based on translating the satisfiability problem of this fragment into the language emptiness problem of a finite state automaton, which in turn is translated into a model-checking problem. This approach takes advantage of efficient Binary Decision Diagram (or BDD) [Bry86] implementations in modern symbolic model checkers. BDDs are data structures that can effectively represent finite state automata. There have been prior attempts at using BDDs to represent the states of automata obtained from Presburger arithmetic

formulas [WB00]. Unlike previous methods, both states and transitions are represented using BDDs in our approach, and hence is more efficient. Finally, various decision procedures for quantifier-free Presburger arithmetic are compared in a systematic fashion. One of the important conclusions of this systematic comparison is that ILP-based approaches outperform other methods on most parameters.

1.5 Organization of this Dissertation

The rest of the dissertation is organized as follows. Chapter 2 discusses the automata-based approach for deciding quantifier-free fragment of Presburger arithmetic and comparisons with other methods for deciding the same. In Chapter 3, a decision procedure for mixed real and integer linear arithmetic is discussed, and characteristics important to making the decision procedure efficient in the context of a combination. In Chapter 4, STP, a decision procedure for the quantifier-free theory of bit-vectors and arrays is discussed. Comparison with other current competing tools is provided. In chapter 5, we conclude with some lessons learnt. In particular, the positive experience with abstraction-refinement based algorithms and SAT-based architectures is discussed. It is conjectured that such an approach may be efficient for deciding important theories other than bit-vectors and arrays.

Chapter 2

Automata and Presburger Arithmetic

2.1 Introduction

As noted previously, efficient decision procedures for logical theories can greatly help in the verification of programs or hardware designs. For instance, quantifier-free Presburger arithmetic [Pre27] has been used in RTL-datapath verification [BD02], and symbolic timing verification of timing diagrams [ABHL97].[1] However, the satisfiability problem for the quantifier-free fragment is known to be NP-complete [Opp78a]. Consequently, the search for practically efficient algorithms becomes very important.

Presburger arithmetic is defined to be the first-order theory of the structure

$$\langle \mathbb{Z}, 0, \leq, + \rangle$$

where $\mathbb{Z}$ is the set of integers. The satisfiability of Presburger arithmetic was shown to be decidable by Presburger in 1927 [Pre27]. This theory is usually defined over the natural numbers $\mathbb{N}$, but can easily be extended to the integers (which is important for practical applications) by representing any integer variable x by two natural variables: $x = x_+ - x_-$. This reduction has no effect on decidability or complexity results.

[1] In [ABHL97] Presburger formulas have quantifiers, but without alternation, and therefore, are easy to convert into quantifier-free formulas.

This chapter [2] focuses on quantifier-free Presburger arithmetic because many verification problems do not require quantification, and because the performance of decision procedures on quantifier-free formulas may be qualitatively different from the quantified case. This chapter has two primary goals: presentation of a new decision procedure based on symbolic model checking and comparison of the various approaches for deciding quantifier-free Presburger arithmetic (primarily conjunction of formulas) and their implementations.

The following distinct ways of solving the satisfiability problem of quantifier-free Presburger arithmetic have been proposed: Cooper's method [Coo72] and variants, Hodes' method [Hod72], Bledsoe's method [Ble75], the integer linear programming (ILP) based approaches, and the automata-based methods. There have been other methods proposed. For example, One could reduce Presburger arithmetic to another logic and use the corresponding decision procedure, e.g. reduction to propositional logic and using Boolean SAT solver. However, such an approach has not been shown to be competitive against the above mentioned approaches.

Also, recently there has been some work on abstraction-refinement based methods for quantifier-free Presburger arithmetic [KOSS04, LM07, Ses05]. However, these methods inherently rely on a Presburger arithmetic and/or Boolean SAT engine internally. One of the goals of this chapter is to focus on a systematic comparison of the performance of the more *basic* methods. It has been noted in [KOSS04] that abstraction based methods outperform others on real world examples.

Cooper's method is based on Presburger's original method for solving quantified formulas, with some efficiency improvements. Using Cooper's method on a quantifier-free formula requires the introduction of existential quantifiers and then eliminating them. This process results in an explosion of new atomic formulas, and hence the method is probably too inefficient to be competitive with other approaches. Hence, Cooper's method is not used in the comparisons discussed in this chapter.

In an excellent survey by Janicic et al. [JGB97] it was shown that variants of Cooper's method outperformed Hodes' method. Bledsoe's method is incomplete, and it is difficult to characterize syntactically the fragment it supports. Hence, we do not consider either

[2] I am very grateful to Dr. Sergey Berezin with whom I collaborated on the work presented in this chapter

Hodes' method or Bledsoe's method (or its variants by Shostak [Sho77]). The rest of the chapter focuses only on ILP and automata-based approaches.

Since atomic formulas in Presburger arithmetic are linear integer equalities and inequalities, it is natural to think of the *integer linear programming* (ILP) algorithms as a means to determine the satisfiability of quantifier-free formulas in Presburger arithmetic. ILP algorithms maximize an objective function, subject to constraints in the form of a conjunction of linear equalities and inequalities. Along the way, the system is checked for satisfiability (usually called *feasibility*), which is the problem of interest in this chapter.

There are many efficient implementations of ILP solvers available. We have experimented with the open source implementations LP_SOLVE [Ber02] by Michael Berkelaar, the OMEGA tool [Pug91] by William Pugh, and the commercial tool CPLEX [cpl]. The OMEGA tool is specifically tuned to solve integer problems, and is an extension of the Fourier-Motzkin linear programming algorithm [DE73] to integers [Wil76].

ILP-based approaches have the following drawbacks, all of which can and have been redressed with some additional effort. First, in the naïve approach, an arbitrary quantifier-free Presburger formula must first be converted to *disjunctive normal form* (DNF), followed by invocation of ILP on each disjunct, until either a satisfiable one is found or all disjuncts have been checked but none is satisfiable. If any of the disjuncts is satisfiable, then the entire formula is satisfiable. Otherwise, the formula is unsatisfiable. This conversion to DNF may lead to an exponential explosion of the formula size. However, this problem can be addressed to a good extent by using an efficient DPLL(T) implementation [GHN$^+$04, DdM06] to handle the *Boolean structure* of the input, without resorting to conversion of the arbitrary quantifier-free Presburger formula into DNF, and then invoking ILP solvers on each of the disjuncts.

The second drawback of the ILP-based methods is that many existing implementations lack the support for arbitrarily large integers and use native machine arithmetic (This problem has been remedied in many of the recent ILP-based tools). This has two consequences. Firstly, it is harder to make a fair comparison of the ILP tools with automata methods, since the two are not feature equivalent. The use of native machine arithmetic by ILP tools may result in wrong results, since native arithmetic is bounded while Presburger arithmetic is not. Secondly, the support for large integers may be crucial in certain hardware verification problems, where the solution set may have integers larger than the *int* types supported natively by the hardware. For instance, many current RTL-datapath verification approaches use ILP [JD01, BD02], but these approaches cannot be scaled with the bit-vector size in the designs. It must be mentioned that more recent ILP-based methods like Yices [DdM06] and Z3 [Bd07] use arbitrary precision arithmetic without any significant loss in efficiency. The work discussed in this chapter was done prior to the availability of tools such as Yices and Z3. However, the fact that no comparison was done with these more recent ILP-based tools does not negate the conclusions presented here. The reason is that the more recent tools are still not as efficient as some of the tools such as CPLEX that are used in the comparisons below.

The third drawback of ILP-based methods is that some are not complete for quantifier-free Presburger arithmetic, e.g. certain branch-and-bound techniques. This can be an issue for some applications that rely on decision procedure being complete. Despite these drawbacks ILP-based methods have been very popular in dealing with conjunctions of linear integer arithmetic formulas due to their efficiency.

Another approach for deciding quantifier-free Presburger arithmetic uses finite automata theory. The idea that an atomic Presburger formula can be represented by a finite-state automaton goes back at least to Büchi [Büc60]. Boudet and Comon [BC96] proposed a more efficient encoding than Büchi's. Later, Wolper and Boigelot [WB00] further improved the method of Boudet and Comon and implemented the technique in the system called LASH. Another automata-based approach is to translate the atomic formulas into WS1S (weak monadic second order logic with one successor) and then use the MONA tool [EKM98]. MONA is a decision procedure for WS1S and uses *Binary Decision Diagrams* (BDDs, [Bry86]) internally to represent automata.

In this chapter, a new automata-based approach for solving the satisfiability problem of quantifier-free Presburger arithmetic using symbolic model checking [CES86] is proposed and evaluated. The key idea is to convert the quantifier-free Presburger formula into a sequential circuit which is then model checked using SMV [McM93]. Experiments indicate that the SMV approach is quite efficient and more scalable on formulas with large coefficients than all the other automata-based techniques that we tested. The reason for this is the use of BDDs to represent *both the states and the transitions* of the resulting automaton.

Another factor which contributes to the efficiency is that SMV uses a highly optimized BDD package. In addition, the use of an existing tool saves a lot of implementation effort. The experiments required only a relatively small Perl script to convert Presburger formulas into the SMV language.

The other tools do not use BDDs for the states because they perform quantifier elimination by manipulating the automata directly. Namely, each quantifier alternation requires projection and determinization of the automaton. The use of BDDs for the states can make the implementation of the determinization step particularly hard.

We also compare various automata and ILP-based approaches on a suite of 400 randomly generated Presburger formulas. The random generation was controlled by several parameters, such as the number of atomic formulas, the number of variables, and maximum coefficient size. For every approach we identify classes of Presburger formulas for which it either performs very poorly or very efficiently. Only one similar comparison has been done previously in [SKR98]. However, their examples consist of a rather small set of quantified Presburger formulas obtained from real hardware verification problems. The goal of our comparison is to study the performance trends of various approaches and tools depending on different parameters of quantifier-free Presburger formulas.

The chapter is organized as follows. Section 2.2 explains the automata construction algorithms which are the same as in [WB00, BC96], except for the tighter bounds on the number of states of the automata. Section 2.3 then describes the implementation issues, the conversion of the satisfiability problem into a model checking problem, and construction of a circuit corresponding to the automaton. Section 2.4 provides experimental results and comparisons with other tools. Finally, Section 3.7 concludes the chapter with the discussion of experimental results and the future work.

2.2 Presburger Arithmetic

We define Presburger arithmetic to be the first-order theory over atomic formulas of the form

$$\sum_{i=1}^{n} a_i x_i \sim c, \tag{2.1}$$

where a_i and c are integer constants, x_i's are variables ranging over integers, and $\sim$ is an operator from $\{=, \neq, <, \leq, >, \geq\}$. The semantics of these operators are the usual ones. In the rest of the chapter we restrict ourselves to only *quantifier-free* fragment of Presburger arithmetic.

A *formula* f is either an atomic formula (2.1), or is constructed from formulas f_1 and f_2 recursively as follows:

$$f ::= \neg f_1 \mid f_1 \wedge f_2 \mid f_1 \vee f_2.$$

Throughout the chapter we use the following typographic conventions. We reserve boldface letters, e.g. $\mathbf{b}$, to represent column vectors and $\mathbf{b}^T$ to represent row vectors. The term *vector* shall always refer to a column vector unless specified otherwise. In this notation, $\mathbf{x}$ represents the vector of variables of the atomic formula:

$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

and $\mathbf{b}$ represents n-bit boolean column vectors. A row vector of coefficients in an atomic formula is denoted by $\mathbf{a}^T$:

$$\mathbf{a}^\mathbf{T} = (a_1,\, a_2,\, \ldots,\, a_n).$$

In particular, an atomic formula in the vector notation is written as follows:

$$f \equiv \mathbf{a}^\mathbf{T} \cdot \mathbf{x} \sim c,$$

where $\mathbf{a}^T \cdot \mathbf{x}$ is the scalar product of the two vectors $\mathbf{a}^T$ and $\mathbf{x}$.

We give the formal semantics of the quantifier-free Presburger arithmetic in terms of the sets of solutions. A *variable assignment* for a formula ϕ (not necessarily atomic) with n free variables is an n-vector of integers $\mathbf{w}$. An atomic formula f under a particular assignment $\mathbf{w}$ can be easily determined to be true or false by evaluating the expression $\mathbf{a}^T \cdot \mathbf{w} \sim c$.

A *solution* is a variable assignment $\mathbf{w}$ which makes the formula ϕ true. We denote the set of all solutions of ϕ by $\mathrm{Sol}(\phi)$, which is defined recursively as follows:

- if ϕ is atomic, then $\mathrm{Sol}(\phi) = \{\mathbf{w} \in \mathbb{Z}^n \mid \mathbf{a}^T \cdot \mathbf{w} \sim c\}$;
- if $\phi \equiv \neg\phi_1$, then $\mathrm{Sol}(\phi) = \mathbb{Z}^n - \mathrm{Sol}(\phi_1)$;
- if $\phi \equiv \phi_1 \wedge \phi_2$, then $\mathrm{Sol}(\phi) = \mathrm{Sol}(\phi_1) \cap \mathrm{Sol}(\phi_2)$;
- if $\phi \equiv \phi_1 \vee \phi_2$, then $\mathrm{Sol}(\phi) = \mathrm{Sol}(\phi_1) \cup \mathrm{Sol}(\phi_2)$.

To simplify the definitions, we assume that all atomic formulas of ϕ always contain the same set of variables. If this is not true and some variables are missing in one of the atomic formulas, then these variables can be added with zero coefficients.

2.2.1 Idea Behind the Automaton

The idea behind the automata-based approach is to construct a *deterministic finite-state automaton* (DFA) A_ϕ for a quantifier-free Presburger formula ϕ, such that the language $L(A_\phi)$ of this automaton corresponds to the set of all solutions of ϕ. When such an automaton is constructed, the satisfiability problem for ϕ is effectively reduced to the *emptiness problem* of the automaton, that is, checking whether $L(A_\phi) \neq \emptyset$.

If a formula is not atomic, then the corresponding DFA can be constructed from the DFAs for the sub-formulas using the complement, intersection, and union operations on the automata. Therefore, to complete our construction of A_ϕ for an arbitrary quantifier-free Presburger formula ϕ, it is sufficient to construct DFAs for each of the atomic formulas of ϕ.

Throughout this section we fix a particular atomic Presburger formula f:

$$f \equiv \mathbf{a}^T \cdot \mathbf{x} \sim c.$$

Recall that a variable assignment is an n-vector of integers $\mathbf{w}$. Each integer can be represented in the binary format in 2's complement, so a solution vector can be represented by a vector of binary strings.

We can now look at this representation of a variable assignment $\mathbf{w}$ as a *binary matrix* where each row, called a *track*, represents an integer for the corresponding variable, and each i^{th} column represents the vector of the i^{th} bits of all the components of $\mathbf{w}$. Equivalently, this matrix can be seen as a *string of its columns*, a string over the alphabet $\Sigma = \mathbb{B}^n$, where $\mathbb{B} = \{0, 1\}$. The set of all strings that together represent all the solutions of a formula f form a *language* L_f over the alphabet Σ. Our problem is now reduced to building a DFA for the atomic formula f that accepts exactly the language L_f.

The automaton A_f reads strings from left to right. The i^{th} letter in the string is an n-column vector of bits, that represents the i^{th} bits of values of the variables occurring in f. The first letter in the string represents the least significant bits (LSB) of the variable assignment (in binary representation) for the variables in f. If the value of the LHS of f is l after reading the string π up to the i^{th} letter, then after appending one more "letter" $\mathbf{b} \in \mathbb{B}^n$ to π on the right, the LHS value changes to $l' = 2l + \mathbf{a}^T \cdot \mathbf{b}$. Notice that only the original value of the LHS l and the new "letter" $\mathbf{b}$ are needed to compute the new value of the LHS l' for the resulting string. This property directly corresponds to the property of the transition relation of an automaton, namely, that the next state is solely determined by the current state and the next input letter.

Following the above intuition, we can define an automaton A_f as follows.The input alphabet is $\Sigma = \mathbb{B}^n$; The states of A_f are labeled by integers representing the values of the LHS of f. On an input $\mathbf{b} \in \Sigma$ the automaton transitions from a state l, representing the current value of f, to $l' = 2l + \mathbf{a}^T \cdot \mathbf{b}$, the new value of f after one more bit of the assignment to the variables has been read by the automaton. The set of accepting states are those states l that satisfy $l \sim c$. Special care has to be taken of the initial state s_{initial}. First, the state s_{initial} is not labeled by any element of $\mathbb{Z}$. Second, the empty string is interpreted to be a column vector of 0's. Thus, the value of the left hand side in the initial state must be equal to 0. The first "letter" read by A_f is the vector of sign bits, and, according to the 2's complement interpretation, the value of the LHS in the next state after s_{initial} must be $l = -\mathbf{a}^T \cdot \mathbf{b}$.

Notice that this automaton is not finite, since we have explicitly defined the set of states to be integers. Later we examine the structure of this infinite automaton and show how to trim the state space to a finite subset and obtain an equivalent DFA, similar to the one in Figure 2.1.

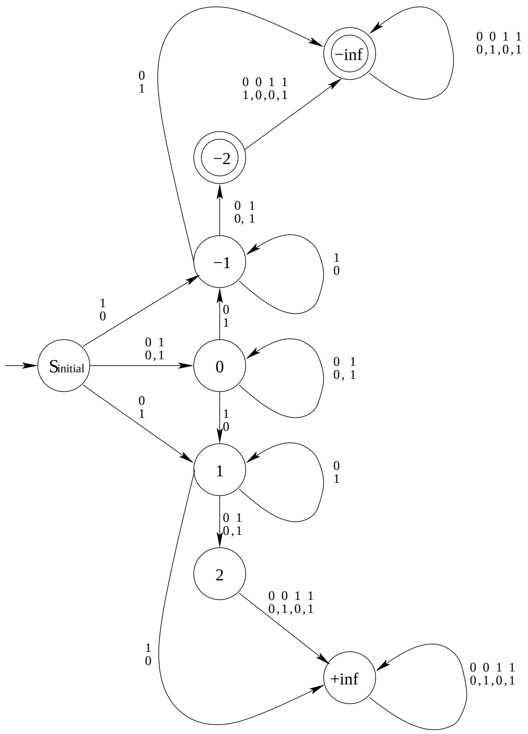

Figure 2.1: Example of an automaton for the atomic Presburger formula $x - y \leq -2$.

2.2.2 Formal Description of the Automaton

An (infinite-state) automaton corresponding to an atomic Presburger formula f is defined as follows:

$$A_f = (S,\, \mathbb{B}^n,\, \delta,\, s_{\text{initial}},\, S_{\text{acc}}),$$

where

- $S = \mathbb{Z} \cup \{s_{\text{initial}}\}$ is the set of states, $\mathbb{Z}$ is the set of integers and $s_{\text{initial}} \notin \mathbb{Z}$;
- s_{initial} is the start state;
- $\mathbb{B}^n$ is the alphabet, which is the set of n-bit vectors, $\mathbb{B} = \{0,\, 1\}$;
- The transition function $\delta : S \times \mathbb{B}^n \to S$ is defined as follows:

$$\begin{array}{lcl} \delta(s_{\text{initial}},\, \mathbf{b}) & = & -\mathbf{a}^T \cdot \mathbf{b} \\ \delta(l,\, \mathbf{b}) & = & 2l + \mathbf{a}^T \cdot \mathbf{b} \end{array}$$

where $l \in \mathbb{Z}$ is a non-initial state.

- The set of accepting states

$$S_{\text{acc}} = \{l \in \mathbb{Z} \mid l \sim c\} \cup \begin{cases} \{s_{\text{initial}}\} & \text{if } \mathbf{a}^T \cdot \mathbf{0} \sim c \\ \emptyset & \text{otherwise.} \end{cases}$$

In the rest of this section we show how this infinite automaton can be converted into an equivalent finite-state automaton. Intuitively, there is a certain finite range of values of the LHS of f such that if A_f transitions outside of this range, it starts *diverging*, or "moving away" from this range, and is guaranteed to stay outside of this range and on the same side of it (i.e. diverging to $+\infty$ or $-\infty$). We show that all of the states outside of the range can be collapsed into only two states (representing $+\infty$ and $-\infty$ respectively), and that those states can be meaningfully labeled as accepting or rejecting without affecting the language of the original automaton A_f.

For a vector of LHS coefficients $\mathbf{a}^T = (a_1, \ldots, a_n)$ define

$$\begin{aligned} ||\mathbf{a}^T||_- &= \sum_{\{i|a_i<0\}} |a_i| \\ ||\mathbf{a}^T||_+ &= \sum_{\{i|a_i>0\}} |a_i| \end{aligned}$$

Notice that both $||\mathbf{a}^T||_-$ and $||\mathbf{a}^T||_+$ are non-negative. Let $\mathbf{b}$ denote an n-bit binary vector, that is, $\mathbf{b} \in \mathbb{B}^n$.

Observe that $-\mathbf{a}^T \cdot \mathbf{b} \leq ||\mathbf{a}^T||_-$ for any value of $\mathbf{b}$, since the expression $-\mathbf{a}^T \cdot \mathbf{b}$ can be rewritten as

$$-\mathbf{a}^T \cdot \mathbf{b} = \left(\sum_{\{j \ | \ a_j<0\}} |a_j| b_j \right) - \left(\sum_{\{i \ | \ a_i>0\}} |a_i| b_i \right).$$

Therefore, the largest positive value of $-\mathbf{a}^T \cdot \mathbf{b}$ can be obtained by setting b_i to 0 whenever $a_i > 0$, and setting b_j to 1 when $a_j < 0$, in which case $-\mathbf{a}^T \cdot \mathbf{b} = ||\mathbf{a}^T||_-$. It is clear that any other assignment to $\mathbf{b}$ can only make $-\mathbf{a}^T \cdot \mathbf{b}$ smaller. Similarly, $\mathbf{a}^T \cdot \mathbf{b} \leq ||\mathbf{a}^T||_+$.

Lemma 1. *Given an atomic Presburger formula $\mathbf{a}^T \cdot \mathbf{x} \sim c$, a corresponding automaton A_f as defined in Section 2.2.2, and a current state of the automaton $l \in \mathbb{Z}$, the following two claims hold:*

1. *If $l > ||\mathbf{a}^T||_-$, then any next state l' will satisfy $l' > l$.*
2. *If $l < -||\mathbf{a}^T||_+$, then any next state l' will satisfy $l' < l$.*

Proof. **The upper bound (claim 1).** Assume that $l > ||\mathbf{a}^T||_-$ for some state $l \in \mathbb{Z}$. Then the next state l' satisfies the following:

$$\begin{array}{rcl} l' & = & 2l + \mathbf{a}^T \cdot \mathbf{b} \\ & \geq & 2l - ||\mathbf{a}^T||_- \\ & > & 2l - l = l. \end{array}$$

The lower bound (claim 2) is similar to the proof of claim 1. □

We now discuss bounds on the states of the automata based on Lemma 1. From this lemma it is easy to see that once the automaton reaches a state outside of

$$\left[\min(-||\mathbf{a}^T||_+, c),\ \max(||\mathbf{a}^T||_-, c)\right],$$

it is guaranteed to stay outside of this range and on the same side of it. That is, if it reaches a state $l < \min(-||\mathbf{a}^T||_+, c)$, then $l' < \min(-||\mathbf{a}^T||_+, c)$ for any subsequent state l' that it can reach from l. If the relation $\sim$ in f is an equality, then $l = c$ is guaranteed to be false from the moment A_f transitions to l onward. Similarly, it will be false forever when $\sim$ is $\geq$ or $>$; however it will always be true for $<$ and $\leq$ relations. In any case, either all of the states l of the automaton A_f below $\min(-||\mathbf{a}^T||_+, c)$ are accepting, or all of them are rejecting. Since the automaton will never leave this set of states, it will either always accept any further inputs or always reject. Therefore, replacing all states below $\min(-||\mathbf{a}^T||_+, c)$ with one single state $s_{-\infty}$ with a self-loop transition for all inputs and marking this state appropriately as accepting or rejecting will result in an automaton equivalent to the original A_f. Exactly the same line of reasoning applies to the states $l > \max(||\mathbf{a}^T||_-, c)$, and they all can be replaced by just one state $s_{+\infty}$ with a self-loop for all inputs.

Formally, the new *finite* automaton has the set of states

$$S = \big[\min(-||\mathbf{a}^T||_+,\, c),\ \max(||\mathbf{a}^T||_-,\, c)\big] \cup \{s_{\mathrm{initial}},\, s_{-\infty},\, s_{+\infty}\}.$$

Transitions within the range coincide with the transitions of the original (infinite) automaton A_f. If in the original automaton $l' = \delta(l, \mathbf{b})$ for some state l and input $\mathbf{b}$, and $l' > \max(||\mathbf{a}^T||_-, c)$, then in the new automaton the corresponding next state is $\delta'(l, \mathbf{b}) = s_{+\infty}$, and subsequently, $\delta'(s_{+\infty}, \mathbf{b}) = s_{+\infty}$ for any input $\mathbf{b}$. Similarly, if the next state $l' < \min(-||\mathbf{a}^T||_+, c)$, then the new next state is $s_{-\infty}$, and the automaton remains in $s_{-\infty}$ forever:

$$\begin{array}{lcl}
\delta'(s_{\mathrm{initial}}, \mathbf{b}) & = & -\mathbf{a}^T \cdot \mathbf{b} \\
\delta'(s_{+\infty}, \mathbf{b}) & = & s_{+\infty} \\
\delta'(s_{-\infty}, \mathbf{b}) & = & s_{-\infty} \\
\delta'(l, \mathbf{b}) & = & \begin{cases} s_{+\infty}, & \text{if } 2l + \mathbf{a}^T \cdot \mathbf{b} > \max(||\mathbf{a}^T||_-, c) \\ s_{-\infty}, & \text{if } 2l + \mathbf{a}^T \cdot \mathbf{b} < \min(-||\mathbf{a}^T||_+, c) \\ 2l + \mathbf{a}^T \cdot \mathbf{b}, & \text{otherwise.} \end{cases}
\end{array}$$

The accepting states within the range are those that satisfy the $\sim$ relation. The new "divergence" states are labeled accepting if the $\sim$ relation holds for some representative state. For instance, for a formula

$$\mathbf{a}^T \cdot \mathbf{x} < c$$

the state $s_{-\infty}$ is accepting, and $s_{+\infty}$ is rejecting. Finally, the initial state s_{initial} is accepting if and only if it is accepting in the original infinite automaton.

We can use the bounds from Lemma 1 to repeat the analysis done by Wolper and Boigelot [WB00] for the number of states of the automaton and obtain new bounds tighter by a factor of 2. Since we have to know the bounds in advance when constructing an SMV model, this saves one bit of state for every atomic formula. Asymptotically, of course, our new bounds stay the same as in the paper by Wolper and Boigelot [WB00].

2.3 Implementation

In the previous section we have shown a mathematical construction of a deterministic finite-state automaton corresponding to a quantifier-free Presburger formula f. In practice, building such an automaton explicitly is very inefficient, since the number of states is proportional to the value of the coefficients in $\mathbf{a}^T$ and the right hand side constant c and, most importantly, the number of transitions from each state is exponential (2^n) in the number of variables in f.

In our approach, we use an existing *symbolic model checker* SMV [McM93] as a means to build the symbolic representation of the automaton and check its language for emptiness. Symbolic model checking expresses a design as a finite-state automaton, and then properties of this design are checked by traversing the states of the automaton. In the past decade, there has been a lot of research in boosting the performance of model checkers. One of the notable breakthrough was in early 90s when binary decision diagrams [Bry86] (BDDs) were successfully used in model checking [McM93], pushing the tractable size of an automaton to as many as 10^{20} states and beyond [BCM$^+$92]. Therefore, it is only natural to try to utilize such powerful and well-developed techniques of handling finite-state automata in checking the satisfiability of Presburger formulas.

The obvious advantages of this approach is that the state-of-the-art verification engines such as SMV are readily available, they have well tuned BDD packages that can efficiently represent finite state automaton, and the only remaining task is to transform the emptiness problem for an automaton into a model checking problem efficiently. The key advantage of our approach with SMV is that we exploit the efficient BDD representation for both states and transitions of the automata, whereas in the other automata-based approaches like MONA or LASH the states are represented explicitly.

One of the attractive features of SMV is the expressiveness of its input language. SMV can directly handle integer variables of finite ranges and perform arithmetic operations on them. Therefore, the encoding of the automaton for an atomic Presburger formula f can be done in a straightforward way using the definitions in Section 2.2.2. However, the cost of evaluating these expressions in SMV is prohibitively high. Nonetheless, this approach is described here since it gives the reader an intuitive idea of the translation of a Presburger

```
MODULE main
VAR state: {start,lhs,minusInf,plusInf}; LHS: {-2,-1,0,1};
    inputX: {0,1}; inputY: {0,1};
DEFINE next_LHS := 2*LHS + inputX - inputY;
       next_state :=
         case state = start: lhs;
              state = lhs & next_LHS >  1: plusInf;
              state = lhs & next_LHS < -2: minusInf;
              1: state;
         esac;
       accept := state = minusInf | (state = lhs & LHS = -2);
ASSIGN init(state) := start; init(LHS) := 0;
       next(state) := next_state;
       next(LHS) := case state = start: (-1)*(inputX-inputY);
                         next_state = lhs: next_LHS;
                         1: LHS;
                    esac;
SPEC AG(!accept)
```

Figure 2.2: SMV code for $x - y \leq -2$ using variables of bounded integer type

formula into an automaton in the SMV language, and the conversion of satisfiability problem of the Presburger formula into the language emptiness problem of the corresponding automaton.

A state of the automaton is represented by the values of two state variables in SMV: `state` and `LHS`. The `state` variable defines the type of the state (initial, special states representing plus or minus infinity, or a state representing the value of the left hand side). The variable `LHS` represent the current value of the left hand side when `state=lhs`. The input variables `inputX` and `inputY` provide the vector of input bits. Since SMV does not directly support inputs from the environment, they are encoded as additional completely nondeterministic state variables. The initial values of the state variables define the initial state of the automaton. The next state function is defined for each of the state variables in the `next` assignments with the help of macros declared in the `DEFINE` section. Finally, the accepting states are those that satisfy the `accept` predicate.

The language of the original automaton is empty when no accepting state is reachable from the initial state. This property is formulated as a *safety property* in SMV:

$$\mathbf{AG}(\neg\texttt{accept}).$$

As noted above, due to the suboptimal way of handling arithmetic in SMV, such a direct implementation results in extremely high complexity of building the BDDs for the transition relation. Essentially, SMV has to enumerate explicitly all of the 2^n transitions in every state of the automaton, which makes all the efficiency gains due to BDDs rather pointless.

However, it was easy to see on the few simple examples which could be done this way, that once the transition relation is constructed, the resulting BDD size is usually small and the actual verification is very fast. This prompted us to search for other more efficient ways of encoding the automaton into SMV that would accelerate the construction of the transition relation.

Internally, SMV represents all the state variables as vectors of boolean variables. Similarly, the representation of the transition relation is a function[3] that takes boolean vectors of the current state variables and the inputs and returns new boolean vectors for the state variables in the next state.

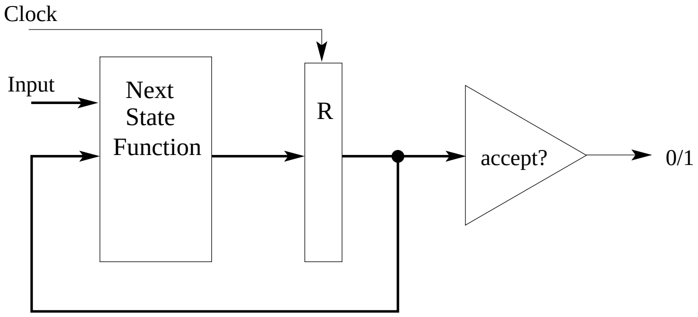

Figure 2.3: Circuit implementing a finite-state automaton.

Effectively, SMV builds an equivalent of a hardware circuit operating on boolean signals, as shown in Figure 2.3. The current state of the automaton is stored in the register R. The next state is computed by a combinational circuit from the value of the current state and the new inputs, and the result is latched back into the register R at the next clock cycle. A special tester circuit checks whether the current state is accepting, and if it is, the sequence of inputs read so far (or the string in automaton terminology) is accepted by the

[3] Strictly speaking, SMV constructs a *transition relation* which does not have to be a function, but here it is indeed a function, so this distinction is not important.

automaton (and represents a solution to f).

The property being checked is that the output of the circuit never becomes 1 for any sequence of inputs:

$$\mathbf{AG}(\mathrm{output} \neq 1)$$

If this property is true, then the language of the automaton is empty, and the original formula f is unsatisfiable. If this property is violated, SMV generates a counterexample trace which is a sequence of transitions leading to an accepting state. This trace represents a satisfying assignment to the formula f.

As noted above, the translation of the arithmetic expressions to such a boolean circuit is the primary bottleneck in SMV. Hence, providing the circuit explicitly greatly speeds up the process of building the transition relation.

A relatively simple Perl script generates such a circuit and the property very efficiently and transforms it into a SMV description. The structure of the resulting SMV code follows very closely the mathematical definition of the automaton, but all the state variables are explicitly represented by several boolean variables, and all the arithmetic operations are converted into combinational circuits (or, equivalently, boolean expressions). In particular, we use ripple-carry adders for addition, “shift-and-add” circuits for multiplication by a constant, and comparators for equality and inequality relations in the tester circuit.

2.4 Experimental Results

The primary purpose of our experiments is to study the performance of automata-based and ILP-based methods and their variations depending on different parameters of Presburger formulas. The tools and approaches that we picked are the following:

- Automata-based tools:
    - Our approach using the SMV model checker (we refer to it as “SMV”);
    - LASH [WB00], a direct implementation of the automata-based approach dedicated to Presburger arithmetic;

  - MONA [EKM98], an automata-based solver for WS1S and a general-purpose automata library.

- Approaches based on Integer Linear Programming (ILP):
  - LP_SOLVE, simplex-based open source tool with branch-and-bound for integer constraints;
  - CPLEX, one of the best commercial simplex-based LP solvers;
  - OMEGA [Pug91], a tool based on Fourier-Motzkin algorithm [Wil76].

The benchmarks consist of many randomly generated relatively small quantifier-free Presburger formulas. The examples have three main parameters: the number of variables, the number of atomic formulas (the resulting formula is a conjunction of atomic formulas), and the maximum value of the coefficients. For each set of parameters we generate 5 random formulas and run this same set of examples through each of the tools.

The results of the comparisons appear in Figures 2.4, 2.5, and 2.6 as plots showing how execution time of each automata-based tool depends on some particular parameter with other parameters fixed, and the success rate of all the tools for the same parameters. Each point in the runtime graphs represents a *successful run* of an experiment in a particular tool. That is, if a certain tool has fewer points in a certain range, then it means it failed more often in this range (ran out of memory or time, hit a fatal error, etc.). The ILP tools either complete an example within a small fraction of a second, or fail. Therefore the runtime is not as informative for ILP tools as the number of completed examples, and hence, only the success rates for those are shown. Figure 2.4 provides the runtime and the number of completed experiments for each tool, depending on the number of atomic formulas in each input. Figure 2.5 provides the runtime and the number of completed experiments for each tool, depending on the number of variables in a single atomic formula. SMV and LASH finish all of the experiments, hence there is no bar chart for those. Figure 2.6 provides the runtime and the number of completed examples for each tool, depending on the (maximum) values of the coefficients in a single atomic formula.

In the case of MONA, the only readily available input language is WS1S, and we have found that translating Presburger formulas into WS1S is extremely inefficient. Even rather simple examples which SMV and LASH solve in no time take significant time in MONA. Due to this inefficient translation, the comparison of MONA with other approaches is not quite fair. Therefore, it is omitted from the graphs and will not be considered in our discussion further.

LASH and SMV both have obvious strengths and weaknesses that can be easily characterized. SMV suffers the most from the number of atomic formulas, as can be seen from Figure 2.4 where the runtime is plotted as a function of the number of atomic formulas. The largest number of formulas it could handle in this batch is 11, whereas the other tools including LASH finished most of the experiments with up to 20 atomic formulas. This suggests that the implementation of the parallel composition of automata for atomic formulas in SMV is suboptimal.

Varying the number of variables (Figure 2.5) makes SMV and LASH look very much alike. Both tools can complete all of the experiments, and the runtime grows approximately exponentially with the number of variables and at the same rate in both tools. This suggests that the BDD-like structure for the transitions in LASH indeed behaves very similarly to BDDs in SMV.

However, since the number of states in the automata are proportional to the values of the coefficients, LASH cannot complete any of the experiments with coefficients larger than 4096 and fails on many experiments even with smaller values. SMV, on the other hand, can handle as large coefficients as 2^{30} with only a moderate increase of the runtime and the failure rate. We attribute this behavior to the fact that in SMV both the states and the transitions of the automata are represented with BDDs, while in LASH (and all the other available automata-based tools) the states are always represented explicitly.

Finally, we have to say a few words about the ILP-based methods. First of all, these methods are greatly superior to the automata-based in general, and they do not exhibit any noticeable increase in runtime when the number of variables or the number of formulas increase. The only limiting factor for ILPs are the values of the coefficients, which cause many failures and overflows starting at about 10^7, especially in LP_SOLVE (As mentioned previously this problem has been addressed in more recent ILP-based methods). Although all of the successful runs of the ILP-based tools are well under a fraction of a second, there are also many failures due to a non-terminating branch-and-bound search, overflow exceptions, and program errors. The OMEGA tool is especially notorious for segmentation faults, and its failure rate greatly increases when the values of the coefficients approach the limit of the machine-native integer or float representation.

Despite overall superiority of the ILP-based methods over the automata-based ones, there are a few cases where the ILP methods fail while the automata-based methods work rather efficiently. The most interesting class of such examples can be characterized as follows. The formula must have a solution in real numbers, but the integer solutions either do not exist or they are rather sparse in the *feasibility set* (the set of real solutions) of the formula. Additionally, the direct implementation of the branch-and-bound method is incomplete when the feasibility set is unbounded, since there are infinitely many integer points that have to be checked. This claim still holds to some extent even in the heuristic-rich top quality commercial tools such as CPLEX, and we have observed their divergence on a few examples that are trivial even for the automata-based techniques.

The OMEGA approach stands out from the rest of ILP tools since it is based on the Fourier-Motzkin method which is complete for integer linear constraints. Unfortunately, the readily available OMEGA tool is very unstable. In chapter 3 we discuss an implementation of the OMEGA approach, as part of the CVC tool, that is stable and supports arbitrary precision arithmetic. A comparison between the two tools is presented as well. However, as discussed there, the CVC implementation is not as efficient as the OMEGA tool, since the CVC implementation has to deal with the overhead of the combination framework and the cost of supporting arbitrary precision arithmetic.

Another common weakness of all of the ILP-based approaches is the limit of the coefficient and solution values due to the rounding errors of native computer arithmetic. It is quite easy to construct an example with large integer coefficients for which CPLEX returns a plainly wrong answer. Large coefficients can be extremely useful in hardware verification when operations on long bit-vectors are translated into Presburger arithmetic.

2.5 Conclusions

Efficient decision procedures for Presburger arithmetic are key to solving many formal verification problems. We have developed a decision procedure based on the idea of converting

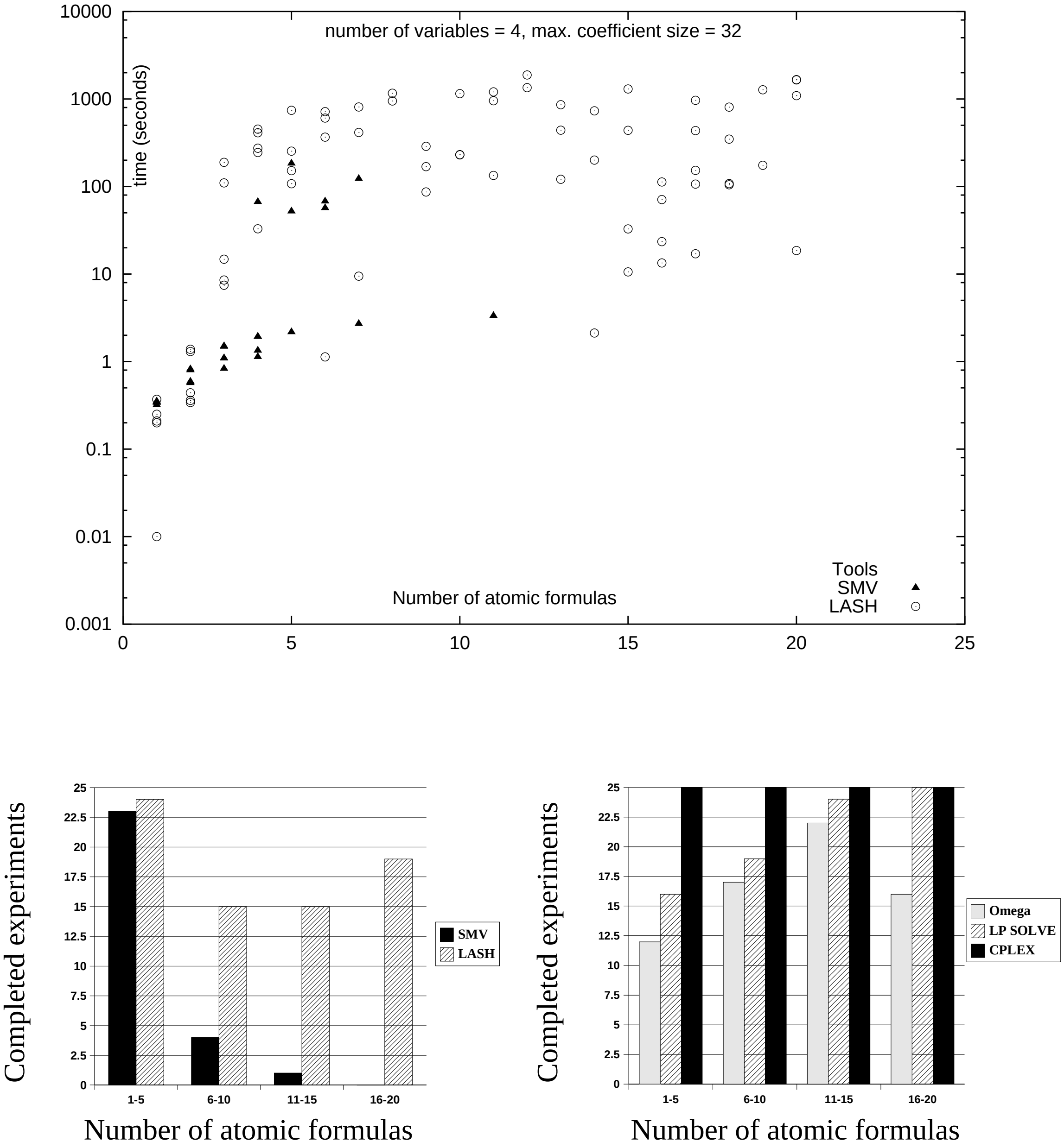

Figure 2.4: Comparison of Presburger arithmetic solvers, parameterized by number of atomic formulas

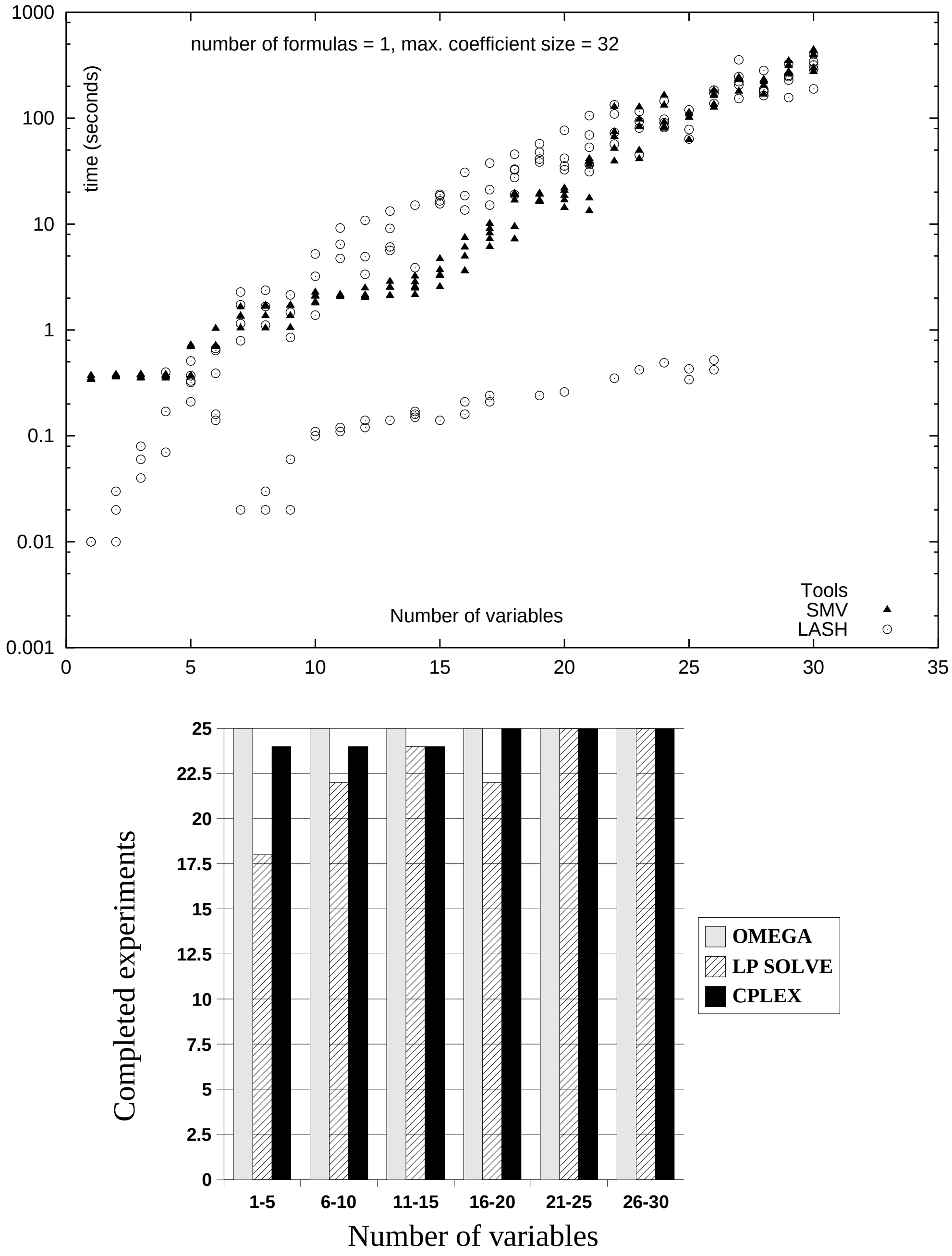

Figure 2.5: Comparison of Presburger arithmetic solvers, parameterized by number of variables

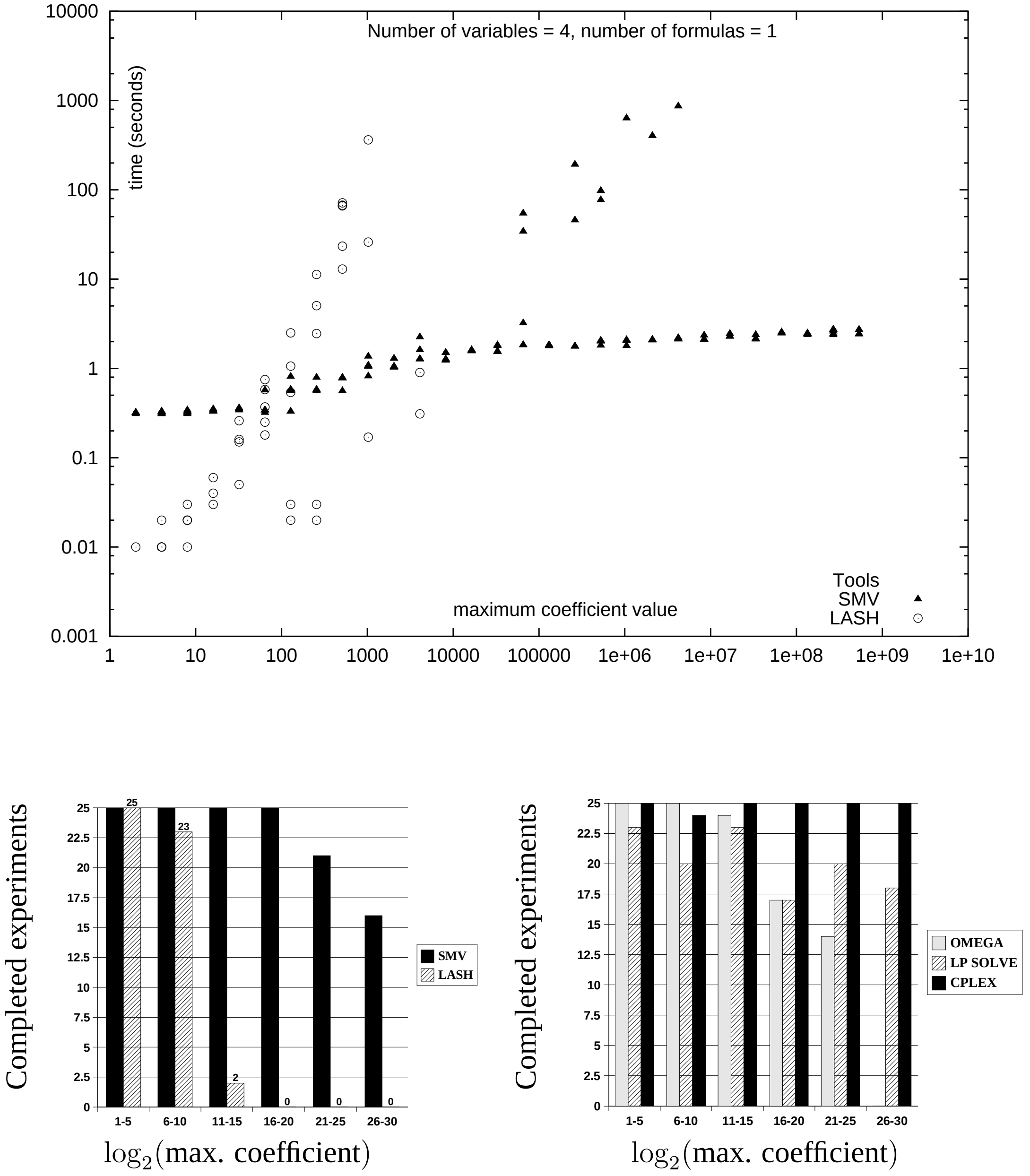

Figure 2.6: Comparison of Presburger arithmetic solvers, parameterized by maximum coefficient in atomic formulas

the satisfiability problem into a model checking problem. Experimental comparisons show that our method can be more efficient than other automata-based methods like LASH and MONA, particularly for formulas with large coefficients. In our approach we use BDDs both for the states and the transitions of the automata while LASH and MONA use BDDs or similar structures only for the transitions. As an additional theoretical result, we provide tighter bounds for the number of states of the automata. This makes our automaton construction in SMV even more efficient.

Another advantage of our approach is that converting the satisfiability problem into model checking problem requires very little implementation effort. We exploit the existing SMV model checker as a back-end which employs a very efficient BDD package. Therefore, the only effort required from us is the translation of a Presburger formula into the SMV input language.

In addition, we compare various automata and ILP-based approaches on a suite of parameterized randomly generated Presburger formulas. For every approach we identify classes of Presburger formulas for which it either performs very poorly or very efficiently. For instance, we found that the ILP-based tools are more likely to fail on examples with unbounded but sparse solution sets and cannot handle large coefficients due to the use of native machine arithmetic (Although this has been remedied in more recent ILP-based tools such as Yices [DdM06] and Z3 [Bd07] which use arbitrary precision arithmetic). The automata-based tools are not as sensitive to these parameters. On the other hand, ILP-based approaches scale much better on the number of variables and atomic formulas, parameters more relevant to applications than the maximum number of bits required to represent any coefficient in the input. This is a key advantage that ILP-based approaches have, and is the primary reason for their wide adoption for deciding linear integer arithmetic as opposed to the automata-based approaches.

One disadvantage of certain ILP-based methods (e.g. some branch-and-bound techniques) is that they are not complete for Presburger arithmetic. However, for many applications this is not such a strong disadvantage.

Within the automata-based approaches SMV scales better with the coefficients' size, but displays poorer performance for large number of atomic formulas when compared to LASH. Both perform equally well as the number of variables is increased.

The reason the other tools do not use BDDs for the states is because they perform quantifier elimination by manipulating the automata directly. Namely, each quantifier alternation requires projection and determinization of the automaton. The use of BDDs for the states can make the implementation of the determinization step particularly hard. This difference is one of the reasons for the relative efficiency of our approach.

The extension of our approach to full Presburger arithmetic can be done by combining it with the traditional quantifier elimination method [KK67]. This method introduces a new type of atomic formulas with the divisibility operator: $\mathbf{a}^T \cdot \mathbf{x} \,|\, c$, and our automaton construction can be easily extended to handle it. However, it is unclear if this would be cost-effective in comparison to other quantifier elimination methods.

Approaches that employ automata efficiently to decide theories other than Presburger arithmetic may benefit from the techniques presented here. In particular, the idea of using BDDs for representing both states and transitions of finite automata could make these methods more efficient.

Chapter 3

Mixed-Linear Arithmetic

3.1 Introduction

As noted in previous chapters, formal verification methods benefit greatly from efficient automatic decision procedures. In practice, constraints from different theories are often mixed together. For example, it is not uncommon to see a constraint like the following:

$$2 * y + x > z \wedge f(x) \neq z \rightarrow A[2 * y + x] > 0,$$

which belongs to the combined theory of arithmetic, arrays, and uninterpreted functions. Consequently, there is a need for a decision procedure for the union of theories.

It is natural to construct a combined decision procedure in a modular fashion out of decision procedures for individual theories. However, such a construction is not a straightforward task. Several combination methods like Nelson-Oppen [NO79], Shostak [Sho84], and their variants [RS01, BDS02a] have been developed in the past. All these methods impose certain requirements on the individual decision procedures in order to achieve a sound, complete, and efficient combination. Satisfying these requirements greatly improves the usability of the individual decision procedures.

One of the tools that combine decision procedures is the CVC Tool (*Cooperating Validity Checker*) [SBD02, BDS00] developed at Stanford. It is based on the framework of cooperating decision procedures developed by Barrett, Dill and Stump [BDS02a] which, in turn, is based on the Nelson-Oppen [NO79] and Shostak [Sho84, RS01] frameworks. Each decision procedure in the framework is responsible for solving satisfiability problem for only one particular theory and does not interact directly with the other theories. The implementation of the CVC tool prior to this work included decision procedures for the theory of uninterpreted functions [NO80], the theory of arrays [SBDL01], and the theory of datatypes [Opp78b]. Additionally, Boolean combinations of constraints are handled at the top level by a SAT solver [BDS02b] based on Chaff [MMZ$^+$01]. Thus, CVC as a whole serves as a decision procedure for the quantifier-free first-order theory of equality with uninterpreted functions, arrays, recursive datatypes, and linear arithmetic as discussed in this chapter.

This chapter [1] presents a decision procedure for the theory of linear arithmetic over integers and reals (also referred to as mixed linear arithmetic), that has been specifically designed to meet the requirements of combination framework presented in [BDS02a]. As with all combination methods, CVC imposes certain requirements on the individual decision procedures. In particular, each decision procedure must be *online* and *proof-producing*. Online means that a new constraint can be added to the set of existing constraints at any time, and the algorithm must be able to take this into account with only an incremental amount of work. proof-producing means that when the set of constraints is determined to be unsatisfiable, the algorithm must also produce a proof of this fact. The online and proof-producing requirements imposed on the individual decision procedures are not specific to the CVC framework. These characteristics are of general interest to any combination framework discussed in the literature so far. Proof-producing decision procedures are of independent interest as well, since their work can be checked by external proof checkers.

Additionally, the theory of linear arithmetic is extended with the predicate $\mathsf{int}(t)$, which holds when a real-valued arithmetic term t is an integer. The decision procedure must be able to handle constraints of the form $\mathsf{int}(t)$ and $\neg\mathsf{int}(t)$ in addition to the usual linear arithmetic constraints.

The reasons for the above requirements can be better understood from the architecture of CVC as shown in Figure 3.1. At a very high level, CVC can be viewed as a Boolean

[1] I am very grateful to Dr. Sergey Berezin with whom I collaborated on the work presented in this chapter

SAT solver which solves the satisfiability problem of the Boolean skeleton of the first-order formulas. Each time the SAT solver makes a decision, a new constraint is produced, which is submitted to the appropriate decision procedure. Since decisions are dynamic, decision procedures must be able to receive a new constraint and process it efficiently. This is the motivation for the online requirement imposed on the decision procedure.

Modern SAT solvers [SS96, MSS99, MMZ$^+$01] use *conflict clauses* and *intelligent backtracking* to enhance performance. When the SAT solver in the CVC framework backtracks it is imperative that the individual decision procedures in the framework also have to retract some constraints dynamically. Hence, the decision procedures must support the backtracking operation. From an implementation point of view, this means that all the internal data structures in the decision procedure must be backtrackable.

To construct a conflict clause, one needs to identify those decisions made by the SAT solver which lead to a contradiction. One possible way of identifying such decisions is to extract them from the proof that the decision procedure constructs when it derives a contradiction. This explains one of the motivations for proof production in the decision procedures in CVC. A more important motivation is that the proofs can be checked by an external proof checker [Stu02], thus increasing the confidence in the results produced by CVC.

The $\mathsf{int}(t)$ predicate arises from the translation of CVC's input language which supports partial functions [Far90] to classical first-order logic. The reason for the translation is that the individual decision procedures inside CVC accept formulas only in classical first-order logic. However, certain applications require that CVC allow partial functions in its input language. More details about partial functions in the input language of CVC, and its successor tool CVC Lite [BB04], can be found in [BBS$^+$04]. It is sufficient to note here that the introduction of the $\mathsf{int}(t)$ predicate in the language of the mixed-linear arithmetic decision procedure enables support for partial functions in CVC.

In summary, a decision procedure is much more useful to the research community when it can be combined with other decision procedures. This combination imposes additional requirements on the decision procedure. Namely, it must be online and proof- producing, and all of its internal data structures must be backtrackable.

The contribution described in this chapter is a decision procedure for mixed-integer linear arithmetic designed to meet the above requirements, and thus, fit the CVC framework. The decision procedure is based on an existing algorithm called *Omega-test* [Pug91], which is extended to be online, proof-producing, and handle the $\mathsf{int}()$ predicate. Additionally, this implementation supports arbitrary precision arithmetic based on the GMP library [GMP07]. The choice of Omega-test over other algorithms for solving mixed-integer linear arithmetic problems (simplex, interior point method [BT97], earlier versions of Fourier-Motzkin elimination [Wil76], etc.) is driven by its simplicity and practical efficiency for a large class of verification problems. In particular, proof production is relatively easy to implement for the Omega-test.

This chapter is organized as follows: Section 3.2 describes the Fourier-Motzkin algorithm for solving a system of linear inequalities and equalities over the reals. Section 3.3 discusses the extension of the Fourier-Motzkin algorithm to the case system of linear equalities and inequalities over the integers. The online versions of Fourier-Motzkin algorithms for the reals and integers are discussed in Sections 3.4 and 3.5 respectively. Section 3.6 discusses proof production, and we conclude in Section 3.7.

3.2 Fourier-Motzkin Elimination over the Reals

The problem can be described as follows: Given a system of linear inequality constraints over real-valued variables of the form

$$\sum_{i=1}^{n} a_i x_i + c < 0 \quad \text{or} \quad \sum_{i=1}^{n} a_i x_i + c \leq 0,$$

where a_i's and c are rational constants, determine if this system is satisfiable. Here and in the rest of the chapter, linear arithmetic terms are often denoted as α, β, γ, or t, possibly with subscripts. In this section, we do not consider equalities in the input, since any equality can be solved for some variable and instantiated into the other constraints, thus obtaining an equivalent system without the equality.

Preliminaries

For the sake of terminological clarity in the rest of the chapter, a variable is said to be ***eliminated*** if it is eliminated by solving an equality constraint and substituting the result in all the other constraints. Although the input is restricted only to inequalities as mentioned above, equalities may be derived in the process of reasoning about inequalities, and hence need to be eliminated. When the variable is eliminated using the Fourier-Motzkin reasoning on inequalities, we say that such a variable is ***projected***.

Throughout the chapter we assume that all the constants and coefficients are rational. Although we often refer to variables as real-valued, it is well-known that under the above conditions, the system of linear constraints is satisfiable in reals if and only if it is satisfiable in rationals.

The Algorithm

Intuitively, Fourier-Motzkin elimination procedure [DE73] iteratively projects one variable x by rewriting the system of inequalities into a new system without x which has a solution if and only if the original system has a solution (i.e. the two systems are *equisatisfiable*). This process repeats until no variables are left, at which point all of the constraints become inequalities over numerical constants and can be directly checked for satisfiability.

More formally, the projection procedure is the following. First, pick a variable present in at least one inequality, say x_n. All the inequalities containing this variable are then rewritten in the form $\beta < x_n$ or $x_n < \alpha$ (or $\beta \leq x_n$ or $x_n \leq \alpha$), depending on the sign of the coefficient a_n. This creates three types of constraints: those with x on the right, with x on the left, and those without x:

$$\left\{\begin{array}{c}\beta_1 < x_n\\ \vdots\\ \beta_{k_1} < x_n\end{array}\right. \qquad \left\{\begin{array}{c}x_n < \alpha_1\\ \vdots\\ x_n < \alpha_{k_2}\end{array}\right. \qquad \left\{\begin{array}{c}\gamma_1 < 0\\ \vdots\\ \gamma_{k_3} < 0.\end{array}\right. \tag{3.1}$$

where some inequalities may be non-strict ($\leq$ instead of $<$). If this system of constraints has a solution, then x_n must satisfy

$$\max(\beta_1, \ldots, \beta_{k_1}) < x_n < \min(\alpha_1, \ldots, \alpha_{k_2}).$$

Since real numbers are dense, such x_n exists if and only if the following constraint holds:

$$\max(\beta_1, \ldots, \beta_{k_1}) < \min(\alpha_1, \ldots, \alpha_{k_2}),$$

(or $\max(\beta_1, \ldots, \beta_{k_1}) \leq \min(\alpha_1, \ldots, \alpha_{k_2})$ in the case where both of the original inequalities are non-strict). This constraint can be equivalently rewritten as

$$\left\{ \beta_i < \alpha_j \qquad \text{for all } i = 1 \ldots k_1,\ j = 1 \ldots k_2, \right. \tag{3.2}$$

which is again a system of linear inequalities. We call them the *shadow constraints*, because they define an $n-1$-dimensional shadow of the n-dimensional shape defined by the original constraints (3.1). The shadow constraints (3.2) combined together with $\gamma_l < 0$ comprise a new system of constraints which is equisatisfiable with (3.1), but does not contain the variable x_n. This process can now be repeated for x_{n-1}, and so on, until all the variables are projected.

Notice, that in the case of non-strict inequalities, it may be happen that $\alpha = \beta$, in which case a new equality can be derived: $x = \alpha$ (or, equivalently, $x = \beta$), and the variable can be eliminated more efficiently. This fact is not necessary for completeness of the offline algorithm presented in this section, however, it becomes important when this decision procedure becomes part of the CVC combination framework.

Observe, that for a system of m constraints each elimination step may produce a new system with up to $(m/2)^2$ constraints. Therefore, eliminating n variables may, in the worst case, create a system of $4 \cdot (m/4)^{2^n}$ constraints. Thus, the decision procedure for linear inequalities based on Fourier-Motzkin even in the case of real variables has a doubly exponential worst case complexity in the number of variables.

Although much more efficient algorithms are known for real linear arithmetic (e.g.

polytime interior-point methods), the Fourier-Motzkin algorithm was chosen primarily because it is relatively easy to make this algorithm (both for the reals and integers) online and proof-producing, characteristics important in the context of combinations. Also, the choice of Fourier-Motzkin based algorithm (Omega test) for the integers was also a factor in making the choice in favor of Fourier-Motzkin algorithm for the reals.

The Omega test for the integers was chosen based on our experience with it. Chapter 2 describes the performance of the Omega test as compared with other methods for integer linear arithmetic. The Omega test performed very well. Also, among the ILP-based approaches, the Omega test was clearly known to be complete for integer linear arithmetic.

3.3 The Omega Test

Our version of the extension of Fourier-Motzkin algorithm to the integer case is largely based on the Omega approach [Pug91], with a few important differences. First, the input for the procedure described here is a system of *mixed integer linear constraints* which, in addition to linear equalities and (strict) inequalities may also contain $\mathsf{int}(t)$ or $\neg\mathsf{int}(t)$ for any linear term t, meaning that the linear term t is restricted to only integer (respectively, fractional) values. This enables support for linear real arithmetic, and also partial functions as mentioned before. The second important difference is that Omega test is offline, while the procedure here is online and can produce proofs.

Any system of mixed integer linear constraints may be converted to an equisatisfiable system of constraints with only equalities, inequalities, and predicates of the form $\mathsf{int}(x)$, where x is a variable. If the term t is not a variable, the constraint $\mathsf{int}(t)$ is satisfiable iff $\mathsf{int}(z) \wedge z = t$ is satisfiable, where z is a new variable. Furthermore, $\neg\mathsf{int}(t)$ is satisfiable for any term t iff $\mathsf{int}(y) \wedge y < t < y + 1$ is satisfiable for a new variable y.

3.3.1 Elimination of Equalities

As in the case of reals, all the equalities are eliminated first. If an equality contains a variable x that is not an integer, then the equality is solved for this variable, and x is eliminated from the system. It is efficient to eliminate all variables known to be non-integers before integer variables are eliminated, since the step for eliminating real variables is computationally cheap, and this process may quickly decide the input.

Now suppose that an equality contains only integer variables:

$$\sum_{i=1}^{n} a_i x_i + c = 0. \tag{3.3}$$

Here we use the same variable elimination algorithm as in the Omega-test [Pug91]. If x is the only variable in equation (3.3), then there is only one value of x which can satisfy this equality constraint, namely $x = -(c/a)$. If this value is integer, we substitute it for x, and otherwise the system is unsatisfiable.

If there is more than one variable in equation (3.3), the equality is normalized such that all the coefficients a_i and the free constant c are relatively prime integers, which can always be done when the coefficients are rational numbers. If, after the normalization, there is a variable x_k whose coefficient is $|a_k| = 1$, then we simply solve for x_k and eliminate it from the rest of the system. If no variable x_k has coefficient whose absolute value is 1, then the algorithm becomes more complicated.

The goal of the following steps of the algorithm is to drive down the coefficient of some suitably chosen x_k to 1 or -1, while at the same time maintaining equisatisfiability between the normalized equation above and the equation obtained by applying these steps. Driving the absolute value of the coefficient of x_k to 1 enables the elimination of x_k, and thus the equation, from the input system of constraints.

Pick a variable x_k whose coefficient a_k is the smallest by the absolute value and define

$$m = |a_k| + 1.$$

Define also a modulus operation with the range $\left[-\frac{m}{2}, \frac{m}{2}\right)$ as follows:

$$a \,\mathbf{mod}\, m = a - m \left\lfloor \frac{a}{m} + \frac{1}{2} \right\rfloor .$$

The important properties of $\mathbf{mod}$ are that $a_k \,\mathbf{mod}\, m = -\mathrm{sign}(a_k)$, where the function

$\mathrm{sign}(a_k)$ represents the sign of a_k ($\mathrm{sign}(a_k)$ can only take values 1 or -1). The other important property of the $\mathbf{mod}$ function is that it distributes over addition and multiplication. All these properties are easy to check.

Informally speaking, the next steps of the algorithm apply the modulus operation $\mathbf{mod}$ to both sides of the normalized equation, while maintaining equisatisfiability. Since this mod operation distributes over addition and multiplication, it is easy to apply it to every term occurring in the equation, and normalize the resultant terms. Also, since $a_k \mathbf{mod}\, m = -\mathrm{sign}(a_k)$, the goal of driving the absolute value of the coefficient of x_k to 1 is achieved. The only complication is that a new integer variable is introduced in the process. However, as we shall see below, the algorithm eventually eliminates all variables including the ones that are introduced.

The remaining steps of the algorithm are as follows. For a new integer variable σ, introduce two new constraints into the system:

$$\mathsf{int}(\sigma) \quad \text{and} \quad \sum_{i=1}^{n}(a_i \,\mathbf{mod}\, m)x_i + (c \,\mathbf{mod}\, m) = m\sigma. \tag{3.4}$$

The second constraint is derivable from the equation (3.3) by applying $\cdot \,\mathbf{mod}\, m$ on both sides of (3.3) and propagating $\mathbf{mod}$ over addition and multiplication, and then applying it to each of the coefficients. Hence, the system remains equisatisfiable with the original (To be technically correct, the new variable σ must be existentially quantified in the system of constraints (3.4), for it to be equisatisfiable with the equation (3.3). However, eventually this variable is eliminated, and hence the quantifier may be omitted for the sake of readability and clarity).

Since $a_k \,\mathbf{mod}\, m = -\mathrm{sign}(a_k)$, and $-\mathrm{sign}(a_k)$ can only take values 1 or -1, the equation in (3.4) can be solved for x_k, and thus eliminated from the system. Isolating x_k yields the following equation:

$$x_k = -\mathrm{sign}(a_k)m\sigma + \sum_{i\in[1..n]-\{k\}} \mathrm{sign}(a_k)(a_i \,\mathbf{mod}\, m)x_i + \mathrm{sign}(a_k)(c \,\mathbf{mod}\, m). \tag{3.5}$$

Substituting the result into the original equation (3.3) yields the following:

$$-|a_k|m\sigma + \sum_{i\in[1..n]-\{k]} (a_i + |a_k|(a_i \,\mathbf{mod}\, m))x_i + c + |a_k|(c \,\mathbf{mod}\, m) = 0.$$

Since $|a_k| = m - 1$, it is the same as

$$\begin{aligned} -|a_k|m\sigma + \sum_{i\in[1..n]-\{k\}} ((a_i - (a_i \,\mathbf{mod}\, m)) + m(a_i \,\mathbf{mod}\, m))x_i \\ +(c - (c \,\mathbf{mod}\, m)) + m(c \,\mathbf{mod}\, m) &= 0. \end{aligned}$$

From the definition of $\mathbf{mod}$ we have,

$$a - (a \,\mathbf{mod}\, m) = m \left\lfloor \frac{a}{m} + \frac{1}{2} \right\rfloor .$$

Instantiating it in the above equation and dividing by m produces a new normalized constraint:

$$-|a_k|\sigma + \sum_{i\in[1..n]-\{k\}} a'_i x_i + c' = 0, \tag{3.6}$$

where

$$a'_i = \left\lfloor \frac{a_i}{m} + \frac{1}{2} \right\rfloor + (a_i \,\mathbf{mod}\, m)$$

and

$$c' = \left\lfloor \frac{c}{m} + \frac{1}{2} \right\rfloor + (c \,\mathbf{mod}\, m)$$

The new system (which is the original system with x_k eliminated using (3.5), and (3.3) rewritten as (3.6)) contains the same number of variables as the original one. Moreover, the new coefficients a' in (3.6) are guaranteed to decrease by absolute value compared to (3.3), namely $|a'_i| \leq \frac{2}{3}|a_i|$, except for the coefficient of σ which remains as large as that of x_k. This ensures that repeating the process above will eventually result in equality (3.6) having some variable with a coefficient 1 or -1. That variable can then be eliminated, reducing the overall dimensionality.

Observe that the new variable σ always takes the value of

$$\sigma = \left\lfloor \left(\sum_{i=1}^{n} (a_i \,\mathbf{mod}\, m) x_i + (c \,\mathbf{mod}\, m) \right) / m \right\rfloor .$$

3.3.2 Projecting Variables from Integer Inequalities

After eliminating all of the equalities, we are left with the system of inequalities over real and integer variables. Similar to the equality case, all the remaining real variables are projected first with the standard Fourier-Motzkin elimination procedure, resulting in a system of inequalities with only integer variables.

At this point, all the inequalities are non-strict [2] normalized to make the coefficients be relatively prime integers, and a variable x_n is chosen for projection. Since x_n has an additional integral constraint, we cannot simply divide the inequality by the coefficient a_n unless it is 1 or -1, and in general, the system of inequalities is rewritten in the equivalent form, very much like in (3.1), only the coefficients of x_n are preserved:

$$\left\{\begin{array}{c} \beta_1 \le b_n^1 x_n \\ \vdots \\ \beta_{k_1} \le b_n^{k_1} x_n \end{array}\right. \qquad \left\{\begin{array}{c} a_n^1 x_n \le \alpha_1 \\ \vdots \\ a_n^{k_2} x_n \le \alpha_{k_2} \end{array}\right. \qquad \left\{\begin{array}{c} \gamma_1 \le 0 \\ \vdots \\ \gamma_{k_3} \le 0, \end{array}\right. \tag{3.7}$$

where the coefficients a_i^j and b_i^j are positive integers. Similar to the original Fourier-Motzkin construction, for each pair of inequalities $\beta \le b x_n$ and $a x_n \le \alpha$, which is equivalent to

$$a\beta \le ab x_n \le b\alpha, \tag{3.8}$$

the *real shadow constraint* is constructed:

$$a\beta \le b\alpha. \tag{3.9}$$

As in the case of reals, it may happen that $a\beta = b\alpha$, in which case $abx_n = a\beta = b\alpha$ is a new (derived) equality, and it is more efficient to eliminate x_n rather than project. In the

[2] A strict integer inequality $\alpha < \beta$ can be converted to an equivalent non-strict one: $\alpha \le \beta - 1$.

context of combination framework of CVC, this new equation also becomes important for completeness.

However, the real shadow is a necessary but not a sufficient condition for the satisfiability of (3.8), since there might not be an integer value abx_n between $a\beta$ and $b\alpha$, even if there is a real one. If the constraint 3.8 is satisfiable, then in addition to the real shadow, at least one point $ab \cdot i$ must exist between $a\beta$ and $b\alpha$ for some integer i. A sufficient (but not necessary) condition is to demand that the gap between $a\beta$ and $b\alpha$ is at least $ab - 1$ wide:

$$\mathbf{D} \equiv b\alpha - a\beta \geq ab - 1. \tag{3.10}$$

This constraint is called the *dark shadow constraint* (the object is "thick enough" to contain an integer point, and therefore casts a darker shadow; the term *dark shadow* is from [Pug91]). The dark shadow constraint is sufficient, but not necessary for an integer solution of x_n to exist. Therefore, if equation (3.10) makes the system unsatisfiable, we have to look for an integer solution outside of $\mathbf{D}$, i.e. in the *gray shadow*:

$$b\alpha - a\beta \leq ab - 2, \tag{3.11}$$

or equivalently:

$$b\alpha \leq a\beta + ab - 2.$$

Following the construction in the Omega-test [Pug91], $b\alpha$ on the right-hand side of (3.8) is replaced by the larger $a\beta + ab$, and dividing the result by a yields the following:

$$\beta \leq bx_n \leq \beta + b - \lceil 2/\alpha \rceil$$

(which is equivalent to $\beta \leq bx_n \leq \beta + b - 2/\alpha$ for $\alpha > 0$ in integer arithmetic). This means that if there is an integer solution to x_n, it must satisfy $bx_n = \beta + i$ for some $0 \leq i \leq b - \lceil 2/\alpha \rceil$, since β contains only integer variables with integer coefficients. We then try each such i in succession until a solution is found. In other words, the *gray shadow constraint* is:

$$\mathbf{G}_b \equiv \bigvee_{i=0}^{b-\lceil 2/\alpha \rceil} bx_n = \beta + i.$$

Alternatively, the equation (3.11) can be written as follows:

$$b\alpha - ab + 2 \leq a\beta.$$

Replacing $a\beta$ with the smaller value $b\alpha - ab$ on the left-hand side of (3.8) yields the following:

$$b\alpha - ab + 2 \leq abx_n \leq b\alpha,$$

which simplifies to

$$\alpha - a + \lceil 2/b \rceil \leq ax_n \leq \alpha.$$

The gray shadow constraint in this case becomes

$$\mathbf{G}_a \equiv \bigvee_{i=0}^{a-\lceil 2/b \rceil} ax_n = \alpha - i.$$

Either one of $\mathbf{G}_a$ or $\mathbf{G}_b$ is sufficient to guarantee integrality of x_n and the completeness of the procedure. Therefore, we can pick the one which requires fewer cases to consider. Namely, pick $\mathbf{G}_a$ if $a < b$, and $\mathbf{G}_b$ otherwise.

This is, obviously, the most expensive step of the algorithm, since it involves a lot of backtracking, but according to [Pug91], the dark shadow constraint almost always suffices in practice, and the gray shadow is often empty. Therefore, as a practical heuristic, the dark shadow constraint $\mathbf{D}$ is always tried first, and only if it fails, then a solution is searched for in the gray shadow $\mathbf{G}$.

3.4 Online Version of Fourier-Motzkin for Reals

In CVC, decision procedures are most effective when they are *online*, that is, the constraints are not given all at once but are fed to the decision procedure one at a time, and for each constraint the algorithm performs some relatively small amount of work to take that constraint into account and derive new constraints that follow from it.

In order to understand the reasons for being online and to clarify the important interface features that the decision procedure relies on, we give a brief introduction to the CVC

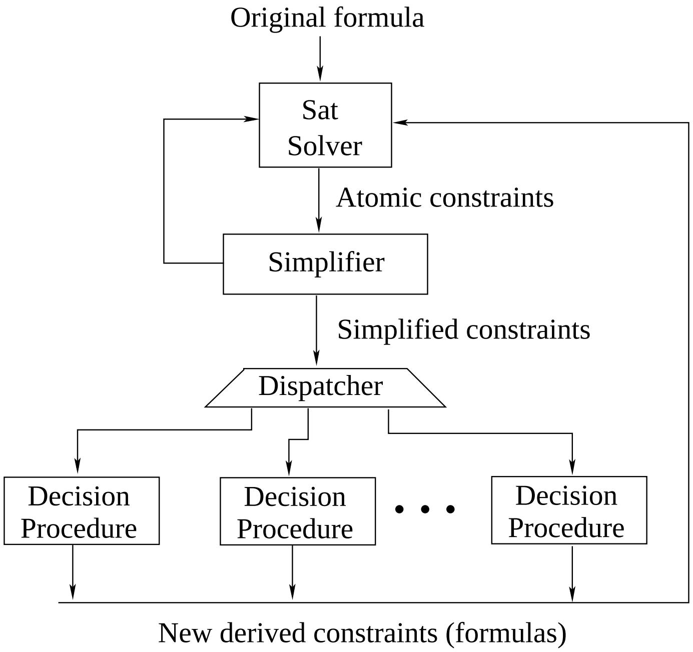

Figure 3.1: Flow of constraints in CVC

framework.

3.4.1 Brief Introduction to CVC Framework

The goal of this subsection is to provide just enough information about the interface and underlying structure of the CVC framework to understand the requirements for the online version of the decision procedure for mixed integer linear arithmetic. Therefore, some of the features are greatly simplified or omitted. For more details on CVC framework the reader is referred to [BDS00, BDS02b, BDS02a].

At a very high level, CVC can be viewed as a SAT solver for the Boolean skeleton of the quantifier-free first-order formulas (Figure 3.1). The SAT solver treats the atomic constraints from different theories as Boolean variables. It solves the satisfiability problem by *splitting cases* on each variable; that is, picking a variable, assigning it values true and false (making a decision), and solving the rest of the formula for each case recursively. If it finds a satisfying assignment to variables, then the original formula is satisfiable. When a particular set of decisions results in a contradiction, the SAT solver backtracks and tries a different decision. If in all branches it derives a contradiction, then the formula is unsatisfiable.

Since the variables represent constraints from various theories, each time the SAT solver makes a decision, a new constraint is produced, which is simplified by the simplifier and dispatched to the appropriate decision procedure. The decision procedure receives a constraint from the simplifier, derives new constraints from the current and previously seen constraints, and then returns new constraints back to the framework.

These decisions are dynamic, which requires decision procedures to be able to receive a new constraint and process it efficiently, deriving a contradiction as soon as possible to cut off the search early. This explains the online requirement. In some cases, however, a simplified constraint may be returned directly to the SAT solver without going through a decision procedure.

Note that the new constraints produced by the individual decision procedures may contain arbitrary Boolean combinations of atomic formulas, but the decision procedures always receives atomic constraints (equalities and theory-specific predicates over terms). In other words, the decision procedure can assume it always solves the satisfiability problem for a conjunction of atomic constraints.

To boost the efficiency of the SAT solver, the *intelligent backtracking* technique is utilized in conjunction with *conflict clauses* [SS96, MSS99]. To construct a conflict clause, one needs to identify a (preferably small) set of decisions made by the SAT solver that lead to the contradiction. One possible way of identifying such decisions is to extract them from the *proof* that the decision procedure constructs when it derives false. This explains the need for proof production in the decision procedures in CVC. Also, these proofs can be checked by an external proof checker [Stu02] to increase the confidence in the results produced by CVC.

As mentioned earlier, each decision procedure assumes that it solves the satisfiability of a conjunction of constraints, and when new constraints arrive, they are added to the conjunction. However, when the SAT solver backtracks, some of the constraints are effectively removed. Therefore, if a decision procedure stores some information about previously received constraints, it must be able to roll back to the appropriate state when the SAT solver backtracks. In other words, all the data structures which persist across calls to the decision procedure must be *backtrackable*. In the description of the algorithm below it is always assumed that such backtracking mechanism is properly implemented and is completely transparent to the decision procedure.

3.4.2 Online Fourier-Motzkin Elimination for Reals

In the following, it is always assumed that the constraints dispatched to the decision procedure are *atomic* (no Boolean connectives) and *simplified*. The simplification step consists of theory-specific rewrites such as normalization of arithmetic constraints and elimination of equalities, so that only normalized inequality constraints reach the decision procedure.

Hence, the description of the algorithm consists of two parts: the simplifier, which is a set of equivalent transformations, and the decision procedure itself. The latter is presented as a function that takes a simplified atomic constraint and returns (possibly a Boolean combination of) new constraints back to the framework.

In the online algorithm below a total fixed ordering $\lessdot$ on all variables is assumed, which in turn defines the order in which the variables are projected from inequalities (This ordering holds for integer variables as well). The ordering $\lessdot$ chosen by the online algorithm is the same as in the offline case. In particular, x is said to be the *maximal* variable from a set of variables when it is the maximum in this set with respect to $\lessdot$.

In this section, only equalities and inequalities over real-valued variables are considered. Handling the constraints with $\mathsf{int}(t)$ predicate and integer variables will be described in Section 3.5.

Simplification Step

Each equality constraint $t_1 = t_2$ is first rewritten as $t_1 - t_2 = 0$ and simplified by grouping like terms. If the resulting equality contains no variables (meaning $t_1 - t_2$ simplifies to a numerical constant), then it is checked for satisfiability, and the result is reported directly to the top-level SAT solver. Otherwise, it is rewritten in the form $x = \alpha$ for some variable x, and then x is replaced by α everywhere in the system, completely eliminating the variable x.

Similarly, an inequality $t_1 < t_2$ is rewritten as $t_1 - t_2 < 0$ and simplified. If the left-hand side simplifies to a constant, the inequality is evaluated to true or false and submitted back to the solver. Otherwise, it is rewritten as $\beta < x$ or $x < \alpha$ for the maximum variable x in $t_1 - t_2$ w.r.t. $\lessdot$, and forwarded to the decision procedure.

The Decision Procedure

Due to the simplification step above, the decision procedure receives only inequalities of the form $\beta < x$ or $x < \alpha$, where x is the maximal variable among all the variables occurring in α and β with respect to the ordering $\lessdot$ (Note that α and β may not contain x). The variable x in such inequalities is referred to as *isolated*.

The decision procedure maintains a backtrackable database $\mathrm{DB}_{<}$ of inequalities indexed by the isolated variable. Whenever a new inequality $x < \alpha$ arrives, this database is searched for the opposite inequalities $\beta < x$ and, for each such inequality, the new shadow constraint $\beta < \alpha$ is constructed and asserted back to the framework. The received constraint $x < \alpha$ is then added to the database. The inequalities of the form $\beta < x$ are handled similarly. The newly generated constraint $\beta < \alpha$ is eventually simplified and submitted back to the decision procedure with a smaller variable (with respect to $\lessdot$) isolated, and this process repeats until no variables remain in the constraint.

The soundness and completeness of the algorithm above follows from the fact that every step in the algorithm maintains equisatisfiability with the original input. Only the question of termination remains.

Termination can be informally argued as follows. The ordering $\lessdot$ on the variables guarantees that all the intermediate constraints (it is guaranteed that there can only be finitely many) that would be derived from an input set of constraints C by the offline version of the procedure are eventually constructed and processed by the online version, even though the constraints from C may be fed to the online version in any order. This fact implies that if the offline version terminates and declares that C is unsatisfiable, the online version also will eventually derive *false* and declare that C is unsatisfiable.

However, the online version may produce more constraints than the offline one. The reason for this is that when variables from equalities are eliminated and substituted into C, the SAT solver may decide to resubmit the modified constraints. But, all these *extra* constraints in the online version are actually redundant and will be normalized to some constraint that is also generated by the offline algorithm. This implies that if the offline version terminates and declares C is satisfiable then the online version will also eventually terminate and declare the same. Since the offline algorithm is known to terminate on all inputs, it follows that the online version also terminates on all inputs.

3.5 Online Fourier-Motzkin Elimination for Integers

The online version of the decision procedure for integers cannot have as direct a correspondence to the offline version as for the reals, since the order of the projection of variables depends on the integrality constraints, $\mathsf{int}(x)$ and $\neg\mathsf{int}(x)$, and the variable may become known to be integer only after it has already been projected or eliminated. A naïve solution would be to backtrack and redo the projection and elimination steps. This could be a very costly operation.

Fortunately, there is a simple and elegant solution to this problem. Whenever a constraint $\mathsf{int}(x)$ arrives, a new constraint $x - \sigma = 0$ is added to the system, where σ is a new integer variable, and the fact $\mathsf{int}(\sigma)$ is recorded into a local database $\mathrm{DB}_{\mathrm{int}}$ indexed by σ. The resulting system is equisatisfiable with the original one (which includes $\mathsf{int}(x)$), but the variable x remains real-valued in the new system. Therefore, the projections and eliminations of x do not have to be redone. At the same time, the integrality of x is enforced by the integrality of σ.

In addition, for any integer constraint $\mathsf{int}(t)$ in $\mathrm{DB}_{\mathrm{int}}$, whenever the term t is rewritten to t' (because of some variable elimination), and t' simplifies to a constant term c, one must check that c is indeed an integer and assert unsatisfiability if it is not. Just like the algorithm for only real variables, the online algorithm for deciding mixed-integer linear constraints consists of two parts: simplification and decision procedure.

Simplification step. This version of the simplifier performs the same transformations as the one for real variables (Section 3.4.2). An equality constraint is first rewritten as $\gamma = 0$ and γ is checked for being a constant. If it is, the constraint is immediately checked for satisfiability and the result is returned directly to the SAT solver. Otherwise, if γ contains real-valued variables, then one such variable x is isolated and eliminated. If γ contains only integer variables, then the iterative equality elimination algorithm is performed, as described in Section 3.3.1. At the end of this process the equality $\gamma = 0$ is rewritten into an equisatisfiable system of equations

$$\begin{cases} x_1 & = & \beta_1 \\ \vdots & & \vdots \\ x_k & = & \beta_k, \end{cases}$$

where each equation $x_i = \beta_i$ corresponds to the equation (3.5) in Section 3.3.1 from each iteration of the algorithm. All the variables x_i are then eliminated by replacing them with their right-hand sides. Thus, equations are handled in the simplification step and never submitted to the actual decision procedure.

Inequalities are also transformed and simplified into the form $\gamma < 0$, then evaluated if γ is a constant. If γ contains variables, the inequality is rewritten to the form $\beta < ax$ or $ax < \alpha$ for some positive integer coefficient a, where the variable x is the maximum with respect to $\lessdot$. The new inequality is then forwarded to the decision procedure. Similar to the offline version of the algorithm, it is important to project real variables first. Therefore, we define $\lessdot$ such that real-valued variables are always higher in the ordering than the integer ones.

In the constraints of the form $\mathsf{int}(t)$ and $\neg\mathsf{int}(t)$ only the term t is simplified by combining like terms, and otherwise these constraints are passed to the decision procedure unmodified.

3.5.1 The Decision Procedure

First, observe that, due to the simplification step, only inequalities of the form $\beta < bx$ and $ax < \alpha$, where the coefficients b and a are positive integers, and integer constraints $\mathsf{int}(t)$ and $\neg\mathsf{int}(t)$ are submitted to the decision procedure. Notice that inequality constraints always have the maximal variable isolated w.r.t. $\lessdot$. These inequalities are stored in the local database $\mathrm{DB}_<$. Additionally, whenever a term t in any constraint $\mathsf{int}(t) \in \mathrm{DB}_{\mathrm{int}}$ is rewritten to t' by the simplifier, $\mathsf{int}(t)$ is automatically replaced by $\mathsf{int}(t')$ in $\mathrm{DB}_{\mathrm{int}}$. Both local databases are also backtrackable.

The decision procedure receives a constraint C from the simplifier and returns, or *asserts*, new constraints back to the framework. We describe it as a case-by-case analysis of the constraint C.

1. $C \equiv \mathsf{int}(t)$:

   (a) If t is a constant, then evaluate $\mathsf{int}(t)$, assert the result to the framework, and return. If t is not a constant, go to step 1b.

   (b) If $t \notin \mathrm{DB}_{\mathrm{int}}$, then create a new integer variable z, add t and z into $\mathrm{DB}_{\mathrm{int}}$, assert the new facts:

   $$\mathsf{int}(z) \quad \text{and} \quad t - z = 0.$$

   Otherwise, if $\mathrm{t} \in \mathrm{DB}_{\mathrm{int}}$, then ignore C and return.

2. $C \equiv \neg\mathsf{int}(t)$:

   Introduce a new integer variable z, add z to $\mathrm{DB}_{\mathrm{int}}$ and assert the new constraints:

   $$\mathsf{int}(z) \quad \text{and} \quad z < t < z + 1.$$

3. $C \equiv a \cdot x \leq \alpha$ (or $C \equiv \beta \leq b \cdot x$, or the strict variants of inequalities):

   (a) Find all inequalities of the form $\beta \leq b \cdot x$ in the database $\mathrm{DB}_{<}$, and for each such inequality perform the following steps:

      i. Locate inequalities of the form $a \cdot x \leq \alpha$ (recall that all inequalities are simplified such that the coefficients a and b are positive integers), and compute simplified terms $a \cdot \beta$ and $b \cdot \alpha$. If no inequalities of the form $a \cdot x \leq \alpha$ exist, then go to step (b)

      ii. If $a \cdot \beta = b \cdot \alpha$, then generate and assert a new equality $b \cdot x = \beta$, and return; otherwise generate and assert the real shadow constraint $R \equiv a \cdot \beta \leq b \cdot \alpha$ and proceed to the next step

iii. If $x \in \mathrm{DB}_{\text{int}}$ (in which case all the variables in α and β must also be in DB_{int}), then generate the integrality constraint $\mathbf{D} \vee \mathbf{G}$ (dark and gray shadows), where $\mathbf{D}$ and $\mathbf{G}$ are defined as in Section 3.3.2:

$$\mathbf{D} \equiv b \cdot \alpha - a \cdot \beta \geq ab - 1$$

and the gray shadow constraint G is

$$\mathbf{G} = \bigvee_{i=0}^{a-\lceil 2/b \rceil} a \cdot x = \alpha - i, \quad \textbf{if } a < b,$$

or

$$\mathbf{G} = \bigvee_{i=0}^{b-\lceil 2/a \rceil} b \cdot x = \beta + i, \quad \textbf{if } a \geq b.$$

Following the heuristic of Pugh [Pug91], the top-level SAT solver should first search for a solution in $\mathbf{D}$ before trying $\mathbf{G}$.

(b) Add the received constraint C to $\mathrm{DB}_{<}$ and return.

It is not hard to see that each step of the algorithm above corresponds very closely to the similar steps in the offline version of the algorithm. The soundness and completeness of the procedure follow from the fact that the set of constraints asserted by the decision procedure at each step is always equisatisfiable with the given constraint C.

Shared Terms Requirements. The combination framework of CVC [Bar03] requires additional properties from the decision procedure in order to guarantee completeness of the entire tool. Namely, all the equalities that can be derived between *shared terms* must be derived and asserted by the decision procedure. A term is *shared* if it belongs to a theory T_1 and occurs in a term or predicate from a theory T_2. For example, in $\mathrm{cons}(4 + 5 * x, \mathrm{lst})$ the term $4 + 5 * x$ is shared, since it belongs to arithmetic theory, but occurs in a term from the theory of lists. The framework notifies the decision procedure every time a new shared term appears.

The decision procedure already propagates all of the equalities that are derived from other equalities. Therefore, only the equalities that can be derived from inequalities need to

#experiments in each suite	#formulas / #vars in each experiment	CVC completed	Omega completed	avg. slow-down factor
5996	1-4/1-5	5990 (99.9%)	5568 (92%)	13.4
395	1-10/1-20	387 (98%)	322 (81%)	192
65	10-50/10-50	61 (95%)	8 (12%)	7.8

Table 3.1: Experimental comparisons of CVC vs. Omega

be considered. In reals, an equality is derived from an inequality only when a shared term t is “squeezed” by two inequalities to a single value:

$$\alpha \leq t \leq \alpha \Longrightarrow t = \alpha. \tag{3.12}$$

In integers, a more general finite interval constraint produces equalities:

$$\alpha \leq t \leq \alpha + n \Longrightarrow \bigvee_{i=0}^{n} t = \alpha + i. \tag{3.13}$$

In reals, it is sufficient to track equalities like (3.12) only when t is a variable. This follows from the fact that $\alpha \leq t \leq \alpha$ will always reduce to $\beta \leq ax \leq \beta$ for some x from t or α. Similarly, tracking finite intervals for all integer variables is also enough to propagate all the necessary equalities. However, this may be inefficient, and a more efficient procedure is desirable.

Whenever a new shared term t is reported, we check whether t is a variable (that is, not a non-trivial arithmetic term like $a + b$ or $a \cdot x$). If it is a variable, we record it into our local database of shared variables Λ. If not, we assert a new formula $t = x$ to the framework, where x is a new variable. Then t is replaced everywhere by x, and eventually x is reported as a new shared term, which this time goes into Λ. Integer equalities from finite intervals are only propagated for variables in Λ. This also means that every time a variable x is added to Λ, this variable has to be checked for any existing inequalities which define a finite interval.

Alternatively, Λ can record all the variables that appear in the shared terms, and then all the inequalities of the form of (3.13) which contain at least one variable from Λ in t or

α need to be transformed into a disjunction of equalities.

Experimental Results

Table 3.1 shows the experimental comparison we did between the CVC Tool and the Omega Tool. The experiments used suites of randomly generated examples. As can be seen from Table 3.1, CVC is much more stable than the Omega tool. However, CVC is generally slower than Omega approximately by a factor of 10. By profiling the code, we concluded that most of the slowdown is due to the overhead of the CVC framework and the use of arbitrary precision arithmetic. However, it is only a constant factor slowdown. Since this implementation is not yet tuned for efficiency, there are a few exceptional cases when CVC performs much worse than Omega, resulting in a large slow-down factor in the second line of Table 3.1.

3.6 Proof Production

When the algorithm in Section 3.5 reports that the system of constraints is unsatisfiable, it produces a proof of this fact which can be verified independently by an external proof checker. This increases our confidence in the soundness of the implementation.

Additionally, proof production mechanism allows CVC framework to extract logical dependencies that drive the backtracking mechanism of the built-in SAT solver (see Section 3.4.1). The details are out of the scope of this chapter, but intuitively, if the proof of unsatisfiability depends only on a small subset of decisions made by the SAT solver, then the SAT solver memorizes this combination of decisions and avoids it in the future. This can dramatically reduce the size of the decision tree.

3.6.1 Natural Deduction

Since the algorithm in Section 3.5 consists of the set of relatively simple transformations which take existing constraints and assert new ones, it is natural to construct the proofs for the new assertions, taking the given facts as assumptions.

The proof production interface, therefore, is implemented as follows. Whenever our algorithm receives a new constraint, it also receives a proof that this constraint holds (or a *proof of this constraint*). Whenever the algorithm asserts a new constraint, it also builds and supplies a proof of this new constraint, which may include the available proofs of the previously submitted constraints. In other words, the algorithm maintains the invariant, that every constraint appearing in the algorithm has an associated proof with it.

The proofs are represented as derivations in natural deduction extended with arithmetic and with specialized derived or admissible rules. The specialized rules can then be proven sound externally, either by manual inspection, or with the help of automated theorem provers. Ideally, every step of the algorithm should correspond to one such proof rule, to minimize the proof size. However, some steps of the algorithm have conditional branches or sub steps, and it is often convenient to break the proof for those into smaller subproofs, resulting in additional but simpler proof rules.

Definitions

A *judgment* (or a *statement*) in our version of natural deduction is just a formula in the quantifier-free first-order logic with arithmetic. Most often, the formula has the following forms:

1. Equivalence of two arithmetic constraints: $A \equiv B$;
2. Equality or inequality of two arithmetic terms: $t_1 = t_2$ or $t_1 < t_2$;
3. Boolean combination of the above.

The *inference rules*, or *proof rules*, are normally of the form

$$\frac{P_1 \quad \cdots P_n \quad S_1 \quad \cdots S_m}{C}\,\mathrm{rule\,name}$$

where judgments P_i are the *premises* (they are assumed to have proofs), S_i are side conditions (the rule is applicable only if all S_i are true), and the judgment C is the *conclusion* of the rule. The semantics of a proof rule is that if all P_i are valid, then C is also valid, provided that all the side conditions S_j are true. Typical rules of this form are “and introduction” and “modus ponens:”

$$\frac{A \quad B}{A \wedge B} \wedge I \qquad \frac{A \quad A \to B}{B}\mathrm{MP}.$$

The other type of proof rules introduces *assumptions*. For instance, in order to show that an implication $A \to B$ is valid, one needs to show that, given a proof of A, it is possible to construct a proof of B. But A does not have to be valid in order for the implication to be valid. Therefore, the “implication introduction” rule has the form:

$$\begin{array}{c} \overline{}\; u \\ A \\ \vdots \\ B \\ \overline{}\; \to I^u \\ A \to B \end{array}$$

Here u is a *parametric derivation*, an assumed proof of A which can only be used above the line of this rule in the proof of B. The dots between A and B must be filled in with an actual proof of B.

A *derivation* of a formula A is a tree, whose nodes are formulas, A is the root, and the children of each node F are premises in some rule application in which F is a conclusion. For example, a proof of a formula $A \wedge (A \to B) \to B$ is the following:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \land (A \to B)}\,u}{A}\,\land E_1 \qquad \dfrac{\dfrac{}{A \land (A \to B)}\,u}{A \to B}\,\land E_2}{B}\,\text{MP}}{A \land (A \to B) \to B}\,\to I^u$$

Notice, that the assumption u is used twice in the proof of B.

Proof Rules for Equivalent Transformations

It is often necessary to rewrite a term to a different but an equivalent term. For instance, our algorithm assumes that all the linear arithmetic terms are reduced by the framework to a canonical form: $\sum_{i=1}^{n} a_i \cdot x_i$, even though the new asserted facts may include non-canonical forms.

To justify such transformations in the proofs, specialized proof rules are introduced of the form

$$\frac{S_1 \cdots S_n}{t_1 = t_2},$$

where S_i are side conditions, and t_i are arithmetic terms. The intended meaning of such a rule is that the term t_1 can be equivalently rewritten to t_2.

Similarly, arithmetic constraints are often rewritten in a similar way:

$$\frac{S_1 \cdots S_n}{c_1 \equiv c_2}.$$

This way constraints can be modified without having to prove them. Sequences of such transformations are glued together with the *transitivity rule*:

$$\frac{t_1 = t_2 \quad t_2 = t_3}{t_1 = t_3}\mathrm{trans},$$

and these equalities can be used to derive new equalities using *substitution*:

$$\frac{t_1 = t'_1 \quad \cdots \quad t_n = t'_n}{f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)}\mathrm{subst}.$$

Proof Rules for The Algorithm

We divide the proof rules for our decision procedure into *transformations* and *inferences*. Generally, all the transformations in the simplifier (canonization of terms, elimination of equalities, and variable isolation) are equivalent transformations. In this chapter, we do not consider canonization of terms and refer the reader to [Stu02].

Fourier-Motzkin projection of a variable and handling the $\mathsf{int}()$ predicate involves deriving new constraints from the existing ones. Therefore, their proof rules have a form of *inference*, that is, a general rule which derives a free-form constraint, not necessarily an equality, from already proven premises.

3.6.2 Proof Rules for Equivalent Transformations

Normalization

Multiplying an (in)equality by a (positive) number preserves the constraint:

$$\frac{f \in \mathcal{R},\ f > 0}{t_1 < t_2 \quad \equiv \quad f \cdot t_1 < f \cdot t_2}\mathrm{norm}_< \qquad \frac{f \in \mathcal{R},\ f > 0}{t_1 \le t_2 \quad \equiv \quad f \cdot t_1 \le f \cdot t_2}\mathrm{norm}_\le$$

$$\frac{f \in \mathcal{R}}{t_1 = t_2 \quad \equiv \quad f \cdot t_1 = f \cdot t_2}\mathrm{norm}_=$$

Variable Isolation

Given an (in)equality, pick a variable to isolate and transform the constraint in such a way that the variable with some positive coefficient is solely on one side, and the rest of the term is on the other. For equalities, the isolated variable must always be on the left-hand side. For inequalities, it depends on the sign of the coefficient: if it is positive, the variable stays on the left-hand side, and all the other terms are moved to the right-hand side; otherwise the variable is isolated on the right-hand side.

The four rules below implement the four cases of variable elimination: for equality and inequality, and for positive and negative coefficients:

$$\frac{a_i > 0}{\begin{array}{ll} & c + a_1 \cdot x_1 + \cdots + a_i \cdot x_i + \cdots + a_n \cdot x_n = 0 \\ \equiv & a_i \cdot x_i = -(c + a_1 \cdot x_1 + \cdots + a_{i-1} \cdot x_{i-1} + a_{i+1} \cdot x_{i+1} + \cdots + a_n \cdot x_n) \end{array}}\mathrm{VI}^+_=$$

$$\frac{a_i < 0}{\begin{array}{ll} & c + a_1 \cdot x_1 + \cdots + a_i \cdot x_i + \cdots + a_n \cdot x_n = 0 \\ \equiv & -a_i \cdot x_i = c + a_1 \cdot x_1 + \cdots + a_{i-1} \cdot x_{i-1} + a_{i+1} \cdot x_{i+1} + \cdots + a_n \cdot x_n \end{array}}\mathrm{VI}^-_=$$

$$\frac{a_i > 0}{\begin{array}{ll} & c + a_1 \cdot x_1 + \cdots + a_i \cdot x_i + \cdots + a_n \cdot x_n < 0 \\ \equiv & a_i \cdot x_i < -(c + a_1 \cdot x_1 + \cdots + a_{i-1} \cdot x_{i-1} + a_{i+1} \cdot x_{i+1} + \cdots + a_n \cdot x_n) \end{array}}\mathrm{VI}^+_<$$

$$\frac{a_i < 0}{\begin{array}{ll} & c + a_1 \cdot x_1 + \cdots + a_i \cdot x_i + \cdots + a_n \cdot x_n < 0 \\ \equiv & c + a_1 \cdot x_1 + \cdots + a_{i-1} \cdot x_{i-1} + a_{i+1} \cdot x_{i+1} + \cdots + a_n \cdot x_n < -a_i \cdot x_i \end{array}}\mathrm{VI}^-_<$$

We omit the similar two rules $\mathrm{VI}^+_\leq$ and $\mathrm{VI}^-_\leq$.

Elimination of Equalities

Most of the rules in this subsection implement the low-level details of the variable elimination in equalities. They are divided into two groups: elimination of real variables and elimination of integer variables.

Elimination of Real Variable

The first two rules are axioms of multiplication by 1:

$$\frac{}{1 \cdot t = t}\text{mult drop one} \qquad \frac{}{t = 1 \cdot t}\text{mult one.}$$

When the isolated variable x is real, its coefficient is normalized to 1:

$$\frac{a \neq 0}{a \cdot x = \alpha \quad \equiv \quad x = \alpha/a}\text{Eq Elim.}$$

After this transformation, when the new fact $x = \alpha/a$ propagates to the global database of facts, the canonical form of x becomes α/a. This fact is recorded into the global database, and all the existing constraints containing x will be automatically updated with the new value of x and re-canonized. This effectively eliminates x from the system of constraints, both current and future ones.

Elimination of Integer Variable

When all the variables in the constraint are integers, including the isolated variable x, the elimination of the variable has to go through a number of iterative transformations, as described in Section 3.3.1. One step of each iteration is formalized as the following proof rule:

$$\frac{\begin{array}{rcl} (C\,\mathbf{mod}\,m + \sum_{i=1}^{n}(a_i\,\mathbf{mod}\,m)\cdot x_i + x)/m & = & t \\ -(C\,\mathbf{mod}\,m + \sum_{i=1}^{n}(a_i\,\mathbf{mod}\,m)\cdot x_i - m\cdot\sigma(t)) & = & t_2 \\ \mathbf{f}(C) + \sum_{i=1}^{n}\mathbf{f}(a_i)\cdot x_i - a\cdot\sigma(t) & = & t_3 \\ \multicolumn{3}{c}{\mathsf{int}(x)\quad \mathsf{int}(x_i)\quad a:\mathcal{N}\quad C:\mathcal{N}\quad a_i:\mathcal{N}\text{ for } i=1..n\quad a\geq 2} \end{array}}{a\cdot x = C + \sum_{i=1}^{n} a_i\cdot x_i \quad\equiv\quad x = t_2 \wedge t_3 = 0}\,\text{Eq Elim int}$$

where $m = a + 1$, and

$$i\,\mathbf{mod}\,m = i - m\left\lfloor\frac{i}{m}+\frac{1}{2}\right\rfloor,$$

$$\mathbf{f}(i) = \left\lfloor\frac{i}{m}+\frac{1}{2}\right\rfloor + i\,\mathbf{mod}\,m.$$

The first three premises of this rule can be thought of as definitions of t, t_2, and t_3 for the use in the rest of the formulas. In the actual implementation, however, t, t_2, and t_3 are the canonized versions of their left-hand sides. Thus, for example, the new "variable" $\sigma(t)$ is constructed from the canonical term t, and not the left-hand side expression in the first premise.

Note, that instead of introducing a new variable σ, we use a single *function* $\sigma(t)$, which is axiomatized as $\sigma(t) = \lfloor t\rfloor$ in Section 3.6.3. This way we avoid the need to generate new variable names (which is not directly expressible in natural deduction) and guarantee that exactly the same "variable" will be introduced for exactly the same equality, even if it is processed several times.

System of Equalities in Solved Form

Note, that the rule above transforms an equality into a conjunction of two equalities; one is in a *solved form* (the variable on the left-hand side has coefficient 1), and the other is not. The equality elimination algorithm guarantees that eventually the second equality can be transformed into solved form directly by isolating a variable, and at that moment the original equality will correspond to a conjunction of n equalities in solved form:

$$\begin{array}{rcl} x_1 & = & t_1 \\ \wedge x_2 & = & t_2 \\ \vdots & & \vdots \\ \wedge x_n & = & t_n. \end{array}$$

For technical reasons (since we implement the algorithm as a Shostak decision procedure [Sho84, RS01]), such conjunction of equalities needs to be transformed into a *system of equalities in solved form*. The additional requirement is that no isolated variable on the left-hand side can appear on the right-hand side of any equality. Since no variable x_j appears in t_i for $j \leq i$, this can be achieved by substituting x_i's in all the previous right-hand sides t_j for $j < i$, starting from x_n.

If done directly, the first equality $x_1 = t_1$ will be transformed $n - 1$ times. This procedure can be optimized to transform each t_i only once by substituting all $x_{i+1}, \ldots, x_n$ *simultaneously* for the new (already transformed) right-hand sides $t'_{i+1}, \ldots, t'_n$ respectively, obtaining a new i-th right-hand side t'_i.

The following proof rule is designed for one step of such substitution:

$$\frac{}{x = t \wedge (x_1 = t_1 \wedge \cdots \wedge x_n = t_n) \equiv x = t' \wedge (x_1 = t_1 \wedge \cdots \wedge x_n = t_n)} \equiv_{\wedge}$$

where $t' = t[t_1/x_1, \ldots, t_n/x_n]$ (the result of the simultaneous substitution of t_i for x_i in t). This rule is derivable from the existing propositional rules and the substitution rules already implemented in CVC, and we provide it here only to "cut the corner" and simplify the proof. The derivation is quite straightforward but rather tedious, and we do not provide it in this chapter.

3.6.3 Elimination of Inequalities: Inference Rules

The proof rules in this section derive new constraints from already existing ones. This type of rules is needed for the actual Fourier-Motzkin elimination, or *projection* of variables from inequalities, and for handling the $\mathsf{int}()$ predicate.

Handling the $\mathsf{int}()$ Predicate

When a non-integer constant is constrained to be an integer, such constraint is unsatisfiable, and the following rule applies:

$$\frac{\mathsf{int}(n:\mathcal{R}) \quad n \notin \mathcal{N}}{\mathsf{false}}\mathrm{Const\_int}.$$

The rules below are axiomatization of $\sigma(t)$ as $\lfloor t \rfloor$. The last two rules are used to handle the negation of the $\mathsf{int}()$ predicate.

$$\frac{}{\mathsf{int}(\sigma(t))}\mathrm{int}_{\sigma} \qquad \frac{\mathsf{int}(t)}{t-\sigma(t)=0}\mathrm{Term\_int}$$

$$\frac{\mathsf{int}(t)\equiv\mathsf{false}}{\sigma(t)<t}\neg\mathrm{int}_{<t} \qquad \frac{\mathsf{int}(t)\equiv\mathsf{false}}{t<\sigma(t)+1}\neg\mathrm{int}_{\mathrm{t}<}$$

Real Shadow

Deriving real shadow from two opposing constraints makes a simple and obvious set of proof rules:

$$\frac{\alpha\le t \quad t\le\alpha}{t=\alpha}\mathrm{Real\ Shadow}_0$$

$$\frac{\beta<t \quad t<\alpha}{\beta<\alpha}\mathrm{Real\ Shadow}_1 \qquad \frac{\beta\le t \quad t\le\alpha}{\beta\le\alpha}\mathrm{Real\ Shadow}_2$$

$$\frac{\beta\le t \quad t<\alpha}{\beta<\alpha}\mathrm{Real\ Shadow}_3 \qquad \frac{\beta<t \quad t\le\alpha}{\beta<\alpha}\mathrm{Real\ Shadow}_4$$

Dark and Gray Shadows

Instead of asserting the disjunction of dark and gray shadow in the fully expanded form as given in the algorithm, we "hide" the constraints under special predicates $\mathbf{D}(t_1,\,t_2)$ and $\mathbf{G}(t_1,\,t_2,\,n)$ and later expand them "lazily." The intended semantics for these predicates is the following:

$$\begin{array}{rcl}
\mathbf{D}(t_1,\, t_2) & \equiv & t_1 \leq t_2 \\
\mathbf{G}(t_1,\, t_2,\, 0) & \equiv & t_1 = t_2 \\
\mathbf{G}(t_1,\, t_2,\, i) & \equiv & t_1 = t_2 + i \vee \mathbf{G}(t_1,\, t_2,\, i-1) \\
\mathbf{G}(t_1,\, t_2,\, -i) & \equiv & t_1 = t_2 - i \vee \mathbf{G}(t_1,\, t_2,\, -i+1),
\end{array}$$

where $i > 0$. Thus, the type of the gray shadow (for $a < b$ and $a \geq b$) is encoded into the sign of the bound n of $\mathbf{G}(t_1,\, t_2,\, n)$, and the large disjunction is expanded one disjunct at a time.

The following rules infer the new constraint $\mathbf{D} \vee \mathbf{G}$ in our special encoding. They also include an optimization for the case when either a or b is 1, so the gray shadow is not necessary. The seeming complexity of the rules comes from the fact that they are only sound when all the variables and coefficients are integer. Such premises and side conditions are the primary reason for the complexity. However, the key premises are always the two opposing inequalities, $\beta \leq b \cdot x$ and $a \cdot x \leq \alpha$, as in the $\mathrm{Real\ Shadow}$ rule.

$$\frac{\begin{array}{l}\beta \leq b \cdot x \quad a \cdot x \leq \alpha \quad \mathsf{int}(\alpha) \quad \mathsf{int}(\beta) \quad \mathsf{int}(x) \\ a, b \in \mathcal{N} \quad 1 \leq a \quad a \leq b \quad 2 \leq b\end{array}}{\mathbf{D}(ab-1,\, b\alpha - a\beta) \vee \mathbf{G}(a \cdot x,\, \alpha,\, -a + \lceil 2/b \rceil)}\,\mathbf{D}\text{ or }\mathbf{G}_{1 \leq a \leq b}$$

$$\frac{\begin{array}{l}\beta \leq b \cdot x \quad a \cdot x \leq \alpha \quad \mathsf{int}(\alpha) \quad \mathsf{int}(\beta) \quad \mathsf{int}(x) \\ a, b \in \mathcal{N} \quad 1 \leq b \quad b \leq a \quad 2 \leq a\end{array}}{\mathbf{D}(ab-1,\, b\alpha - a\beta) \vee \mathbf{G}(b \cdot x,\, \beta,\, b - \lceil 2/a \rceil)}\,\mathbf{D}\text{ or }\mathbf{G}_{1 \leq b \leq a}$$

$$\frac{\beta \leq x \quad x \leq \alpha \quad \mathsf{int}(\alpha) \quad \mathsf{int}(\beta) \quad \mathsf{int}(x)}{\mathbf{D}(0,\, \alpha - \beta)}\,\mathbf{D}\text{ or }\mathbf{G}_{a=b=1}$$

Expanding Dark and Gray Shadow

When the dark shadow constraint arrives as a new fact to our decision procedure, it is time to expose its interpretation:

$$\frac{\mathbf{D}(t_1,\, t_2)}{t_1 \leq t_2}\text{Expand } \mathbf{D}.$$

In principle, the built-in SAT solver may also decide to assert the negation of the dark shadow in some other decision branch, but then it will have to assert the gray shadow. Therefore, there is no rule for the negation of $\mathbf{D}(t_1,\, t_2)$, and it is simply ignored.

Similarly, when $\mathbf{G}(t_1,\, t_2,\, n)$ arrives as a new fact, we would like to expose its interpretation to the framework (and ignore its negation, similar to the dark shadow). However, when n is large, the resulting formula is a big disjunction, and asserting it all at once may cause too many unnecessary decision points for the SAT solver. Therefore, the disjuncts are expanded one at a time.

Non-constant RHS: The following rules expand $\mathbf{G}(t_1,\, t_2,\, n)$ exactly as prescribed by the intended semantics:

$$\frac{\mathbf{G}(t_1,\, t_2,\, i : \mathcal{N}) \quad i \geq 1}{\mathbf{G}(t_1,\, t_2,\, i-1) \vee t_1 = t_2 + i}\text{Expand } \mathbf{G}_{\geq 1}$$

$$\frac{\mathbf{G}(t_1,\, t_2,\, i : \mathcal{N}) \quad i \leq -1}{\mathbf{G}(t_1,\, t_2,\, i+1) \vee t_1 = t_2 + i}\text{Expand } \mathbf{G}_{\leq -1}$$

$$\frac{\mathbf{G}(t_1,\, t_2,\, 0)}{t_1 = t_2}\text{Expand } \mathbf{G}_{=0}.$$

When t_2 is a constant term c, the expansion of $\mathbf{G}(t_1,\, c,\, n)$ creates disjuncts of the form $a \cdot x = c'$ for some constants c' from a finite range (assuming that $n > 0$). Such disjunct is satisfiable only if $c' \equiv 0 \pmod{a}$. If $a > 1$, then most of the disjuncts will be unsatisfiable and can be safely eliminated from the expansion, significantly reducing the number of disjuncts. The following three rules are designed for this particular optimization.

Constant RHS: First, define the following abbreviations:

$$
\begin{array}{rcl}
|t| & := & \textbf{if } t \geq 0 \textbf{ then } t \textbf{ else } -t \\
\mathrm{sign}(x)\cdot y & := & \textbf{if } x > 0 \textbf{ then } y \textbf{ elsif } x = 0 \textbf{ then } 0 \textbf{ else } -y \\
j(c,\, b,\, a) & := & \begin{cases} b > 0: & (c+b) \bmod a \\ b < 0: & (a-(c+b)) \bmod a \end{cases}
\end{array}
$$

Intuitively, $j(c,\, b,\, a)$ is the minimal distance from b towards 0 to make $c+b$ divisible by a. That is, $c + b - \mathrm{sign}(b)\cdot j(c,\, b,\, a) \equiv 0 \pmod{a}$. Using the above notation, the proof rules for the case when t_2 is a constant in $\mathbf{G}(t_1,\, t_2,\, n)$ are written as follows:

$$\frac{\mathbf{G}(a\cdot x,\, c:\mathcal{N},\, b:\mathcal{N}) \quad b\neq 0 \quad j(c,\, b,\, a) > |b|}{\mathsf{false}}\text{Expand}\,\mathbf{G}\,\text{const}_0$$

$$\frac{\mathbf{G}(a\cdot x,\, c:\mathcal{N},\, b:\mathcal{N}) \quad b\neq 0 \quad j(c,\, b,\, a) \leq |b| < a + j(c,\, b,\, a)}{a\cdot x = c + b - \mathrm{sign}(b)\cdot j(c,\, b,\, a)}\text{Expand}\,\mathbf{G}\,\text{const}_1$$

$$\frac{\mathbf{G}(a\cdot x,\, c:\mathcal{N},\, b:\mathcal{N}) \quad b\neq 0 \quad a + j(c,\, b,\, a) \leq |b|}{\begin{array}{ll} & a\cdot x = c + b - \mathrm{sign}(b)\cdot j(c,\, b,\, a) \\ \vee & \mathbf{G}(a\cdot x,\, c,\, b - \mathrm{sign}(b)\cdot(a + j(c,\, b,\, a))) \end{array}}\text{Expand}\,\mathbf{G}\,\text{const.}$$

This completes the set of proof rules used in our implementation of Omega test in CVC. These rules can be thought of as a practical axiomatization of linear arithmetic, where every step in the decision procedure has a corresponding proof rule justifying that step.

3.7 Conclusion

This chapter presents the theory and some implementation detail of an online and proof-producing decision procedure for a theory of mixed-integer linear arithmetic extended with the $\mathsf{int}()$ predicate. Additionally, the decision procedure supports arbitrary precision arithmetic.

A decision procedure is much more useful to the research community when it can be combined with other decision procedures. Therefore, designing a stand-alone decision procedure is only the very first step in the design process. The next and more difficult task is to enhance the algorithm with additional properties which enable it to communicate with other decision procedures. Namely, the decision procedure must be online and must support backtracking. Furthermore, it is very useful for the decision procedure to be proof-producing. A proof-producing decision procedure has the nice property that its work can be checked by an external proof checker. Additionally, as explained above, proofs can be used to extract conflict clauses, thus making the top-level SAT solver of the combination framework more efficient.

In our experience, conceptually the most difficult is the online property. Just adapting the original Omega-test to an online algorithm required significant effort. Proof production is the next difficult problem in the design process. It could have easily been the hardest one if CVC did not already have a thoroughly worked-out methodology for adding proof production to existing decision procedures. Nevertheless, the implementation and especially debugging of proof production still presents a challenge. Finally, backtracking is relatively easy to design and implement in the context of CVC, since the framework provides all the necessary data structures.

Since our algorithm is largely based on Omega-test, its performance is comparable with that of the original implementation of Omega-test. The overhead of the CVC framework and the additional requirements on the decision procedure slow it down by about a factor of 10. This is a very reasonable price to pay for having an arithmetic decision procedure as part of powerful combination framework such as CVC.

This reimplementation adds arbitrary precision arithmetic, and generally is much more stable than the Omega library code. The arbitrary precision arithmetic is crucial for solving sizable systems of mixed-integer constraints using Fourier-Motzkin approach, since repeatedly generating shadow constraints produces large coefficients even if the coefficients in the original input are relatively small.

Chapter 4

Bit-Vectors and Arrays

4.1 Introduction

Linear arithmetic over the integers and reals discussed in the previous chapters is very useful in modeling behavior of computer programs. For example, many program analysis applications have used the integer linear arithmetic implementation in tools such as CVC [SBD02] and CVC Lite [BB04] to capture program behavior. However, the arithmetic used by digital computers is bounded, and consequently it is often more efficient or appropriate for a variety of applications [CGP$^+$06, Aik06, XA07] to capture program behavior in terms of bounded or bit-vector arithmetic. Also, applications such as automated bug finders [CGP$^+$06, NBFS06, BHL$^+$07] require that the memory of programs be modeled as well. The memory of a program can easily be represented as an array. Other applications that require support for bounded arithmetic and arrays include hardware verification and theorem proving. All these applications provide a strong motivation for research into efficient decision procedures for the theory of bit-vectors and arrays.

Decision procedures for the theory of bit-vectors and arrays have now been implemented as part of several tools. Important and recent examples of such a decision procedure include Yices [DdM06], SVC [BDL96], CVC Lite [BB04], UCLID [LS04] and Z3 [Bd07]. In this chapter we introduce STP [GD07a, CGP$^+$06, GD07b], a decision procedure for the quantifier-free first order logic with bit-vector and array datatypes.

Although theorem-proving and hardware verification have been the primary users of decision procedures for the theory of bit-vectors and arrays, they are increasingly being used in large-scale program analysis, bug finding and test generation tools. These applications often symbolically analyze code and generate constraints for the decision procedure to solve, and use the results from the decision procedure to further guide the analysis or generate new test cases.

The software analysis tools create demands on decision procedures that are different from those imposed by hardware applications. These applications often generate very large array constraints, especially when they choose to model system memory as one or more arrays. Also, there is a need for reasoning about mod-2^n arithmetic, which is an important source of incorrect system behavior. The constraint problems are often very large and extremely challenging to solve.

The design of STP has been driven primarily by the demands of software analysis research projects. As of June 2007, STP is being used in dozen or so software analysis, bug finding and hardware verification research projects in academia and industry. Many of these applications rely critically on the robustness and efficiency of STP to be effective. Notable among these is the EXE project [CGP$^+$06] at Stanford University, which generates test cases for C programs using symbolic execution, and uses STP to solve the constraints generated during symbolic execution. Other projects include the Replayer project [NBFS06] at Carnegie Mellon University, a tool that replays an application dialog between two hosts communicating over a computer network, with the goal of analyzing security exploits. Minesweeper [BHL$^+$07] is another project at Carnegie Mellon University, that automatically analyzes certain malicious behavior in Unix utilities and malware by symbolically executing them, and generating constraints for STP to solve. The CATCHCONV project [MWS07] at Berkeley tries to catch errors due to type conversion in C programs. The CATCHCONV project produced the largest example solved by STP so far. It is a 412 Mbyte formula, with 2.12 million 32-bit bit-vector variables, array write terms which are tens of thousands of levels deep, a large number of array reads with non-constant indices (corresponding to aliased reads in memory), many linear constraints, and liberal use of bit-vector functions and predicates, and STP solves it in approx. 2 minutes on a 3.2 GHz Linux box (This result was recorded in January 2007).

4.2 Preliminaries

We first discuss the standard model for the theory of bit-vectors and arrays, before formally defining its terms and formulas. In the standard model, bit-vectors are simply finite fixed-length strings over the alphabet $\{0, 1\}$ (the empty string is not part of the model). The rightmost *bit* is called the Least Significant Bit (LSB), and the leftmost one is called the Most Significant Bit (MSB). These strings are understood to be the binary representations of the natural numbers. For example, 0110 is a 4-bit representation of the natural number 6. Arrays are finite ordered lists of bit-vectors with n entries, where each location is indexed by a bit-vector, and the indices range from 0 to $n - 1$.

The terms in this theory are recursively defined below. Although the theory is first-order, there is some implicit typing. A term is either of type bit-vector of certain length, or an array term. For example, array write terms correspond to an array, and hence cannot occur in any of the bit-vector terms except as the first argument in an array read term.

1. constants that are finite positive length strings over the alphabet $0, 1$ whose length remains fixed, e.g. $0, 1, 01, 001, 0101..$

2. variables usually represented by letters such x,y,z... each denoting a bit-vector with fixed finite length

3. arithmetic operations over terms such as addition(+), subtraction(-), unary minus, multiplication(*), signed and unsigned division(/), signed and unsigned modulo ($\%$). All operands or arguments for these functions are required to be of the same length

4. concatenation(@) of two terms t_1 and t_2, where the length of resulting terms is the sum of the lengths of t_1 and t_2

5. extraction of consecutive bits over a term t represented as t[i:j], where i,j are non-negative integers and $i \geq j$, and length of t[i:j] is $i - j + 1$

6. left and right shift of a term by a natural number

7. bitwise Boolean operations like bitwise and, or and negation. All operands of these functions are required to be of the same length

8. sign extension

9. array read terms represented as $read(A, i)$, where A denotes an array and i denotes a term that indexes into the array

10. array write terms represented as $write(A, i, val)$ where A denotes an array, i is an index term as above, and val is a term. It is worth noting here that array write terms implicitly refer to a new array which possibly differs from A at location i.

11. if-then-else terms (also called ite terms), written as $ite(c, t_1, t_2)$, where c is a formula, and t_1 and t_2 are terms

Formulas are defined in the usual way as Boolean combinations of atomic formulas or negation of atomic formulas. An atomic formula is either

1. An equality over terms, written as $t_1 = t_2$, where t_1,t_2 are terms

2. An unsigned inequality over terms, written as $t_1 < t_2$, and in the signed case as $t_1 <_s t_2$

4.2.1 Semantics

Constants are interpreted as themselves in the standard model. The ite term $ite(c, t_1, t_2)$ evaluates to t_1 under a variable assignment ρ if the formula c evaluates to true in the standard model under ρ, and t_2 otherwise. The remaining operators or functions have the same semantics as that of the corresponding functions in the C programming language as defined over unsigned operands, and extended to arbitrary length bit-vectors. The standard definitions for the C programming language can be found at International Standards Organizations document ISO/IEC 9899:1999 [ISO99] (more popularly referred to as the C99 standard). For example, arithmetic operations, with the exception of signed division and modulo, follow the laws of two's complement arithmetic (as given in the C99 standard or described in a standard programming language textbook).

The formula $t_1 = t_2$ is true in the standard model for some variable assignment ρ, if both t_1 and t_2 evaluate to the same bit-vector under ρ. The formula $t_1 < t_2$ is true in the standard model for some variable assignment ρ, if the natural number corresponding to the bit-vector for t_1 under ρ is smaller than the natural number corresponding to t_2 under ρ. The evaluation of $t_1 <_s t_2$ depends on the top bits of t_1 and t_2. If the top bits of t_1 and t_2 are 1 under ρ, then they are construed as negative numbers. The formula $t_1 <_s t_2$ evaluates to true if the integer corresponding to the bit-vector t_1 is indeed less than the one corresponding to t_2, and false otherwise.

4.3 Related Work

There has been considerable amount of work for decision procedures for various quantifier-free fragments of the theory of bit-vectors and arrays. It has been shown that the satisfiability problem for various fragments of the theory of bit-vectors and arrays is NP-complete [BDL98, SBDL01, Möl98]. It is easy to establish that the satisfiability problem for the theory under consideration in this chapter is also NP-complete.

In the past, arrays and bit-vectors were often treated as separate theories, unlike in STP, where they are treated as a single theory. Consequently, our discussion of previous work is split into separate subsections for bit-vectors and arrays.

4.3.1 Bit-vectors

Decision procedures for bit-vectors can be categorized into one of the following:

1. Translation to SAT and variants: It is well known that the quantifier-free formulas in the theory of bit-vectors presented here can be equivalently translated into quantifier-free formulas over Boolean variables, thus reducing the satisfiability problem over bit-vectors to the satisfiability problems over Boolean variables or SAT. One such method is discussed in detail in [GBD07]. A SAT solver is then employed to solve the resultant SAT problem. Often preprocessing steps precede the translation to SAT. Examples of tools that use this approach include Cogent [CKS05], CVC Lite [BB04, GBD07], and STP, the tool discussed in this chapter.

   This method takes advantage of the ever-increasing raw efficiency of SAT solvers. Furthermore, modern SAT solvers provide hooks which allow their users to modify them in order to make them more effective in handling bit-vector arithmetic (a feature not yet widely or fully exploited). On the other hand, the primary drawback of the method is that a naïve translation to SAT without preprocessing does not take advantage of the inherent structure in the input problem.

Often effective preprocessing in the form of *solvers* for linear arithmetic, algebraic transformations and simplifications, and appropriate use of *normalizers* can overcome this drawback, and make this approach efficient for most of bit-vector arithmetic, except possibly the non-linear fragment. At this point in time, more research needs to be done to see whether this architecture (i.e. preprocessing followed by SAT) is fundamentally suited for non-linear bit-vector arithmetic.

2. Shostak-style procedures: Previous work by Cyrluk, et al. [CMR97] and Barrett, et al. [BDL98] (in their work on the Stanford Validity Checker or SVC) on decision procedures for bit-vector arithmetic centered on using a Shostak-style approach [RS01, Sho84] of using a *canonizer* [Sho84] followed by a *solver* [Sho84] for that theory. The primary motivation of such an approach was to enable the decision procedure to be *combined* with other Shostak-style decision procedures [RS01]. In a Shostak-style combination each decision procedure is required to have a canonizer and solver in order for the combination to be sound and complete.

   These methods have so far been developed only for linear bit-vector arithmetic with concatenation, extraction and bitwise Boolean operations. More importantly, they have not been shown to be competitive vis-a-vis other approaches such as translation to SAT (with preprocessing) or *abstraction-refinement* based methods. However, in our work with STP we found that solvers for linear arithmetic (similar to the one developed for SVC [BDL98]) can be very useful in eliminating variables from the input formulas, thus making the formulas easier to decide by the subsequent steps in STP.

3. Procedures for modular (or residue) and bounded arithmetic: There is a large body of work on using procedures based for modular (or residue) arithmetic to decide both linear and non-linear bit-vector arithmetic. We first focus on procedures for linear arithmetic.

- Linear solvers based on Gaussian elimination for modular arithmetic: Gaussian elimination (or its variants like Gauss-Jordan elimination method) is a very well-known algorithm which determines whether a system of linear equations over the rationals has a solution or not (i.e. is satisfiable or not). Several authors have modified this technique to solve systems of linear equations in modular arithmetic [New67, ST67, HG69a, HG69b]. An important characteristics of these algorithms is that they are offline, or not online. For a variety of reasons discussed below, it is very useful in the context of decision procedures like STP that the algorithms employed in them be online.

- Solve-and-Substitute methods: Several methods have been proposed [CLR98] based on the solve-and-substitute paradigm. In this paradigm, a variable is solved for, and then substituted away from the system. STP also uses the same paradigm to decide the linear fragment of bit-vector arithmetic. The difference is that in STP we can handle systems of linear equations over any number of variables, while the previously proposed methods are for solving systems of congruences over a single variable.

- Linear solver based on the p-adic method: A method has been proposed by G. Malaschonok [Mal03] for the linear fragment of bit-vector arithmetic. Unfortunately, due to the unavailability of an implementation it is difficult for us compare and contrast this method with others. A method for solving non-linear arithmetic is proposed by Babic and Musuvathi [BM05].

4. Translation to automata (WS1S): In [Möl98] and other papers translation of bit-vector arithmetic to weak second order logic with one sucessor [Büc60] (WS1S) has been proposed. WS1S is a decidable fragment of second order logic. Using the MONA tool [EKM98] the bit-vector formulas are transformed to finite automata and solved. It has been noted in [Möl98] that such an approach is not very efficient relative to other methods.

5. Procedures based on the Abstraction-Refinement Paradigm: Recently, Bryant, et

al. [BKO^{+}07] have extended the abstraction-refinement paradigm to the case of bit-vector arithmetic as part of the UCLID tool [BKO^{+}07]. In brief, the method alternatively computes an over-approximation and an under-approximation of the input formula, until the correct result is obtained. It is hoped that approximations are easier to solve than the original formula, and the number of iterations to compute the appropriate approximations is relatively small in practice. In this work, STP was compared with UCLID and Yices [DdM06] over examples involving only bit-vector arithmetic. It was noted that on certain non-linear arithmetic benchmarks UCLID outperforms STP. This is a consequence of the fact that STP does not do any special processing for the non-linear fragment of bit-vector arithmetic.

4.3.2 Arrays

In recent years, there has been a lot of work on the theory of arrays. The first decision procedure for the quantifier-free *extensional* theory of arrays is discussed by Stump, et al. [SBDL01] (Extensional means that the theory supports equality over array terms. More precisely, for each index if the elements of the two arrays are equal, then the arrays are equal). In their recent paper [BMS06b], Bradley, Manna and Sipma discuss decidability results for extensional theory of arrays with limited quantification. As far as actual decision procedure tools are concerned, Z3, Yices [DdM06], BAT [MSV07] have support for quantifier-free extensional theory of arrays, while UCLID [LS04] and STP have support for non-extensional quantifier-free theory of arrays.

4.3.3 STP in Comparison with Other Tools

STP's architecture is different from most decision procedures that support both bit-vectors and arrays [SBD02, BB04, DdM06], which are based on backtracking and a framework for combining specialized theories such as Nelson-Oppen [NO79]. Instead, STP consists of a series of word-level transformations and optimizations that eventually convert the original problem to a conjunctive-normal form (CNF) formula for input to a high-speed solver for the satisfiability problem for propositional logic formulas (SAT) [ES03]. Thus, STP fully exploits the speed of modern SAT solvers while also taking advantage of theory-specific optimizations for bit-vectors and arrays. In this respect, STP is most similar to UCLID [LS04].

The goal of this chapter is to describe the factors that enable STP to handle the large constraints from software applications. In some cases, simple optimizations or a careful decision about the ordering of transformations can make a huge difference in the capacity of the tool. In other cases, more sophisticated optimizations are required. Two are discussed in detail: An on-the-fly solver for mod-2^n linear arithmetic, and abstraction-refinement heuristics for array expressions. The rest of the chapter discusses the architecture of STP, the basic engineering principles, and then goes into more detail about the optimizations for bit-vector arithmetic and arrays. Performance on large examples is discussed, and there is a comparative evaluation with Yices and Z3, both of which are well known for their efficiency.

4.4 STP Overview

STP's input language has most of the functions and predicates implemented in a programming language such as C or a machine instruction set, except that it has no floating point datatypes or operations. The current set of operations supported include $TRUE$, $FALSE$, propositional variables, arbitrary Boolean connectives, bitwise Boolean operators, extraction, concatenation, left and right shifts, addition, multiplication, unary minus, (signed) division and modulo, array read and write functions, and relational operators. The semantics parallel the semantics of the SMTLIB bit-vector language [RT06] or the C programming language, except that in STP bit-vectors can have any positive length, not just 32 or 64 (as is the case with current C language standards). Also, all arithmetic and bitwise Boolean operations require the inputs to have the same length. STP can be used as a stand-alone program, and can parse input files in a special human readable syntax and also the SMTLIB QF_UFBV syntax [RT06]. It can also be used as a library, and has a special C-language API that makes it relatively easy to integrate with other applications. It must be emphasized that STP is designed to primarily deal with conjunction of literals, and not with arbitrary Boolean combinations of them.

STP converts a decision problem in its logic to propositional CNF, which is solved with a high-performance off-the-shelf CNF SAT solver (Boolean SAT solver), MiniSat [ES03] (MiniSat has a nice API, and it is concise, clean, efficient, reliable, and relatively unencumbered by licensing conditions). However, the process of converting to CNF includes many word-level transformations and optimizations that reduce the difficulty of the eventual SAT problem. Problems are frequently solved during the transformation stages of STP, so that SAT does not need to be called.

STP's architecture differs significantly from many other decision procedures based on case splitting and backtracking, including tools like SVC, and CVC Lite [BDL96, BB04], and other solvers based on the Davis-Putnam-Logemann-Loveland (DPLL(T)) architecture [GHN$^+$04]. Conceptually, those solvers recursively assert atomic formulas and their negations to a theory-specific decision procedures to check for consistency with formulas that are already asserted, backtracking if the current combination of assertions is inconsistent. In recent versions of this style of decision procedure, the choice of formulas to assert is made by a conventional DPLL SAT solver, which treats the formulas as propositional variables until they are asserted and the decision procedures invoked.

Architectures based on assertion and backtracking invoke theory specific decision procedures in the “inner loop” of the SAT solver. However, modern SAT solvers are very fast largely because of the incredible efficiency of their inner loops, and so it is difficult with these architectures to take the best advantage of fast SAT solvers.

STP on the other hand does all theory-specific processing *before* invoking the SAT solver. The SAT solver works on a purely propositional formula, and its internals are not modified, including the highly optimized inner loop. Optimizing transformations are employed before the SAT solver when they can solve a problem more efficiently than the SAT solver, or when they reduce the difficulty of the problem that is eventually presented to the SAT solver.

DPLL(T) solvers often use Nelson-Oppen combination [NO79], or variants thereof, to link together multiple theory-specific decision procedures. Nelson-Oppen combination needs the individual theories to be disjoint, stably-infinite and requires the exchange of equality relationships deduced in each individual theory, leading to inflexibility and implementation complexity. In return, Nelson-Oppen combination ensures that the combination

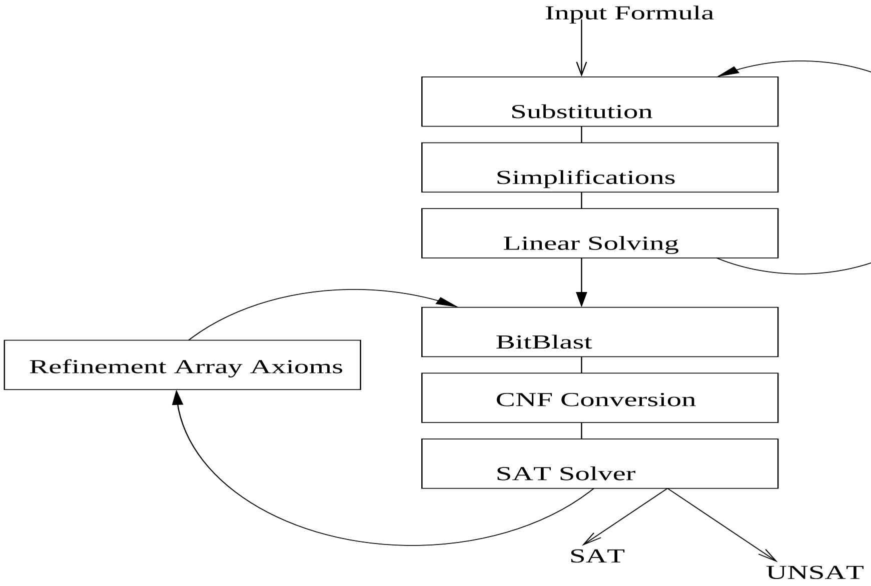

Figure 4.1: STP Architecture

of theories is complete. STP is complete because the entire formula is converted by a set of satisfiability preserving steps to CNF, the satisfiability of which is decided by the SAT solver. So there is no need to worry about meeting the conditions of Nelson-Oppen combination. Furthermore, the extra overhead of communication between theories in the Nelson-Oppen style decision procedures can become a bottleneck for the very large inputs that we have seen, and this overhead is avoided in STP.

The STP approach is not always going to be superior to a good backtracking solver. A good input to STP is a conjunction of many formulas that enable local algebraic transformations. On the other hand, formulas with top-level disjunctions may be very difficult, which may be much easier with a DPLL(T) style approach. Fortunately, the software applications used by STP tend to generate large conjunctions, and hence STP's approach has worked well in practice.

In more detail, STP's architecture is depicted in Figure 4.1. Processing consists of three phases of word-level transformations; followed by conversion to a purely Boolean formula and Boolean simplifications (this process is called "Bit Blasting"); and finally conversion to propositional CNF and solving by a SAT solver. The primary focus of this chapter is on word-level optimizations for arithmetic and arrays, and also refinement for arrays.

Expressions are represented as directed acyclic graphs (DAGs), from the time they are created by the parser or through the C-interface, until they are converted to CNF. In the DAG representation, isomorphic subtrees are represented by a single node, which may be pointed to by many parent nodes. This representation has advantages and disadvantages, but the overwhelming advantage is compactness.

It is possible to identify some design principles that have worked well during the development of STP. The overarching principle is to procrastinate when faced with hard problems. That principle is applied in many ways. Transformations that are risky because they can significantly expand the size of the expression DAG are postponed until other, less risky, transformations are performed. The hope is that the less risky transformation will reduce the size and number of expressions before the more risky transformations are applied. This approach is particularly helpful for array expressions.

Counter-example-guided abstraction-refinement is now a standard paradigm in formal tools, which can be applied in a variety of ways. It is another application of the procrastination principle. For example, the UCLID tool abstracts and refines the precision of integer variables.

A major novelty of STP's implementation is the particular implementation of the refinement loop in Figure 4.1. In STP, abstraction is implemented (i.e., an *abstract formula* is obtained) by omitting conjunctive constraints from a *concrete formula,* where the concrete formula must be equisatisfiable with the original formula. (Logical formulas ϕ and ψ are equisatisfiable iff ϕ is satisfiable exactly when ψ is satisfiable.)

When testing an abstract formula for satisfiability, there can be three results. First, STP can determine that the abstracted formula is unsatisfiable. In this case, it is clear that the original formula is unsatisfiable, and hence STP can return "unsatisfiable" without additional refinement, potentially saving a massive amount of work.

A second possible outcome is that STP finds a satisfying assignment to the abstract formula. In this case, STP converts the satisfying assignment to a (purported) concrete model [1], and also assigns zero to any variables that appear in the original formula but not the abstract formula, and evaluates the original formula with respect to the purported

[1] A model in this context refers to an assignment of constants to all of the variables in a formula such that the formula is *satisfied*

model. If the result of the evaluations is $TRUE$, the purported model is truly a model of the original formula (i.e., the original formula is indeed satisfiable) and STP returns the model without further refinement iterations.

The third possible outcome is that STP finds a purported model, but evaluating the original formula with respect to that model returns $FALSE$. In that case, STP refines the abstracted formula by heuristically choosing additional conjuncts, at least one of which must be false in the purported model and conjoining those formulas with the abstracted formula to create a new, less abstract formula. In practice, the abstract formula is not modified; instead, the new formulas are bit-blasted, converted to CNF, and added as clauses to the CNF formula derived from the previous abstract formula, and the resulting CNF formula solved by the SAT solver.

This process is iterated until a correct result is found, which must occur because, in the worst case, the abstract formula will be made fully concrete by conjoining every formula that was omitted by abstraction. When all formulas are included, the result is guaranteed to be correct because of the equisatisfiability requirement above.

4.5 Arrays in STP

As was mentioned above, arrays are used heavily in software analysis applications, and reasoning about arrays has been a major bottleneck in many examples. STP's input language supports one-dimensional arrays [SBDL01] that are indexed by bit-vectors and contain bit-vectors. The theory is not *extensional*. The operations on arrays are $read(A, i)$, which returns the value at location $A[i]$ where A is an array and i is an index expression of the correct type, and $write(A, i, v)$, which returns a new array with the same value as A at all indices except possibly i, where it has the value v. The value of a $read$ is a bit-vector, which can appear as an operand to any operation or predicate that operates on bit-vectors. The value of an array variable or an array write has an array type, and may appear only as the first operand of a $read$ or $write$, or as the *t*hen or *e*lse operand of an if-then-else. In particular, values of an array type cannot appear in an equality or any other predicate.

In the unoptimized mode, STP reduces all formulas to an equisatisfiable form that contains no array $read$s or $write$s, using three transformations. (In the following, the expression $ite(c_1, e_1, e_2)$ is shorthand for *if* c_1 *then* e_1 *else* e_2 *endif*.) These transformations are all standard.

The **Ite-lifting** transformation converts

$$read(ite(c, write(A, i, v), e), j)$$

into

$$ite(c, read(write(A, i, v), j), read(e, j))$$

where e is an arbitrary array expression (There is a similar transformation when the $write$ is in the "else" part of the ite). The **read-over-write** transformation eliminates all write terms by transforming $read(write(A, i, v), j)$ to $ite(i = j, v, read(A, j))$. Finally, the **read elimination** transformation eliminates $read$ terms by introducing a fresh bit-vector variable for each such expression, and adding more predicates to ensure consistency. Specifically, whenever a term $read(A, i)$ appears, it is replaced by a fresh variable v, and new predicates are conjoined to the formula $i = j \Rightarrow v = w$ for all variables w introduced in place of read terms $read(A, j)$, having the same array term as first operand. As an example of this transformation, the simple formula $(read(A, 0) = 0) \wedge (read(A, i) = 1)$ would be transformed to $v_1 = 0 \wedge v_2 = 1 \wedge (i = 0 \Rightarrow v_1 = v_2)$. The formula of the form $(i = 0 \Rightarrow v_1 = v_2)$ is called an *array read axiom*.

4.5.1 Optimizing Array Reads

Read elimination, as described above, expands each formula by up to $n(n-1)/2$ nodes, where n is the number of syntactically distinct index expressions. Unfortunately, software analysis applications can produce thousands of reads with variable indices, resulting in a lethal blow-up when this transformation is applied. While this blow-up seems unavoidable in the worst case, appropriate procrastination leads to practical solutions for many very large problems. Two optimizations which have been very effective are *array substitution* and abstraction-refinement for reads, which we call *read refinement*.

The array substitution optimization reduces the number of array variables by substituting out all constraints of the form $read(A, c) = e_1$, where c is a constant and e_1 does not contain another array read. Programs often index into arrays or memory using constant indexes, so this is a case that occurs often in practice.

The optimization has two passes. The first pass builds a substitution table with the left-hand-side of each such equation ($read(A, c)$) as the key and the right-hand-side (e_1) as the value, and then deletes the equation from the input query. The second pass over the expression replaces each occurrence of a key by the corresponding table entry. Note that for soundness, if a second equation is encountered whose left-hand-side is already in the table, the second equation is not deleted and the table is not changed. For example, if STP saw $read(A, c) = e_1$ then $read(A, c) = e_2$, the second formula would not be deleted and would later be simplified to $e_1 = e_2$.

The second optimization, *read refinement*, delays the translation of array *read*s with non-constant indexes in the hope of avoiding read elimination blowup. Its main trick is to solve a less-expensive approximation of the formula, check the result in the original formula, and try again with a more accurate approximation if the result is incorrect.

Read formulas are abstracted by performing read elimination (*i.e.*, replacing reads with new variables), but not adding the array read axioms. This abstracted formula is processed by the remaining stages of STP. As discussed in the overview, if the result is unsatisfiable, that result is correct and can be returned immediately from STP. If not, the abstract model found by STP is converted to a concrete model and the original formula is evaluated with respect to that model. If the result is $TRUE$, the answer is correct and STP returns that model. Otherwise, some of the array read axioms from read elimination are added to the formula and STP is asked to satisfy the modified formula. This iteration repeats until a correct result is found, which is guaranteed to happen (if memory and time are not exhausted) because all of the finitely many array read axioms will eventually be added in the worst case.

The choice of which array read axioms to add during refinement is a heuristic that is important to the success of the method. A policy that seems to work well is to find a non-constant array index term for which at least one axiom is violated, then add all of the violated axioms involving that term. Adding at least one false axiom during refinement guarantees that STP will not find the same false model more than once. Adding all the axioms for a particular term seems empirically to be a good compromise between adding just one formula, which results in too many iterations, and adding all formulas, which eliminates all abstraction after the first failure.

For example, suppose STP is given the formula $(read(A, 0) = 0) \wedge (read(A, i) = 1)$. STP would first apply the substitution optimization by deleting $read(A, 0) = 0$ from the formula, and inserting the pair $(read(A, 0), 0)$ in the substitution table. Then, it would replace $read(A, i)$ by a new variable v_i, thus generating the under-constrained formula $v_i = 1$. Suppose STP finds the solution $i = 1$ and $v_i = 1$. (Note that even though the variable i is not part of the under-constrained formula, STP randomly assigns it a value. This is clearly a sound operation.)

STP then translates the solution to the variables of the original formula to get the following: $(read(A, 0) = 0)$ and $read(A, 1) = 1)$. This solution is satisfiable in the original formula as well, so STP terminates since it has found a true satisfying assignment.

However, suppose that STP finds the solution $i = 0$ and $v_i = 1$. Under this solution, the original formula eventually evaluates to $read(A, 0) = 0$ and $read(A, 0) = 1$, which after substitution gives $0 = 1$. Hence, the solution to the under-constrained formula is not a solution to the original formula.

In this case, STP adds the array read axiom $i = 0 \Rightarrow read(A, i) = read(A, 0)$. When this formula is checked, the result must be correct because the new formula includes the complete set of array read axioms.

4.5.2 Optimizing Array Writes

Efficiently dealing with array writes is crucial to STP's utility in software applications, some of which produce deeply nested write terms when there are many successive assignments to indices of the same array. The **read-over-write** transformation creates a performance bottleneck by destroying sharing of sub terms, creating an unacceptable blow-up in DAG size. Consider the simple formula:

$$read(write(A, i, v), j) = read(write(A, i, v), k)$$

in which the $write$ term is shared. The **read-over-write** transformation translates this to

$$ite(i = j, v, read(A, j)) = ite(i = k, v, read(A, k))$$

When applied recursively to the deeply nested $write$ terms, it essentially creates a new copy of the entire DAG of write terms for every distinct read index, which exhausts memory in large examples. The resulting blow-up is $O(n * m)$, where n is number of read indices to the shared write term, and m is nesting depth of the write term.

Once again, the procrastination principle applies. The **read-over-write** transformation is delayed until after other simplification and solving transformations are performed, except in special cases like $read(write(A, i, v), i + 1)$, where the read and write indices simplify to terms that are always equal or not equal. In practice, the simple transformations convert many index terms to constants. The **read-over-write** transformation is applied in a subsequent phase. When that happens, the formula is smaller and contains more constants. This simple optimization is enormously effective, enabling STP to solve many very large problems with nested writes that it is otherwise unable to do.

Abstraction and refinement is also used on write expressions, when the previous optimization leaves large numbers of $read$s and $write$s untouched, leading to major speed-ups on some large formulas. For this optimization, array read-over-write terms are replaced by new variables to yield a conjunction of formulas that is equisatisfiable to the original set. The example above is transformed to:

$$\begin{array}{rcl} v_1 & = & v_2 \\ v_1 & = & ite(i = j, v, read(A, j)) \\ v_2 & = & ite(i = k, v, read(A, k)) \end{array}$$

where the last two formulas are called *array write axioms*. For the abstraction, the array write axioms are omitted, and the abstracted formula $v_1 = v_2$ is processed by the remaining phases of STP. As with array reads, the refinement loop iterates only if STP finds a model of the abstracted formula that is also not a model of the original formula. Write axioms are added to the abstracted formula, and the refinement loop iterates with the additional axioms until a definite result is produced.

The success of the abstraction-refinement heuristic in the case of array writes can be illustrated with the following conjunction of simple literals:

$$\begin{array}{rcl} \mathit{read}(\mathit{write}(A,i,v),j)=0 & \wedge & \mathit{read}(\mathit{write}(A,i,v),k)=1 \\ (i=j)\wedge(i\neq k) & \wedge & (v\neq 0) \end{array}$$

Observe that there is a shared write term in the above formula. Also, since $i = j$, the first literal is equivalent to $v = 0$. However, this contradicts one of the other literals $v \neq 0$ from the input, and hence the formula is unsatisfiable in actuality. In the abstraction phase, the above formula is transformed into:

$$\begin{array}{rcl} t_1=0 & \wedge & t_2=1 \\ (i=j)\wedge(i\neq k) & \wedge & (v\neq 0) \end{array}$$

where t_1 and t_2 are new variables. This abstracted formula is then fed to the SAT solver stage of STP. It is easy to see that the abstracted formula is satisfiable. This satisfying assignment is clearly not an assignment for the original formula. This prompts STP to refine. STP chooses the axiom that is false in the current assignment, namely $t_1 = \mathit{ite}(i = j, v, \mathit{read}(A, j))$. Adding this axiom forces STP to conclude that $\mathit{ite}(i = j, v, \mathit{read}(A, j)) = 0$, by substitution of t_1. Further simplification steps reduce this deduced formula to $v = 0$. But, v is not equal to 0 in the input and the abstracted formula. This prompts STP to declare unsatisfiability and stop. In the above example only one of the axiom was needed to get the correct result. This is precisely the kind of scenario where STP has a big win. STP avoids doing the **read-over-write** transformation and does not have to refine all the axioms. More generally, a large number of real-world examples are characterized by write terms where many of the reads don't actually read anything of consequence from the shared write terms. Such examples are prime targets for the abstraction-refinement based heuristic.

4.6 Linear Solver and Variable Elimination

One of the essential features of STP for software analysis applications is its efficient handling of linear twos-complement arithmetic. The heart of this is an *on-the-fly* solver. The main goal of the solver is to eliminate as many bits of as many variables as possible, to reduce the size of the transformed problem for the SAT solver. In addition, it enables many other simplifications, and can solve purely linear problems outright, so that the SAT solver does not need to be used.

The solver solves for one equation for one variable at a time. That variable can then be eliminated by substitution in the rest of the formula, whether the variable occurs in linear equations or other formulas. In some cases, it cannot solve an entire variable, so it solves for some of the low-order bits of the variable. After bit-blasting, these bits will not appear as variables in the problem presented to the SAT solver. Non-linear or word-level terms (extracts, concatenations etc.) appearing in linear equations are treated as bit-vector variables.

The algorithm has worst-case time running time of $O(k^2n)$ multiplications, where k is the number of equations and n is the number of variables in the input system of linear bit-vector equations.[2] If the input is unsatisfiable the solver terminates with $FALSE$. If the input is satisfiable it terminates with a set of equations in *solved form*, which symbolically represent all possible satisfying assignments to the input equations. So, in the special case where the formula is a system of linear equations, the solver leads to a sound and complete polynomial-time decision procedure. Furthermore, the equations are reduced to a closed form that captures all of the possible solutions.

Solved Form: A list of equations is in *solved form* if the following invariants hold over the equations in the list.

1. Each equation in the list is of the form $x[i : 0] = t$ or $x = t$, where x is a variable and t is a linear combination of the variables or constant times a variable (or extractions thereof) occurring in the equations of the list, except x

2. Variables on the left hand side of the equations occurring earlier in the list may not occur on the right hand side of subsequent equations. Also, there may not be two equations with the same left hand side in the list

[2] As observed in [BDL98], the theory of linear mod 2^n arithmetic (equations only) in tandem with concatenate and extract operations is NP-complete. Although STP has concatenate and extraction operations, terms with those operations are treated as independent variables in the linear solving process, which is polynomial. A hard NP-complete input problem constructed out of linear operations, concatenate and extract operations will not be solved completely by linear solving, and will result in work for the SAT solver.

3. If extractions of variables occur in the list, then they must always be of the form $x[i:0]$, i.e. the lower extraction index must be 0, and all extractions must be of the same length

4. If an extraction of a variable $x[i:0] = t$ occurs in the list, then an entry is made in the list for $x = x^1@t$, where x^1 is a new variable referring to the top bits of x and $@$ is the concatenation symbol

The algorithm is illustrated on the following system:

$$\begin{aligned} 3x + 4y + 2z &= 0 \\ 2x + 2y + 2 &= 0 \\ 4y + 2x + 2z &= 0 \end{aligned}$$

where all constants, variables and functions are 3 bits long. The solver proceeds by first choosing an equation and always checks if the chosen equation is *solvable*. It uses the following theorem from basic number theory to determine whether an equation is solvable or not: $\Sigma_{i=1}^{n} a_i x_i = c_i$ mod 2^b is solvable for the unknowns x_i if and only if the greatest common divisor of $\{a_1, \ldots, a_n, 2^b\}$ divides c_i (Note that, the numbers a_i and c_i are treated as part of a finite ring on the left side of the iff, while treated as integers on the right side).

In the example above, the solver chooses $3x + 4y + 2z = 0$ which is solvable since the $gcd(3, 4, 2, 2^3)$ does indeed divide 0. It is also a basic result from number theory that a number a has a multiplicative inverse mod m iff $\gcd(a, m) = 1$, and that this inverse can be computed by the extended greatest-common divisor algorithm [CLR98] or a method from [BDL98]. So, if there is a variable with an odd coefficient, the solver isolates it on the left-hand-side and multiplies through by the inverse of the coefficient. In the example, the multiplicative inverse of 3 mod 8 is also 3, so $3x + 4y + 2z = 0$ can be solved to yield $x = 4y + 6z$.

Substituting $4y + 6z$ for x in the remaining two equations yields the system

$$\begin{aligned} 2y + 4z + 2 &= 0 \\ 4y + 6z &= 0 \end{aligned}$$

where all coefficients are even. Note that even coefficients do not have multiplicative inverses in arithmetic mod 2^b, and, hence we cannot isolate a variable. However, it is possible to solve for *some bits* of the remaining variables.

The solver transforms the whole system of solvable equations into a system which has at least one summand with an odd coefficient. To do this, the solver chooses an equation which has a summand whose coefficient has the minimum number of factors of 2. In the example, this would the equation $2y + 4z + 2 = 0$, and the summand would be $2y$. The whole system is divided by 2, and the high-order bit of each variable is dropped, to obtain a reduced set of equations

$$\begin{aligned} y[1:0] + 2z[1:0] + 1 &= 0 \\ 2y[1:0] + 3z[1:0] &= 0 \end{aligned}$$

where all constants, variables and operations are 2 bits. Next, $y[1:0]$ is solved to obtain $y[1:0] = 2z[1:0] + 3$. Substituting for $y[1:0]$ in the system yields a new system of equations $3z[1:0] + 2 = 0$. This equation can be solved for $z[1:0]$ to obtain $z[1:0] = 2$. It follows that original system of equations is satisfiable. It is important to note here that the bits $y[2:1]$ and $z[2:1]$ are unconstrained. The solved form in this case is $x = 4y + 6z$ $\wedge$ $y[1:0] = 2z[1:0] + 3 \wedge z[1:0] = 2$ (Note that in the last two equations all variables, constants and functions are 2 bits long).

Algorithms for deciding the satisfiability of a system of equations and congruences in modular or residue arithmetic have been well known for a long time. However, most of these algorithms do not provide a solved form that captures all possible solutions. Some of the ideas presented here were devised by Clark Barrett and implemented in the SVC decision procedure [HC01, BDL98], but the SVC algorithm has exponential worst-case time complexity while STP's linear solver is polynomial in the worst-case.

The closest related work is probably in a paper by Huang and Cheng [HC01], which reduces a set of equations to a solved form by Gaussian elimination. On the other hand, STP implements an online solving and substitution algorithm that gives a closed form solution. Such algorithms are easier to integrate into complex decision procedures.

8495 tests	**Read Refinement ON**	**Read Refinement OFF**
Time Taken for all tests	624 sec	3378 sec
Number of Timeouts	1	36

Table 4.1: STP with array read abstraction-refinement switched on and off

More importantly, an online solver is crucial to the efficiency of decision procedures such as STP. The reason is that online solver solves incrementally, one equation at a time. It solves for a suitable variable in the input equation, and substitutes it away from the input system of constraints. When this variable is substituted away, many new opportunities for optimizations are opened up.

For example, consider the formula $x + 2y = 1 \wedge y = 0 \wedge ite(x = 1, \varphi_1, \varphi_2)$, where φ_1 and φ_2 are any arbitrary formulas. When y is substituted out and x is solved for by the linear solver, the value of x is 1. Once this value is substituted into the original formula, the result reduces to φ_1. Thus an optimization was triggered by the process of solving and substitution executed by the online solver.

4.7 Experimental Results

This section presents empirical results on large examples obtained from software analysis tools. The effects of abstraction-refinement algorithms and linear solving in STP are examined. Also, STP is compared with, Z3 [Bd07] and Yices [DdM06], two of the best tools among the various decision procedures that support both bit-vectors and arrays as of 2007. Yices is an offering from SRI International at Menlo Park, CA, USA. Z3 is an offering from Microsoft Research located at Redmond, WA, USA. Other tools which support this theory include CVC Lite, BAT and UCLID. The current implementation of CVC Lite is not efficient for this fragment, and hence we decided not to compare against it. Both BAT and UCLID do not support a format common with STP over the entire fragment involving arrays and bit-vectors. Consequently, it was hard to do a fair comparison.

In the Table 4.1 STP is compared in two modes: with read abstraction-refinement switched on, and STP with read abstraction-refinement switched off. The testcases for this experiment were obtained from the EXE automated bug finding tool [CGP^{+}06]. The

Example Name (Node Size)	Result	Lin-ON	Lin-OFF
testcase15 (0.9M)	sat	66	MO
testcase16 (0.9M)	sat	67	MO
thumbnailout-noarg (2.7M)	sat	840	MO
thumbnailout-spin1 (3.2M)	sat	115	MO
thumbnailout-spin1-2 (4.3M)	sat	1920	MO

Table 4.2: STP performance over large examples with linear solver on and off

total number of testcases used was 8495. The timeout was set at 60 seconds per testcase. The timeout limit was dictated by the needs of the EXE application which calls STP very often. Many similar program analyses based applications demand that the decision procedure be very quick in deciding the input or have a relatively small timeout limit. Notice that STP with read refinement switched on is 5 times faster than without read refinement, when the times for the individual testcases are aggregated. However, for many of the harder individual cases STP was 50-100 times faster with read-refinement switched on.

The more important metric is the number of testcases that timeout. As seen from the Table 4.1, this number falls dramatically when refinement is ON. If the timeout limit is increased to two minutes, then STP with refinement ON finishes on the remaining testcase as well. On the other hand, STP with refinement switched OFF continues to timeout on a large fraction of the testcases that timed out with a one minute timeout. Another metric that we tracked was the number of times STP had to go around the refinement loop during read refinement. For the examples mentioned in Table 4.1, STP never did more than 6 refinement loops, and most often went around the refinement loop only once.

In the Tables 4.2, 4.3, and 4.4 the names of examples are followed by their nodesize. Approximate node size are in either 1000s of nodes or millions of nodes. 1K denotes 1000 nodes, and 1M denotes one million nodes. Shared nodes are counted exactly once. NR stands for No Result. All timings are in seconds. MO stands for out of memory error. TO stands for the tool running for more than the timeout limit. These examples were generated using the CATCHCONV tool, Replayer and Minesweeper tools. The timeout was set at 35 minutes per example. All experiments were run on a 3.2GHz/2GB RAM 32-bit Intel machine running Debian GNU/Linux. The system gave out of memory exception

Example Name (Node Size)	**Result**	**WRITE Abstraction**	**NO WRITE Abstraction**
610dd9dc (15K)	sat	37	101
grep0084 (69K)	sat	18	506
grep0106 (69K)	sat	227	> 600
grep0777 (73K)	NR	MO	MO
testcase20 (1.2M)	sat	56	MO

Table 4.3: STP performance over examples with deep nested writes

when a process consumed 3.2 GB of memory. All tests in this section are available at *http://verify.stanford.edu/stp.html*.

In Table 4.2, STP is compared with linear-solving on (Lin-ON), and linear solving off (Lin-OFF) some of the very large examples obtained from real-world applications. These examples also contain a large number of arrays reads. Table 4.2 includes some of the hardest of the examples which have usually tens of megabytes of text, hundreds of thousands of 32 bit bit-vector variables, lots of array reads, and large number of linear constraints derived from [MWS07]. The value addition provided by the linear solver is clear from this table.

Table 4.3 summarizes STP's performance, with and without array write abstraction, on the big array examples with deeply nested writes. Table 4.3 includes examples with deeply nested array writes and modest amounts of linear constraints derived from various applications. The "grep" examples were generated using the Minesweeper tool while trying to discover malicious behavior in the Unix grep program. The 610dd9c formula was generated by a Minesweeper analysis of a program that was in a "botnet" attack. The formula `testcase20` was generated by CATCHCONV. As expected, STP with write abstraction-refinement ON yields a very large improvement over STP with write abstraction-refinement switched OFF, although it is not always faster.

Table 4.4 compares STP with Yices and Z3. For the comparison of STP with Yices and Z3, the examples chosen were obtained from different real-world applications that were using STP. These examples were also submitted to the SMTCOMP annual competition [RT06], held alongside the annual conference on Computer Aided Verification (CAV), where various decision procedures for theories such as linear arithmetic, bit-vectors and

Example	**Result**	**STP**	**Z3**	**Yices**
610dd9c (15K)	sat	37	TO	MO
grep65 (60K)	unsat	4	0.3	TO
grep84 (69K)	sat	18	176	TO
grep106 (69K)	sat	227	130	TO
blaster4 (262K)	unsat	10	MO	MO
testcase20 (1.2M)	sat	56	MO	MO

Table 4.4: STP vs. Z3 vs. Yices

arrays compete with each other. These examples were considered the hardest among all the industrial ones submitted to the competition.

4.8 Conclusion

Software applications such as program analysis, bug finding, and symbolic simulation of software tend to impose different conditions on decision procedures than hardware applications. In particular, arrays become a bottleneck. Also, the constraints tend to be very large with lots of linear bit-vector arithmetic in them. Abstraction-refinement algorithms is often helpful for handling large array terms. Also, the approach of doing phased word-level transformations, starting with the least expensive and risky transformations, followed by translation to SAT seems like a good design for decision procedures for the applications considered. Finally, linear solving, when implemented carefully, is effective in variable elimination.

Chapter 5

Conclusions

In this thesis, some issues relating to the design and implementation of practical and efficient decision procedures were addressed. The motivation for such research is strong, since efficient decision procedures are rapidly being adopted to effectively solve problems in various areas of computer science. The contributions are summarized below, along with a discussion of lessons learnt and some suggestions for future work.

5.1 Summary of Contributions

The design and algorithms introduced as part of the STP tool form an important contribution of this thesis. The SAT-based architecture of the STP tool has several advantages. First, it takes advantage of highly tuned off-the-shelf SAT solvers, which are increasing in efficiency with every passing year. At the same time STP compensates for the weaknesses of SAT solvers through suitable preprocessing steps. Second, such an approach dramatically simplifies the design and implementation. Also, STP supports two theories without relying on a combination based approach. A combination framework can introduce complexities into the design of the tool, that may make the task of building an efficient procedure difficult. Such costs are avoided in a SAT-based approach as used in STP.

The abstraction-refinement based heuristics for arrays discussed here take advantage of the fact that formulas generated from many real-world applications need not be processed in their entirety to determine satisfiability correctly. If the input formulas are suitably abstracted then the decision problem often becomes computationally much easier. Refinements may be necessary to get the correct result. However, in our experience the number of refinement loop was relatively constant and small, irrespective of the size of the input formula. In the examples with large number of array reads the average number of refinement was around 3 for the read refinements, and never exceeded 6.

In other contributions, a decision procedure was introduced for mixed real and integer linear arithmetic that was engineered to be online and proof-producing. A decision procedure that produces proof is attractive since its work can be checked by an external proof checker. These characteristics have been shown to be important in making the decision procedure efficient in the context of combinations. Also, a new decision procedure for quantifier-free Presburger arithmetic based on model-checking was introduced. One of the conclusions in this work was that the integer linear programming based methods consistently outperform all other methods for deciding quantifier-free Presburger arithmetic.

5.2 Lessons Learnt

The STP project took root because previous offerings like CVC and CVC Lite from our group were deemed inadequate by their users. Both CVC and CVC Lite had complex architectures based on combination frameworks [Bar03], were feature rich, but difficult to maintain and modify, and clearly not very efficient. In STP, it was decided that the architecture be made as simple as possible. The simplest, yet reasonably efficient, approach at the time of STP's development was translation of the decision problem to Boolean logic, and then invoking a SAT solver. The next step was to compensate for the weaknesses of this approach by preprocessing and abstraction-refinement based heuristics. Overall, this has worked very well for the theory of bit-vectors and arrays. However, it is an open question as to whether a SAT-based architecture as in the case of STP can be extended to build an efficient decision procedure for the union of a larger set of theories.

Preprocessing played a key role in compensating for the weaknesses of the SAT solver inside STP. The decision problem for the linear equational fragment of bit-vector arithmetic is polynomial time. This fact is exploited by the linear solver in STP, which preprocesses the inputs before they reach SAT. The solver helps get rid of redundant variables in the input. In general, we believe it is a good idea to consider preprocessing polytime fragments of the input theory.

Also as noted already, abstraction-refinement based heuristics have proven to be crucial in the case of STP. In this work we have taken a small step towards the use of abstraction-refinement approach in the context of arrays. Other tools such as UCLID have demonstrated the efficacy of the abstraction-refinement paradigm in the case of linear integer arithmetic, albeit using an abstraction method that is markedly different from the one used in STP. On the whole, we believe that this paradigm is bound to play a much more central role in the design of decision procedures in the future.

Finally, the differences in the kinds of formulas generated by hardware and software applications was a surprise. Traditionally, hardware application have been the prime users of decision procedures. They have tended to generate formulas which don't make much use of arrays, and are more arithmetic heavy. However, software program analyses tend to generate formulas that have a lot of array terms. The reason for this is that many sofware analyses model the memory of the program being analyzed, and do so using arrays.

5.3 Future Work

The work presented in the context of STP has been shown to be very effective in handling real-world formulas which are rich in linear bit-vector arithmetic and arrays. However, STP does not do any special preprocessing for disjunctions or non-linear bit-vector arithmetic. Also, STP does not support quantifiers. Another line of future work that seems promising is to extend STP with more theories like linear arithemtic and uninterpreted function symbols. There has been a strong demand for improving STP to effectively handle these additional requirements.

5.3.1 Disjunctions

In the context of the CVC and CVC Lite tools, some work has been done to handle disjunctions (more generally the Boolean structure) in the input formulas. More recently, the

DPLL(T) approach [GHN$^+$04] has proven to be quite effective in handling the Boolean structure of formulas over arbitrary theories. At a very high level, this approach seeks to exploit existing techniques from modern SAT solvers to handle the disjunctions in the input, while using the decision procedures for individual theories to handle the literals belonging to these theories, at the leaves of the Boolean tree of the input formula.

It is conceivable that a DPLL(T) approach can be used to handle disjunctions for designs such as that of STP. In order to achieve this STP would be required to identify a small *unsatisfiable core* or *certificate of unsatisfiability*, whenever an input formula is determined to be unsatisfiable. STP would have to interact with a DPLL(T) framework, while only accepting conjunction of literals. There is some evidence that such an approach would work efficiently with an architecture similar to STP. For example, the Z3 tool has a DPLL(T) framework, while simultaneously uses abstraction-refinement based heuristics for arrays similar to the ones in STP.

5.3.2 Non-linear Bit-vector Arithmetic

There has been very little work on efficient decision procedure for non-linear bit-vector arithmetic. There is clearly strong motivation to research this area. The most notable work in this direction has been done by Bryant, et al. [BKO$^+$07]. In this approach the decision procedure alternatively constructs under and over approximations of the input formula until the correct result is found. The hope is that the correct answer can be found quickly by deciding the abstractions as opposed to the computationally harder input formula.

One disadvantage of this approach is that it uses a SAT solver to decide the abstractions. SAT solvers may not be the most efficient for such a purpose. We hope to adapt this approach by using STP to check the abstractions instead of SAT solvers. Currently the abstractions are simply Boolean formulas. In a recent work, Lahiri and Mehra [LM07] have used interpolants to compute abstractions. It is conceivable that interpolants can be used to compute linear abstractions from non-linear formulas, and then these abstractions can be checked using STP rather than a SAT solver.

5.4 Additional Information

The STP system and some representative real-world examples can be downloaded from the website http://verify.stanford.edu/stp.html. The website also contains papers about STP and a list of applications where STP is being used successfully.

Bibliography

[ABHL97] Tod Amon, Gaetano Borriello, Taokuan Hu, and Jiwen Liu. Symbolic timing verification of timing diagrams using Presburger formulas. In *Design Automation Conference*, pages 226–231, 1997.

[ADK+05] Nina Amla, Xiaoqun Du, Andreas Kuehlmann, Robert P. Kurshan, and Kenneth L. McMillan. An analysis of SAT-based model checking techniques in an industrial environment. In Dominique Borrione and Wolfgang J. Paul, editors, *CHARME*, volume 3725 of *Lecture Notes in Computer Science*, pages 254–268. Springer, 2005.

[Aik06] Alex Aiken. Scalable program analysis using boolean satisfiability. In *4th ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE06)*, page 89, Napa, California, USA, July 2006. IEEE. KeyNote Speech.

[Bar03] Clark W. Barrett. *Checking Validity of Quantifier-Free Formulas in Combinations of First-Order Theories*. PhD thesis, Stanford University, January 2003. Stanford, California.

[BB04] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *CAV*, Lecture Notes in Computer Science. Springer, 2004.

[BBS+04] Clark Barrett, Sergey Berezin, Igor Shikanian, Marsha Chechik, Arie Gurfinkel, and David L. Dill. A practical approach to partial functions in CVC Lite. In *PDPAR'04 Workshop, Cork, Ireland*, July 2004.

[BC96] Alexandre Boudet and Hubert Comon. Diophantine equations, Presburger arithmetic and finite automata. In H. Kirchner, editor, *Colloquium on Trees in Algebra and Programming (CAAP'96)*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43. Springer Verlag, 1996.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.

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

[BD02] R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear programming. In *IEEE VLSI Design'01 & Asia and South Pacific Design Automation Conference, Bangalore*, pages 741–746, 2002.

[Bd07] Nikolaj Bjørner and Leonardo deMoura. System description: z3 0.1. Website: http://research.microsoft.com/z3/smtcomp07.pdf, July 2007. This system description was published as part of the SMTCOMP competition held at CAV 2007.

[BDL96] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods In Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, November 1996. Palo Alto, California, November 6–8.

[BDL98] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. A decision procedure for bit-vector arithmetic. In *Proceedings of the 35th Design Automation Conference*, June 1998. San Francisco, CA.

[BDS00] C. Barrett, D. Dill, and A. Stump. A Framework for Cooperating Decision Procedures. In David McAllester, editor, *17th International Conference on*

Computer Aided Deduction, volume 1831 of *LNAI*, pages 79–97. Springer-Verlag, 2000.

[BDS02a] C. Barrett, D. Dill, and A. Stump. A Generalization of Shostak's Method for Combining Decision Procedures. In *4th International Workshop on Frontiers of Combining Systems (FroCos)*, 2002.

[BDS02b] C. Barrett, D. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *14th International Conference on Computer-Aided Verification*, 2002.

[Ber02] Michael Berkelaar. LP_SOLVE. Website: http://www.cs.sunysb.edu/ algorith/implement/lpsolve/implement.shtml, 2002.

[BGD02] Sergey Berezin, Vijay Ganesh, and David L. Dill. Online proof-producing decision procedure for mixed-integer linear arithmetic. Technical Report CSTR 2007-07, Stanford University, Computer Science Department, 353 Serra Mall, Stanford, CA, USA 94305-9025, 2002.

[BGD03] Sergey Berezin, Vijay Ganesh, and David L. Dill. An online proof-producing decision procedure for mixed-integer linear arithmetic. In H. Garavel and J. Hatcliff, editors, *TACAS'03*, volume 2619 of *Lecture Notes in Computer Science*, pages 521–536, Warsaw Poland, April 2003. Springer Verlag. to appear.

[BHL$^+$07] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Towards automatically identifying trigger-based behavior in malware using symbolic execution and binary analysis. Technical Report CMU-CS-07-105, Carnegie Mellon University School of Computer Science, January 2007.

[BHZ06] Lucas Bordeaux, Youssef Hamadi, and Lintao Zhang. Propositional satisfiability and constraint programming: A comparative survey. *ACM Comput. Surv.*, 38(4), 2006.

[BKO^{+}07] Randal E. Bryant, Daniel Kroening, Joel Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. Deciding bit-vector arithmetic with abstraction. In *13th Intl. Conference on Tools and Algorithms for the Construction of Systems (TACAS)*, 2007.

[Ble75] W. W. Bledsoe. A new method for proving certain Presburger formulas. In *4th Joint International Conf. on Articial Intelligence*, pages 15–21, Tbilisi, Georgia, U.S.S.R, September 1975.

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

[BMS06a] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In E. Allen Emerson and Kedar S. Namjoshi, editors, *VMCAI*, volume 3855 of *Lecture Notes in Computer Science*, pages 427–442. Springer, 2006.

[BMS06b] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Verification, Model-Checking, and Abstract-Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 427–442. Springer-Verlag, 2006.

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

[BT97] Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, Belmont, Massachusetts, 1997.

[Büc60] J. R. Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für mathematische Logik und Grundladen der Mathematik*, 6:66–92, 1960.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CGP$^+$06] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, October-November 2006.

[CGP$^+$07] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, Dawson Engler, and David L. Dill. EXE: A system for automatically generating inputs of death using symbolic execution. *ACM Transactions on Information and System Security*, 2007.

[CKS05] Byron Cook, Daniel Kroening, and Natasha Sharygina. Cogent: Accurate theorem proving for program verification. In *17th International Conference on Computer Aided Verification (CAV)*, pages 296–300. Springer, 2005.

[CLR98] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, chapter 11, pages 820–825. MIT Press, 1998.

[CMR97] David Cyrluk, Michael Oliver Möller, and Herald Rueß. An efficient decision procedure for the theory of fixed-sized bit-vectors. In *9th International Conference on Computer Aided Verification (CAV)*, Lecture Notes in Computer Science, pages 60–71. Springer-Verlag, 1997.

[Coo72] D. C. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*, volume 7, pages 91–99, New York, 1972. American Elsevier.

[cpl] ILOG CPLEX. Website: http://www.ilog.com/products/cplex.

[DdM06] Bruno Dutertre and Leonardo de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of the 18th Computer-Aided Verification conference*, volume 4144 of *LNCS*, pages 81–94. Springer-Verlag, 2006.

[DE73] George B. Dantzig and B. Curtis Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.

[EKM98] Jacob Elgaard, Nils Klarlund, and Anders Mller. Mona 1.x: New techniques for WS1S and WS2S. In *Computer Aided Verification, CAV '98, Proceedings*, volume 1427 of *LNCS*. Springer Verlag, 1998.

[End72] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[ES03] Niklas Een and Niklas Sorensson. An extensible SAT-solver. In *Proc. Sixth International Conference on Theory and Applications of Satisfiability Testing*, pages 78–92, May 2003.

[Far90] W. M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55:1269–1291, 1990.

[GBD02] V. Ganesh, S. Berezin, and D. L. Dill. Deciding Presburger Arithmetic by Model Checking and Comparisons with Other Methods. In *Fourth International Conference on Formal Methods in Computer-Aided Design*, 2002.

[GBD07] Vijay Ganesh, Sergey Berezin, and David L. Dill. A decision procedure for fixed-width bit-vectors. Technical Report CSTR 2007-06, Stanford University, 353 Serra Mall, Computer Science Department, Stanford, CA, USA 94305-9025, 2007.

[GD07a] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV '07)*, Berlin, Germany, July 2007. Springer-Verlag.

[GD07b] Vijay Ganesh and David L. Dill. Website for STP: A decison procedure for bit-vectors and arrays. Website: http://verify.stanford.edu/stp.html, 2007.

[GHN$^+$04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification, CAV'04 (Boston, Massachusetts)*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004.

[GMP07] GMP (GNU Multiple Precision) library for arbitrary precision arithmetic. Website: http://www.gmplib.org, 2007. GNU Software from Free Software Foundation (FSF).

[HC01] C.Y. Huang and K.T. Cheng. Assertion checking by combined word-level atpg and modular arithmetic constraint-solving techniques. In *Design Automation Conference (DAC)*, pages 118–123, 2001.

[HG69a] Jo Ann Howell and Robert T. Gregory. An algorithm for solving linear algebraic equations using residue arithmetic I. *BIT Numerical Mathematics*, 9(3):200–224, September 1969.

[HG69b] Jo Ann Howell and Robert T. Gregory. An algorithm for solving linear algebraic equations using residue arithmetic II. *BIT Numerical Mathematics*, 9(4):324–337, December 1969.

[Hod72] Louis Hodes. Solving problems by formula manipulation in logic and linear inequalities. *Artificial Intelligence*, 3(1-3):165–174, 1972.

[Hod93] Wilfrid Hodges. *Model Theory*. Cambridge University Press, 1993.

[Hug93] R.I.G Hughes, editor. *A Philosophical Companion to First-Order Logic*. Hackett Publishing Company, 1993.

[ISO99] International Standards Organization. Website: http://www.iso.org, 1999.

[JD01] P. Johannsen and R. Drechsler. Formal verification on the RT level computing one-to-one design abstractions by signal-width reduction. In *IFIP International Conference on Very Large Scale Integration (VLSI'01), Montpellier, 2001*, pages 127–132, 2001.

[JGB97] P. Janicic, I. Green, and A. Bundy. A comparison of decision procedures in Presburger arithmetic, 1997.

[Kau06] Henry A. Kautz. Deconstructing planning as satisfiability. In *Proceedings of the Twenty-first Conference on Artificial Intelligence (AAAI-06)*, Boston, MA, USA, July 16-20 2006. AAAI Press.

[KK67] G. Kreisel and J. Krivine. Elements of mathematical logic, 1967.

[KMS96] Henry A. Kautz, David McAllester, and Bart Selman. Encoding plans in propositional logic. In *Proceedings of the Fifth International Conference on the Principle of Knowledge Representation and Reasoning (KR'96)*, pages 374–384, 1996.

[KOSS04] Daniel Kroening, Joel Ouaknine, Sanjit Seshia, and Ofer Strichman. Abstraction-based satisfiability solving of Presburger arithmetic. In *Intl. Conf. on Computer-Aided Verification (CAV)*, number 3114 in Lecture Notes in Computer Science, pages 308–320. Springer-Verlag, July 2004.

[KS92] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, 1992.

[LM07] Shuvendu K. Lahiri and Krishna K. Mehra. Interpolant based decision procedure for quantifier-free presburger arithmetic. *Journal of Satisfiability, Boolean Modeling and Computation (JSAT)*, 1:187–207, May 2007.

[LS04] Shuvendu K. Lahiri and Sanjit A. Seshia. The uclid decision procedure. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 475–478. Springer, 2004.

[Mal03] G. I. Malaschonok. Solution of systems of linear equations by the p-adic method. *Programming and Computer Software*, 29(2):59–71, 2003. Translated from original Russian journal titled Programmirovanie.

[McM93] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[MMZ$^+$01] M. Moskewicz, C. Madigan, Y. Zhaod, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *39th Design Automation Conference*, 2001.

[Möl98] M. Oliver Möller. Solving bit-vector equations - a decision procedure for hardware verification, 1998. Diploma Thesis, available at `http://www.informatik.uni-ulm.de/ki/Bitvector/`.

[MSS99] J. Marques-Silva and K. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[MSV07] Panagiotis Manolios, Sudarshan K. Srinivasan, and Daron Vroon. Bat: The bit-level analysis tool. In *Computer Aided Verification (CAV 2007)*. Springer, 2007.

[MWS07] David Molnar, David Wagner, and Sanjit A. Seshia. Catchconv : A tool for catching conversion errors. Personal Communications, 2007.

[NBFS06] James Newsome, David Brumley, Jason Franklin, and Dawn Song. Replayer: Automatic protocol replay by binary analysis. In *In the Proceedings of the $13th$ ACM Conference on Computer and and Communications Security (CCS)*, 2006.

[New67] M. Newman. Solving equations exactly. In *Journal of Research of the National Bureau of Standards*, volume 17B, pages 171–179, NIST, 100 Bureau Drive, Stop 1070, Gaithersburg, MD 20899-1070, USA, 1967. National Instiute of Standards and Technology (NIST).

[NO79] G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–57, 1979.

[NO80] G. Nelson and D. Oppen. Fast decision procedures based on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–64, 1980.

[Opp78a] Derek C. Oppen. A $2^{2^{2pn}}$ upper bound on the complexity of Presburger arithmetic. *Journal of Computer and System Sciences*, 16(3):323–332, June 1978.

[Opp78b] Derek C. Oppen. Reasoning about recursively defined data structures. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 151–157, New York, NY, USA, 1978. ACM Press.

[PBG05] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *STTT*, 7(2):156–173, 2005.

[Pre27] M. Presburger. Über de vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchen, die addition als einzige operation hervortritt. In *Comptes Rendus du Premier Congrès des Mathématicienes des Pays Slaves*, pages 92–101, 395, Warsaw, 1927.

[Pug91] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[RS01] H. Rueß and N. Shankar. Deconstructing Shostak. In *16th IEEE Symposium on Logic in Computer Science (LICS)*, pages 19–28, Boston, MA, USA, 2001.

[RT06] Silvio Ranise and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2006.

[SBD02] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A cooperating validity checker. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 500–504, London, UK, 2002. Springer-Verlag.

[SBDL01] A. Stump, C. Barrett, D. Dill, and J. Levitt. A Decision Procedure for an Extensional Theory of Arrays. In *16th IEEE Symposium on Logic in Computer Science*, pages 29–37. IEEE Computer Society, 2001.

[Ses05] Sanjit A. Seshia. *Adaptive Eager Boolean Encoding for Arithmetic Reasoning in Verification*. PhD thesis, Carnegie Mellon University, 2005.

[Sho77] Robert E. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the ACM*, 24(4):529–543, 1977.

[Sho84] R. Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, 1984.

[SKR98] T. R. Shiple, J. H. Kukula, and R. K. Ranjan. A comparison of Presburger engines for EFSM reachability. In A. J. Hu and M. Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer Aided Verification*, volume 1427, pages 280–292. Springer-Verlag, 1998.

[SS96] J. P. M. Silva and K. A. Sakallah. GRASP – A new search algorithm for satisfiability. In *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design*, pages 220–227, 11 1996.

[ST67] N.S. Szabo and R.I. Tanaka. *Residue Arithmetic and Its Applications to Computer Technology*. McGraw Hill, New York, 1967.

[Stu02] Aaron Stump. *Checking Validities and Proofs with CVC and flea*. PhD thesis, Stanford University, 2002.

[TH96] C. Tinelli and M. Harandi. A New Correctness Proof of the Nelson-Oppen Combination Procedure. In F. Baader and K. Schulz, editors, *1st International Workshop on Frontiers of Combining Systems (FroCoS'96)*, volume 3 of *Applied Logic Series*. Kluwer, 1996.

[WB00] Pierre Wolper and Bernard Boigelot. On the construction of automata from linear arithmetic constraints. In *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, March 2000. Springer-Verlag.

[Wel99] Daniel S. Weld. Recent advances in AI planning. *AI Magazine*, 20(2):93–123, 1999.

[Wil76] H. P. Williams. Fourier-Motzkin elimination extension to integer programming problems. *Journal of Combinatorial Theory (A)*, 21:118–123, 1976.

[XA07] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3):16, 2007.