# Adaptive Eager Boolean Encoding for Arithmetic Reasoning in Verification

Sanjit A. Seshia

May 2005

CMU-CS-05-134

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.*

**Thesis Committee:**
Prof. Randal E. Bryant, Chair
Prof. Edmund M. Clarke
Prof. Jeannette M. Wing
Prof. David L. Dill, Stanford University



This research was sponsored in part by a National Defense Science and Engineering Graduate Fellowship, the National Science Foundation under grant CCR-9805366, and the U.S. Army under ARO grant DAAD19-01-1-0485.

**Keywords:** Decision procedures, automated theorem proving, model checking, Boolean satisfiability, integer linear programming, quantified Boolean formulas, first-order logic, timed automata, difference constraints, timed circuits, infinite-state systems, software security, machine learning, verification, reliability, security, UCLID, TMV.

*Dedicated to Appa, Amma, Ashwin, and Sunny*

## Abstract

Decision procedures for first-order logics are widely applicable in design verification and static program analysis. However, existing procedures rarely scale to large systems, especially for verifying properties that depend on data or timing, in addition to control.

This thesis presents a new approach for building efficient, automated decision procedures for first-order logics involving arithmetic. In this approach, decision problems involving arithmetic are transformed to problems in the Boolean domain, such as Boolean satisfiability solving, thereby leveraging recent advances in that area. The transformation automatically detects and exploits problem structure based on new theoretical results and machine learning. The results of experimental evaluations show that our decision procedures can outperform other state-of-the-art procedures by several orders of magnitude.

The decision procedures form the computational engines for two verification systems, UCLID and TMV. These systems have been applied to problems in computer security, electronic design automation, and software engineering that require efficient and precise analysis of system functionality and timing. This thesis describes two such applications: finding format-string exploits in software, and verifying circuits that operate under timing assumptions.

# Contents

# List of Figures

# List of Tables

# Acknowledgments

I would like to thank my advisor, Randal E. Bryant, for his exemplary guidance, strong support, and timely and sound advice. His vision, depth and breadth of knowledge, and keen insight has facilitated my growth as a researcher. He has allowed me much freedom to explore many different ideas and projects throughout my graduate education and has whole-heartedly encouraged all of my aspirations. I have been extremely fortunate to have had the opportunity to work with him.

I thank Jeannette M. Wing for her constant support and encouragement over the years, and for her incisive feedback on my thesis work. Her class on computer security instilled in me a deep interest and sound foundation in the area. I have benefited much from her advice over the years on all aspects of research and graduate education. I owe a special thanks to Edmund M. Clarke, who introduced me to the fields of model checking and program analysis, and has always provided me with critical insights, valuable counsel, and whole-hearted support on my research and career goals. I am grateful to David L. Dill for serving on my dissertation committee, and for his insightful comments and questions.

I have had the privilege of working with several faculty, both at and outside CMU, and with researchers from industry. I am grateful to Somesh Jha for his advice over the years and for sharing his expertise in computer security with me. The work on software security described in this dissertation is a result of a collaboration with Somesh and Thomas W. Reps. I would also like to thank Kenneth L. McMillan, with whom I did an internship and have had many fruitful discussions. Shaz Qadeer gave me my start in software verification, along with much valuable advice; I am fortunate to have worked closely with him. I thank Kenneth S. Stevens for providing expert guidance on navigating the field of asynchronous circuit design, and giving freely of his time and knowledge. I have learnt a great deal about computer security from Dawn Song. I have also benefited greatly from the classes I have taken and the projects I have undertaken with many CMU faculty, both within and outside SCS.

My thesis work has also benefited greatly from collaborations with several graduate students, post-doctoral fellows, and visiting researchers, including Mihai Christodorescu, Vinod Ganapathy, Daniel Kroening, Shuvendu Lahiri, Joël Ouaknine, Ofer Strichman, and K. Subramani. Shuvendu and I worked very closely on an early version of UCLID, writing many joint papers and having several fun technical discussions. This thesis has also benefited a lot from the many insightful discussions I have had over the years with Joël and Ofer.

I am grateful to Ignacio Grossmann and his group in the Chemical Engineering department for graciously sharing their copy of CPLEX with me. I would like to acknowledge Sergey Berezin, Louis Latour, Chris Myers, and Leonardo de Moura for their assistance with the tools they support with which I have made comparisons in this thesis. I thank Sagar Chaki, Michael Ernst, Ranjit Jhala, Rupak Majumdar, and Stephen McCamant for providing me with benchmarks. I am also grateful for the feedback received from the early users of UCLID at other universities, including Karem Sakallah and Zaher Andraus at Michigan, Ganesh Gopalakrishnan and his students at Utah, and Panagiotis Manolios and his students at GeorgiaTech.

I have been lucky to share offices (and laughter) with many wonderful people over the years, including Henry Rowley, Bwolen Yang, James Thomas, Caitlin Kelleher, Dennis Strelow, Robert Miller, Bartosz Przydatek, Peter Richter, Varun Gupta, and Todd Phillips. I am grateful for the counsel and friendship of Mihai Budiu, Pankaj Chauhan, Amit Goel, Anubhav Gupta, Karthik Kannan, Sanjay Rao, and Weng-Keen Wong.

Over the years, the administrative support provided by Rosemary Battenfelder, Sharon Burks, Cynthia Chemsak, Catherine Copetas, Joan Maddamma, Karen Olack, and Cleah Schlueter has made my stay at CMU a delightful and productive experience, and I am grateful for their help.

I thank my extended family in the U.S. and India for their love, advice, and encouragement over the years.

Finally and foremost, I would like to thank my parents, my brother Ashwin, and Sunny, our inspiration; their love has always been and continues to be my pillar of strength.

# Chapter 1

# Introduction

Our increasing reliance on computer systems places an ever greater need for ensuring that they perform as expected. Errors in system design and implementation as well as malicious attacks pose a major barrier to exploiting the benefits of computing, creating problems ranging from lagging productivity to dangerous vulnerabilities in safety-critical systems. According to a recent survey [117], the cost of software bugs to the U.S. economy is of the order of 60 billion dollars every year, underlining the costs of failure in computer systems.

Errors can be found at various stages in a system's lifetime, ranging from design-time, through compile-time, to run-time, and even post-mortem. It is preferable to find errors as early as possible, as the costs of failure in deployed systems, particularly in unsupervised, safety-critical settings, can be enormous. Techniques for formal design verification and static program analysis are targeted towards improving the reliability and security of systems before run-time. The input to every such technique comprises a system description and a specification, and outputs a yes/no answer as to whether the system satisfies its specification (and possibly "don't know" in some cases). The scalability of these techniques depends on that of the computational engines, or *decision procedures*, that underlie them. These decision procedures analyze a formal model, usually expressed in mathematical logic, to provide the yes/no answer.

Decision procedures for decidable fragments of first-order logic have found use in analyzing many kinds of systems, including application and system software, gate-level circuit designs, hybrid systems, and high-level microprocessor designs. For example, decision procedures play important roles in extended static checking [55], predicate abstraction-based software verification (e.g., [11, 36, 69]), finite-state model checking (e.g., [33, 41]), model checking timed systems (e.g., [71]), and processor verification (e.g., [29, 34]). Of these applications, the previous industrial-scale applications have been largely restricted to analyzing systems with Boolean state (such as finite-state systems or pushdown systems) or techniques that generate Boolean abstractions. These successes have been driven in large part by the efficiency of techniques for reasoning about and manipulating Boolean functions, such as Binary Decision Diagrams (BDDs) [27] and Boolean satisfiability (SAT) solvers (e.g., [63, 104]). In this thesis, these techniques are collectively referred to as *Boolean methods*.

The efficiency benefits of modeling systems purely with Boolean state are counterbalanced by a loss of modeling precision. Reduced precision results in false alarms, and the inability to verify properties depending heavily on data and timing, in addition to control. The successes of finite-state model checking and predicate abstraction-based software analysis have been restricted to analyzing control properties, such as verifying cache-coherence protocols and checking device driver usage protocols. Examples of analyses that require more precise modeling of data and timing include detection of malicious code (such as viruses or worms), high-level microprocessor design verification, array-bounds checking and buffer overrun detection, and verifying real-time systems and timed circuits. In these tasks, a rich set of non-Boolean data-types must often be modeled, including finite- and arbitrary-precision integers, real and floating-point numbers, memories, arrays, and data structures such as queues or lists. The resulting decision problems are only expressible in first-order logics or sometimes even only in higher-order logics. Previous decision procedures for these more expressive logics have rarely scaled to industrial-scale systems without some form of manual assistance.

This thesis presents a new approach to building efficient, automated decision procedures for first-order logics involving arithmetic based on Boolean methods. The practicality of this approach is demonstrated by incorporating it in verification tools that have been successfully applied to industrial-scale hardware and software systems. There are two key ideas in this approach.

1. *Leverage Boolean methods:* The decision procedures presented in this thesis operate by performing a Boolean encoding of the decision problem, either as a Boolean satisfiability (SAT) problem, or a problem involving manipulation of quantified Boolean formulas (QBF). Moreover, the encoding is *eager*, meaning that it is done in a single step. This enables us to easily leverage recent dramatic advances in Boolean methods.

2. *Use adaptive encoding:* The Boolean encoding algorithms are *adaptive*, meaning that an encoding algorithm or its parameters are automatically chosen based on the structure of its input. This is achieved by a combination of theoretical results on formalizing problem structure and the application of machine learning to inputs encountered in the past. The use of adaptive Boolean encoding enables us to solve the resulting Boolean problems more efficiently, in many cases by orders of magnitude compared to previous approaches.

The decision procedures proposed in this thesis form the computational engines for two verification systems, UCLID and TMV. These systems have been applied to a variety of application areas; the ones explored in this thesis are software security and the verification of timed circuits.

## 1.1 Boolean Encoding Techniques

We review and classify previous work on decision procedures based on Boolean methods so as to place the contributions of this thesis in context. Detailed surveys of previous work on specific topics, including application areas, are included in the corresponding chapters.

Decision procedures based on Boolean encoding methods fall into two main categories:

1. *Eager encoding methods:* Decision procedures in this class perform the Boolean encoding in a single step. For the quantifier-free logics considered in this thesis, the input formula is translated to an equi-satisfiable Boolean formula in a single step, and a SAT solver is invoked on the result. For the quantified logic considered, the translation generates a logically equivalent quantified Boolean formula (QBF), which can be manipulated using well-known techniques for QBF based on BDDs or SAT.

   These methods have been developed for the theories of uninterpreted functions and equality [29, 122], a restricted set of lambda expressions (which can model arrays, memories, and some data structures) [30], and various theories of linear arithmetic over the integers and the rationals [30, 146, 148], with very limited support for quantifiers [89]. Counterexamples at the level of the original logical theories are easily generated, by mapping back from assignments generated by the SAT solver.

   Eager encoding techniques can be further divided into two kinds. The first kind [28, 30, 122] exploit a *small model property* of the underlying theory; i.e., if a satisfying assignment exists for the original formula, then there is one in which the values of ground terms are bounded. This naturally leads to a bit-vector encoding of the ground terms. The second class of techniques [32, 62, 148] are *direct* encoding techniques, in which each atomic predicate is encoded as a Boolean variable. The resulting Boolean encoding is augmented with the Boolean encoding of instantiations of first-order axioms, such as congruence and transitivity of equality, over the ground terms in the formula.

2. *Lazy encoding methods:* Procedures in this category (e.g., [8, 13, 51, 56]) construct the Boolean encoding iteratively. Provers based on these methods, such as CVC and CVC-Lite [13, 14], ICS [51], and VeriFun [56], are designed to handle a fairly general class of first-order logic; in addition to the theories handled by the afore-mentioned eager techniques, these provers can handle a subset of the theories of bit-vectors, lists, and records, and some also provide support for quantifiers. Another advantage of these methods is that they are typically designed to be proof-generating.

   The lazy encoding procedures work, in essence, as follows. They start with a direct Boolean encoding of the original formula, obtained by replacing each atomic predicate with a corresponding Boolean variable. If the SAT solver returns this formula to be unsatisfiable, it means that the original formula is also unsatisfiable. Otherwise, the SAT solver returns a satisfying assignment, which must be checked for consistency with the first-order theories. This is performed using a first-order prover for checking the satisfiability of conjunctions, also known as a *ground decision procedure*. If the assignment is consistent, then it implies that the original formula is satisfiable. Otherwise, the proof of unsatisfiability generated by the ground decision procedure is analyzed to generate additional clauses that are added to the Boolean encoding to constrain the search of the SAT solver, and the process repeats.

The differences between the various provers based on lazy encoding methods are mainly with respect to the tightness of integration between the SAT solver and the ground decision procedures, and the details of the ground decision procedures themselves. The ground decision procedures are generally based on a technique for combining decision procedures for individual theories, such as that given by Nelson and Oppen [109] or Shostak [141].

The lazy encoding approach has also been applied to quantifier-elimination in decidable quantified first-order logics [52].

The decision procedures proposed in this thesis fall into the first category. The quantifier-free logic considered in this thesis is a combination of the theories of uninterpreted functions and equality, quantifier-free Presburger arithmetic [125], and the restricted set of lambda expressions mentioned above (described in Chapter 7). In addition, this thesis presents the first eager encoding approach to performing quantifier-elimination in quantified difference logic (described in Chapter 8).

Let us compare lazy and eager encoding methods for the quantifier-free fragment of first-order logic considered in this thesis.

Eager encoding methods have the advantage that the resulting SAT problem has all the "first-order information" necessary to constrain the SAT solver's search, whereas adding this information lazily might cause the SAT solver to explore many assignments that are inconsistent with the first-order theories (exponentially many in the worst-case). Also, with eager methods, it is trivial to replace one SAT solver with another, and thus readily leverage any advances in SAT solving; this can be far harder in lazy techniques depending on how tightly the SAT solver is integrated into the decision procedure.

On the other hand, it is also possible for eager encoding algorithms to add too much "first-order information," generating SAT problems beyond the reach of current SAT solvers. Lazy methods are particularly effective when very little first-order reasoning is required (for example, when propositional reasoning suffices to decide unsatisfiability). Furthermore, many lazy methods are proof-generating, which is useful for certified verification (such as proof-carrying code [106]) as well as for abstraction refinement [67]. It is not yet clear how to generate proofs with eager encoding methods, especially those based on the small-domain encoding.

In the rest of this thesis, we will compare eager and lazy methods for specific theories via experimental evaluation.

## 1.2 Thesis Contributions

My thesis statement is:

> Adaptive Boolean encoding methods enable the construction of efficient and automated decision procedures for expressive first-order logics involving arithmetic, increasing the precision and scalability of verification tools for hardware and software systems.

This thesis makes contributions in a number of areas. The main theoretical and conceptual contributions include:

- The first decision procedure for quantifier-free Presburger arithmetic that is based on a polynomial-time, polynomial-size translation to SAT, and which formally exploits the structure of linear constraints in software analysis (Chapter 5);
- New theoretical results on bounding the size of solutions for generalized 2SAT constraints and quantifier-free Presburger arithmetic (Chapters 4 and 5);
- The first approach to automated algorithm selection in a theorem proving context, based on the use of machine learning (Chapter 6);
- The first eager encoding approach for quantifier elimination in quantified difference logic (Chapter 8);
- The notion of *generalized relative timing* for modeling timing assumptions in circuits (Chapter 9).

There are also several applied contributions, including tools and industrial case studies:

- A publicly-available, multipurpose verification tool, called UCLID, for verifying systems modeled using the quantifier-free fragment of first-order logic mentioned earlier, with demonstrated applications in processor verification and software security (Chapter 7);
- The application of UCLID to finding a class of security exploits called *format-string exploits*, demonstrated on widely-used software packages (Chapter 7);
- A fully symbolic model checker, called TMV, for model checking timed automata (Chapter 9);
- The application of TMV to the verification of timed circuits, including a published circuit of the Pentium 4 microprocessor (Chapter 9).

## 1.3 Thesis Overview

This thesis covers a wide range of areas, spanning theory, hardware, and software. Accordingly, the thesis is organized into three parts, including one on background material, so that the content of each main part of the thesis is fairly independent of that of the other.

The first part of the thesis, comprising Chapters 2 and 3, gives background material needed in the rest of the thesis. Chapter 2 covers basic notation and linear programming concepts. Chapter 3 describes *difference logic*, a basic logic that forms the foundation for the concepts in this thesis, and two Boolean encoding algorithms: the *small-domain* encoding and the *direct* encoding algorithm. The material in Chapter 3 is based on joint work with R. E. Bryant, S. K. Lahiri, and O. Strichman [30, 148].

The second part of the thesis (Part I) presents our new decision procedures for linear arithmetic over the integers, extensions to handle other theories, and the implementation and application of the UCLID system. Chapter 4 describes how the small-domain Boolean encoding method can be extended to a logic of generalized 2SAT linear constraints, and is based on joint work with R. E. Bryant and K. Subramani [138]. Chapter 5 shows how the same class of encoding algorithms can be extended to quantifier-free Presburger arithmetic, by exploiting the sparse structure of linear constraints in software analysis; this is joint work with R. E. Bryant [135]. Chapter 6 compares the two encoding algorithms for difference logic and shows how they can be combined using machine learning to automatically select encodings for sub-formulae. A very preliminary version of the work in this chapter appeared in a joint paper with R. E. Bryant and S. K. Lahiri [133], and the material in this chapter is a substantial revision of that work. Part I is closed by Chapter 7, which describes how theories other than integer linear arithmetic are encoded to SAT, along with a description of the UCLID verification system and an application of UCLID to finding format-string exploits. The initial part of this chapter is based on joint work with R. E. Bryant and S. K. Lahiri [30]. The application to format-string exploits is based on joint work with R. E. Bryant, V. Ganapathy, S. Jha, and T. W. Reps [58]; in particular, the idea of viewing the format-string as a sequence of commands to `printf` is due to my co-authors Ganapathy, Jha, and Reps.

The third part of this thesis (Part II) describes how operations in quantified difference logic can be handled using Boolean methods, and describes an application to model checking timed circuits. Chapter 8 describes the operations on quantified difference logic (QDL), and is based on joint work with R. E. Bryant [134]. Although the content of this chapter is used for model checking timed systems, other applications are possible, and the material is fairly independent of the application explored in this thesis. Chapter 9 describes how we use the QDL operations in TMV, a model checker for timed automata, and the application of TMV to the verification of timed circuits. This chapter also describes the notion of generalized relative timing, which is a new technique for modeling timing assumptions in systems. The material in this chapter is based on joint papers with R. E. Bryant and K. S. Stevens [134, 136].

Finally, Chapter 10 summarizes the major conceptual contributions and design decisions in this thesis, and proposes several directions for future work.

# Chapter 2

# Preliminaries

We introduce notation and concepts from linear programming that are useful in the rest of this thesis. Standard textbooks (e.g. [111, 131]) can be consulted for additional information.

## 2.1 Notation

We will use $m$ to denote the number of linear constraints, and $n$ to denote the number of variables.

A system of $m$ linear constraints in $n$ variables is written as follows:

$$A\,\mathbf{x} \geq \mathbf{b} \tag{2.1}$$

In general, $A = [a_{i,j}]$ is an $m \times n$ matrix with entries in $\mathbb{R}$, $\mathbf{b}$ is a $m \times 1$ vector of real-valued entries, and $\mathbf{x}$ is a $n \times 1$ vector of real-valued variables.

System (2.1) defines a polyhedron in $\mathbb{R}^n$ formed by the intersection of half-spaces corresponding to the linear constraints.

For Part I of this thesis, we will only consider integer variables and constants; that is, for all $i$ and $j$, $a_{i,j}, b_i, x_j \in \mathbb{Z}$.

In system (2.1), the entries in $\mathbf{x}$ can be negative. A standard transformation (see, e.g., [119]) can be used to constrain the variables to be non-negative. The transformation involves adding a dummy variable $x_0$ that refers to the "zero value," replacing each original variable $x_i$ by $x'_i - x_0$, and then adjusting the coefficients in the matrix $A$ to get a new constraint matrix $A'$ and the following system:

$$\begin{aligned} A'\mathbf{x}' &\geq \mathbf{b} \\ \mathbf{x}' &\geq 0 \end{aligned} \tag{2.2}$$

Note that $x_0$ is an element of the vector $\mathbf{x}'$ of dimension $n+1$. Matrix $A'$ has dimensions $m \times n+1$, where the last column corresponds to $x_0$. The $(i,j)^{\text{th}}$ entry of $A'$ is the same as that for $A$ for $1 \leq j \leq n$, and $a'_{i,n+1}$ is $-\sum_{j=1}^{n} a_{i,j}$.

The transformation from system (2.1) to system (2.2) preserves satisfiability, as shown here:

**Proposition 2.1** *System (2.1) has a solution if and only if system (2.2) has one.*

*Proof:* For the "if part", suppose we have a solution $\mathbf{x}'$ to (2.2). Construct a candidate solution vector $\mathbf{x}$ by setting $x_j = x'_j - x_0$. Then, consider the $i^{\text{th}}$ constraint in $A'$, for any $i$. The following sequence of inequalities holds:

$$\begin{aligned}
(\sum_{j=1}^{n} a'_{i,j} x'_j) + a'_{i,n+1} x_0 &\geq b_i \\
(\sum_{j=1}^{n} a_{i,j} x'_j) + (-\sum_{j=1}^{n} a_{i,j}) x_0 &\geq b_i \\
\sum_{j=1}^{n} a_{i,j} (x'_j - x_0) &\geq b_i \\
\sum_{j=1}^{n} a_{i,j} x_j &\geq b_i
\end{aligned}$$

Thus, we can conclude that the $i^{\text{th}}$ constraint of $A$ is satisfied by $\mathbf{x}$ for all $i$. Thus, we have found a solution to system (2.1).

Now consider the "only if" part, where we start with a solution to system (2.1). Clearly, any value of $\mathbf{x}'$ that sets $x'_j = x_j + x_0$ for all $j$ will satisfy $A'\mathbf{x}' \geq \mathbf{b}$. But we also need to satisfy $\mathbf{x}' \geq 0$. If none of the $x_j$ are negative, then simply set $x'_j = x_j$ and $x_0 = 0$ and we are done. Otherwise, set $x_0 = -\min_{k, x_k < 0} x_k$, and set $x'_j = x_j + x_0$. Note that $x_0 > 0$ by construction. Thus, if for a particular $j$, $x_j > 0$, then $x'_j > 0$. Suppose not. Then, $x_j \geq \min_{k, x_k < 0} x_k$ and so $x'_j = x_j - \min_{k, x_k < 0} x_k \geq 0$. Thus, we have a solution $\mathbf{x}'$ that satisfies (2.2). $\square$

Finally, we define the quantities $a_{\max}$ and $b_{\max}$ as follows:

$$a_{\max} = \max_{(i,j)} |a_{i,j}| \tag{2.3}$$

$$b_{\max} = \max_{t} |b_t| \tag{2.4}$$

In words, the quantity $b_{\max}$ is the $L_\infty$ norm of the vector $\mathbf{b}$. We note that $a_{\max}$ and $b_{\max}$ are (tight) upper bounds on the absolute values of entries of $A$ and $\mathbf{b}$ respectively.

## 2.2 Variable Classes

Given a set of $m$ linear constraints over $n$ variables, the set of variables can be partitioned into subsets as follows: Two variables are placed in the same subset if there is a constraint in which they both appear with non-zero coefficients. We will refer each resulting subset as a *variable class*.

If the set of constraints appears in a system of constraints, like system 2.1, then partitioning variables into variable classes corresponds to partitioning the system into sub-systems that can be solved independently of each other. (In matrix terms, the matrix $A$ is transformed to block-diagonal form.) Importantly, note that this partitioning optimization can be performed *before* adding the "zero" variable $x_0$. A different zero variable is then used for each variable class.

The notion of variable classes can be extended to Boolean combinations of linear constraints by applying it to the set of all linear constraints appearing in the formula. For example, consider the formula

$$x_1 + x_2 \geq 1 \ \wedge \ \big(x_2 - x_3 \geq 0 \ \vee \ x_4 - x_5 \geq 0\big)$$

In this case, variables $x_1$, $x_2$, and $x_3$ fall into one class, while $x_4$ and $x_5$ will be put into a different class.

## 2.3 Fourier-Motzkin Elimination

Fourier-Motzkin (FM) elimination [49] is a classic technique for projecting a variable from a set of linear constraints.

Consider system (2.1). In order to obtain the system of linear constraints after projecting out variable $x_j$, FM elimination proceeds as follows:

1. Partition the system of constraints into three sets $P_j$, $N_j$, $Z_j$ as follows. For each constraint $i$, $1 \leq i \leq m$, we add it to:

$$\begin{array}{ll} P_j, & \text{if } a_{i,j} > 0; \\ N_j, & \text{if } a_{i,j} < 0; \\ Z_j, & \text{otherwise.} \end{array}$$

2. Initialize the set of new constraints, $\Phi$, to $Z_j$.

3. For every pair of constraints $(i_P,\, i_N)$, where $i_P \in P_j$ and $i_N \in N_j$, add the following constraint to $\Phi$:

$$\sum_{k=1}^{n} \big(a_{i_P,j} \cdot a_{i_N,k} - a_{i_P,k} \cdot a_{i_N,j}\big) \cdot x_k \geq b_i$$

Clearly, the coefficient of $x_j$ in every constraint in $\Phi$ is $0$.

If $\mathbf{x} \in \mathbb{R}^n$, this transformation preserves satisfiability. In other words, there is a solution to the system of constraints in $\Phi$ if and only if there is one to the system (2.1). Thus, by using FM elimination to project out all variables, we can conclude that the original system is satisfiable if and only if the system with zero variables does not have a trivially false constraint (such as $0 \geq 1$).

In the worst case, the number of new constraints generated by $n$ steps of FM elimination can be $m^{2^n}$, i.e., doubly exponential in the input size [37].

# Chapter 3

# Difference Logic

A simple but extremely useful form of linear constraint is the *difference constraint*. This chapter presents Boolean encoding techniques for a logic of difference constraints, termed as *difference logic*. These encoding techniques form the basis for many ideas in the rest of this thesis.

**Definition 3.1** *A difference constraint is a linear constraint of the form $x_i - x_j \bowtie b_t$ or $x_i \bowtie b_t$, where $x_i$ and $x_j$ are real-valued variables, $b_t$ is a real-valued constant, and $\bowtie$ denotes a relational symbol in the set* $\{>, \geq, =, <, \leq\}$.

A constraint of the form $x_i \bowtie b_t$ can be written as $x_i - x_0 \bowtie b_t$ where $x_0$ is a special "variable" denoting zero. This convention is followed in the rest of the thesis, unless stated otherwise.

Difference constraints are also referred to in the literature as *difference-bound constraints* or *separation predicates*, and difference logic is also commonly termed as *separation logic*. We will use DL as an acronym for difference logic.

$$
\begin{array}{rcl}
\textit{bool-expr} & ::= & \mathbf{true} \mid \mathbf{false} \mid \textit{bool-var} \mid \neg \textit{bool-expr} \\
 & & \mid (\textit{bool-expr} \wedge \textit{bool-expr}) \mid (\textit{num-expr} \geq \textit{num-expr}) \\
\textit{num-expr} & ::= & x_i \mid \textit{num-expr} + b \mid \textit{ITE}(\textit{bool-expr}, \textit{num-expr}, \textit{num-expr})
\end{array}
$$

Figure 3.1: **Difference logic syntax.** $x_i$, $0 \leq i \leq n$, and $b$ denote a variable and constant respectively.

Figure 3.1 summarizes the expression syntax for difference logic. Expressions can be of two types: numerical or Boolean. Boolean expressions are formed by using Boolean connectives to combine equalities, inequalities, or Boolean variables. Numerical expressions are either numerical (integer or real) variables, or are formed by adding a constant offset to numerical expressions, or by applying the *ITE* ("if-then-else") operator. The *ITE* operator chooses between two values based on a Boolean control value, i.e., *ITE*$(\mathtt{true},\, x_1,\, x_2)$ yields $x_1$ while *ITE*$(\mathtt{false},\, x_1,\, x_2)$ yields $x_2$. Boolean and relational operators not used in Figure 3.1 can be expressed in terms of those employed.

### Remarks on notation

Note that the grammar in Figure 3.1 permits real and integer variables to be mixed in relational comparisons and if-then-else expressions. For the purposes of this thesis, we will consider either only integer variables, or only real variables, depending on the context. For the remainder of this chapter, we will restrict all variables and constants to be integer-valued.

Second, as noted in Chapter 2, multiple zero variables will usually be introduced, one for each variable class. In the rest of this chapter, we will assume that these variables have already been introduced into the DL formula, so that every difference constraint comprises exactly two variables, each taking values in $\mathbb{N}$.

Finally, although the syntax permits us to write expressions of the form *num-expr* $\geq$ *num-expr*, we will use notation in which only variables appear only on the left-hand side, and no more than one constant term appears on the right-hand side. Thus, a difference constraint will usually be written either as $x_i - x_j \geq b_t$ or as $x_i \geq x_j + b_t$.

### Complexity of the decision problem

The problem of deciding the satisfiability of a DL formula $F_{\mathit{diff}}$ over the integers is NP complete. It is NP-hard since Boolean satisfiability can be trivially reduced to it. In addition, it is in NP because the logic has a *small-model property*: A DL formula $F_{\mathit{diff}}$ is satisfiable if and only if there exists a satisfying assignment whose size, measured in bits, is polynomially bounded in the size of $F_{\mathit{diff}}$. A proof of the latter property is presented in Section 3.2.

However, if we restrict the syntax of DL by disallowing Boolean variables, *ITE* expressions, and all Boolean connectives except $\wedge$, the satisfiability problem is polynomial-time solvable. This restricted problem is simply that of finding a feasible solution to a system of difference constraints, and can be solved using a formulation as a shortest-path problem [43].

### Overview

In this chapter, we present two approaches to deciding difference logic via eager encoding to SAT:

1. **Small-Domain Encoding** [30]: This approach exploits the small-model property of DL, and works as follows:

   (a) Compute the polynomial bound $S$ on solution size.

   (b) Search for a satisfying solution to $F_{\mathit{diff}}$ in the bounded space $\{0, 1, \ldots, 2^S - 1\}^n$.

   The small-domain encoding approach is also termed as *finite instantiation*.

   In the methods described in this thesis, the search in Step (b) is conducted using a SAT solver. To do this, $F_{\mathit{diff}}$ is translated to a Boolean formula by encoding each integer variable as a vector of Boolean variables of length $S$. Arithmetic and relational operators are encoded as arithmetic circuits and comparators.

   However, note that a non-SAT-based search technique can just as well be used.

2. **Direct Encoding** [148]: A decision procedure based on the direct encoding method operates in $4$ steps:

   (a) Eliminate the *ITE* construct from the formula, to get a formula that is a Boolean combination of difference constraints.

   (b) Replace each unique difference constraint with a fresh Boolean variable to get a Boolean formula $F_{bvar}$.

   (c) Generate a Boolean formula $F_{arith}$ that constrains the values of the introduced Boolean variables so as to preserve the arithmetic information in the original formula.

   (d) Decide the satisfiability of Boolean formula $F_{bvar} \wedge F_{arith}$ using a SAT solver.

   The direct encoding approach has also been termed as *per-constraint encoding*.

At the time of writing this thesis, all decision procedures based on eager encoding to SAT can be viewed as instances of one of the above two methods.

## 3.1 Constraint Graph

We begin by describing a basic data structure used in the rest of this chapter.

Given a set of $m$ difference constraints involving $n$ variables, we construct a weighted, directed multigraph as follows:

1. A vertex $v_i$ is introduced for each variable $x_i$.

2. For each difference constraint of the form $x_i - x_j \geq b_t$, we add a directed edge from $v_i$ to $v_j$ of weight $b_t$.

The resulting structure has $m$ edges and $n$ vertices and is termed the *constraint graph*. It is, in general, a multigraph since there can be multiple constant (right-hand side) terms for a given left-hand side expression $x_i - x_j$. However, we will refer to it simply as a graph.

**Example 3.1** Consider the following set of 8 constraints involving 7 variables:

$$
\begin{array}{rclcrcl}
x_1 - x_2 & \geq & 0 & \quad & x_5 - x_6 & \geq & 50 \\
x_2 - x_3 & \geq & 0 & & x_6 - x_4 & \geq & -100 \\
x_3 - x_1 & \geq & 1 & & x_6 - x_5 & \geq & -49 \\
x_4 - x_5 & \geq & 100 & & x_7 - x_4 & \geq & -100
\end{array}
$$

The constraint graph representing the above set of constraints is depicted in Figure 3.2 □

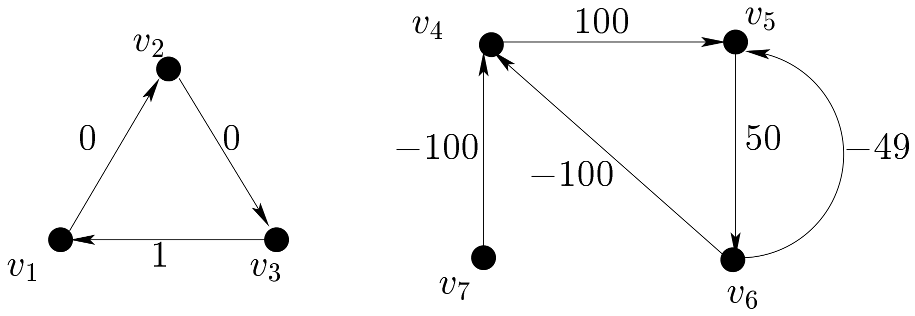



Figure 3.2: **Example of constraint graph**

## 3.2 Small-Domain Encoding

The crucial piece of information needed to implement the small-domain encoding method is the bound on solution size, $S$.

In this section, we obtain a bound $d$ on the *values* of variables in a DL formula such that it is sufficient only to search for satisfying solutions in the space $\{0, 1, 2, \ldots, d\}^n$. Then, $S$ is computed using the following equation:

$$S = \lceil \log(d + 1) \rceil \tag{3.1}$$

We prove the following theorem:

**Theorem 3.1** *Let $F_{diff}$ be a DL formula with $n$ variables. Let $b_{\max}$ be the maximum over the absolute values of all difference constraints in $F_{diff}$. Then, $F_{diff}$ is satisfiable if and only if it has a* *solution in* $\{0, 1, 2, \ldots, d\}^n$ *where*

$$d = (n-1) \cdot (b_{\max} + 1)$$

*Proof:* The "if" part of the proof is trivial. Let us consider proving the "only if" part.

Assume initially that $F_{\mathit{diff}}$ does not have any *ITE* expressions.

Since $F_{\mathit{diff}}$ is satisfiable, let $\sigma$ be a satisfying assignment. Under $\sigma$, each difference constraint evaluates to $\mathtt{true}$ or $\mathbf{false}$. Construct the set of difference constraints $\Phi$ as follows:

1. If $\sigma[x_i - x_j \geq b_t] = \mathtt{true}$, add $x_i - x_j \geq b_t$ to $\Phi$.
2. If $\sigma[x_i - x_j \geq b_t] = \mathbf{false}$, add the negation of $x_i - x_j \geq b_t$, viz., $x_j - x_i \geq -b_t + 1$, to $\Phi$.

Consider the constraint graph $\mathcal{G}$ corresponding to $\Phi$. There are $n$ vertices, one for each variable, and at most $m$ edges, one for each constraint or its negation. Note that, while negating constraints, the constant term can increase by at most 1. Therefore, the weight of any edge in $\mathcal{G}$ is at most $b_{\max} + 1$ in absolute value.

The constraint corresponding to each edge in $\mathcal{G}$ is $\mathtt{true}$ under $\sigma$. Therefore, there cannot be any cycles in the graph such that the sum of the weights of the cycle's edges is positive.

Now, construct a graph $\mathcal{G}'$ as follows:

1. Negate the weight of every edge in $\mathcal{G}$. Thus, there is an edge from $v_i$ to $v_j$ of weight $b_t$ in $\mathcal{G}'$ iff $x_i - x_j \geq -b_t$ is a constraint in $\Phi$.
2. Introduce a source vertex $v_{\text{source}}$, and edges of weight 0 from $v_{\text{source}}$ to every $v_i$.

Shortest paths $\delta_i$ from $v_{\text{source}}$ to every $v_i$ are guaranteed to exist since there are no negative cycles in $\mathcal{G}'$. Moreover, for every edge in $\mathcal{G}'$ from $v_i$ to $v_j$ of weight $b_t$, $\delta_j \leq \delta_i + b_t$. In other words, an assignment $\sigma'$ such that $\sigma'[x_i] = \delta_i$ is a satisfying assignment to $F_{\mathit{diff}}$.

Any path in $\mathcal{G}'$ has at most $n-1$ edges, each of weight at most $b_{\max} + 1$. Therefore, for all $i$, $\delta_i = \sigma'[x_i] \leq (n-1) \cdot (b_{\max} + 1)$.

Thus, there exists a satisfying solution in $\{0, 1, 2, \ldots, d\}^n$ where

$$d = (n-1) \cdot (b_{\max} + 1)$$

Finally, if $F_{\mathit{diff}}$ has *ITE* expressions, we can eliminate them using the rewrite rule

$$\mathit{ITE}(\mathit{bool\text{-}expr}, \mathit{num\text{-}expr}_1, \mathit{num\text{-}expr}_2) \geq \mathit{num\text{-}expr}_3$$
$$\downarrow$$
$$[(\mathit{bool\text{-}expr} \implies \mathit{num\text{-}expr}_1 \geq \mathit{num\text{-}expr}_3) \wedge (\neg \mathit{bool\text{-}expr} \implies \mathit{num\text{-}expr}_2 \geq \mathit{num\text{-}expr}_3)]$$

Application of this rewrite rule cannot decrease the values of $n$ or $b_{\max}$. Thus, the bound $d$ applies even if $F_{\mathit{diff}}$ has *ITE* expressions. □

We observe that $S = \mathcal{O}(\log n \cdot \log b_{\max})$, which is polynomial in the input size.

**Remark 3.1** Note that the above analysis is conservative in two respects:

1. Suppose that there are multiple variable classes. There are no edges between vertices corresponding to different variable classes, and hence a separate bound can be computed and employed for each class. If $n_i$ and $b_{\max i}$ are values of $n$ and $b_{\max}$ for variable class $i$, a bound of $(n_i - 1) \cdot (b_{\max i} + 1)$ suffices for variables in that class.
2. The term $(n-1) \cdot (b_{\max} + 1)$ can be replaced by $\sum_{j=1}^{n-1} |b_{i_j} + 1|$, where $b_{i_1}, b_{i_2}, \ldots, b_{i_{n-1}}$ are the $n-1$ largest elements of $b$, in absolute value.

□

**Example 3.2** Consider the following DL formula:

$$(x_1 \geq x_2 \;\wedge\; x_2 \geq x_3 \;\wedge\; x_3 \geq x_1 + 1)$$
$$\vee$$
$$(x_4 \geq x_5 + 100 \;\wedge\; \mathit{ITE}(x_5 \geq x_6 + 50,\, x_6,\, x_7) \geq x_4 - 100)$$

There are two variable classes, viz., $\{x_1, x_2, x_3\}$ and $\{x_4, x_5, x_6, x_7\}$.

For the first class, $n = 3$ and $b_{\max} = 1$. The value of $d$ is therefore $2 \cdot 2 = 4$.

For the second class, $n = 4$ and $b_{\max} = 100$. The value of $d$ is therefore $3 \cdot 101 = 303$.

Using the observation made in Part (2) of Remark 3.1 does not improve the above bounds. □

### Complexity

Computing $n$ and $b_{\max}$ requires a linear scan of the input DL formula.

The propositional encoding can also be done in polynomial time. Each variable is encoded using $S$ bits, and $S$ is polynomial in the input size. Adder, comparator, and multiplexor circuits (required to encode the operators $+$, $\geq$, and *ITE* respectively) are all polynomial in the size of their arguments.

Thus, the small-domain encoding method can be performed in polynomial time. Furthermore, the resulting encoding is polynomial in the size of the input DL formula.

## 3.3 Direct Encoding

Given a DL formula $F_{diff}$, the DIRECT method translates it to an equi-satisfiable Boolean formula $F_{bool}$ in the following $4$ steps:

1. **Preprocessing**: First, all *ITE* expressions are eliminated from $F_{diff}$ by recursively using the following rewrite rules:

$$\begin{array}{c} \textit{ITE}(\textit{bool-expr}, \textit{num-expr}_1, \textit{num-expr}_2) \geq \textit{num-expr}_3 \\ \downarrow \\ [(\textit{bool-expr} \wedge \textit{num-expr}_1 \geq \textit{num-expr}_3) \vee (\neg \textit{bool-expr} \wedge \textit{num-expr}_2 \geq \textit{num-expr}_3)] \end{array} \tag{3.2}$$

$$\begin{array}{c} \textit{num-expr}_1 \geq \textit{ITE}(\textit{bool-expr}, \textit{num-expr}_2, \textit{num-expr}_3) \\ \downarrow \\ [(\textit{bool-expr} \wedge \textit{num-expr}_1 \geq \textit{num-expr}_2) \vee (\neg \textit{bool-expr} \wedge \textit{num-expr}_1 \geq \textit{num-expr}_3)] \end{array} \tag{3.3}$$

   Next, negations are eliminated from the resulting formula. Let the result be $F_{norm}$.

2. **Generate Boolean skeleton**: Each difference constraint $x_i \geq x_j + b$ in $F_{norm}$ is replaced by a fresh Boolean variable $e^b_{i,j}$. This preserves only the Boolean structure of $F_{norm}$. The resulting Boolean formula is denoted by $F_{bvar}$.

3. **Generate transitivity constraints**: In order to preserve the arithmetic information in $F_{norm}$, constraints are generated to disallow satisfying assignments to $F_{bvar}$ that cannot be extended to a satisfying assignment to $F_{norm}$. These constraints, termed as *transitivity constraints*, are generated as follows:

   (a) Construct the constraint graph $\mathcal{G}_{norm}$ corresponding to the set of difference constraints appearing in $F_{norm}$.

   (b) Initialize the Boolean formula $F_{trans}$ to `true`.

   (c) Pick a vertex $v_i$. (Usually, $v_i$ is a vertex for which the product of its in-degree and out-degree is minimum.) If no vertex exists, skip to Step (4).

   Let $(v_j, v_i, b_1)$ and $(v_i, v_k, b_2)$ denote a pair of incoming and outgoing edges incident at $v_i$. For every such pair:

- If $j \neq k$, we add the edge $(v_j,\ v_k,\ b_1 + b_2)$. Additionally, we update $F_{trans}$ as follows:

$$F_{trans} \leftarrow F_{trans} \wedge (e^{b_1}_{j,i} \wedge e^{b_2}_{i,k} \implies e^{b_1+b_2}_{j,k})$$

- If $j = k$, and $b_1 + b_2 > 0$, we update $F_{trans}$ as follows:

$$F_{trans} \leftarrow F_{trans} \wedge (e^{b_1}_{j,i} \wedge e^{b_2}_{i,k} \implies \mathbf{false})$$

(d) Delete $v_i$ and all its incident edges, and return to Step (3c).

Note that the vertex elimination step in the above procedure is Fourier-Motzkin elimination viewed graph-theoretically.

4. **Assemble Boolean encoding**: The final Boolean encoding $F_{bool}$ is $F_{bvar} \wedge F_{trans}$.

**Theorem 3.2** *$F_{diff}$ and $F_{bool}$ are equi-satisfiable.*

*Proof:* First, note that $F_{diff}$ and $F_{norm}$ are equi-satisfiable. Secondly, if $F_{norm}$ is satisfiable, so is $F_{bool}$, since the assignment to difference constraints in $F_{norm}$ can be directly applied to satisfy $F_{bool}$.

We therefore focus on proving that if $F_{bool}$ is satisfiable, so is $F_{norm}$. In particular, we claim we can extend any satisfying assignment $\sigma$ of $F_{bool}$ to $F_{norm}$ such that

$$\sigma[x_i \geq x_j + b] = \sigma[e^b_{i,j}]$$

We will say that an edge $(v_i, v_j, b)$ of $\mathcal{G}_{norm}$ is $\mathtt{true}$ if $\sigma[e^b_{i,j}] = \mathtt{true}$.

The formula $F_{norm}$ is satisfied by $\sigma$ if $\mathcal{G}_{norm}$ does not contain any cycles of positive cumulative weight with all edges $\mathtt{true}$. We will show that under $\sigma$, at least one edge of each positive weight cycle must be **false**.

Consider an arbitrary cycle $C : v_1, v_2 \ldots, v_n, v_1$ of positive cumulative weight. Let $b_2, b_3 \ldots, b_n, b_1$ be the weights of edges $(v_1, v_2), (v_2, v_3), \ldots, (v_n, v_1)$ respectively and let $w(C)$ denote the cumulative weight of cycle $C$. Thus, $w(C) = \sum_{i=1}^{n} b_i > 0$.

Assume without loss of generality that the elimination order is $v_1 < v_2 < \ldots < v_n$. Starting with $C_0 = C$, the $i$th elimination step results in a new cycle $C_i$ such that $|C_i| = |C_{i-1}| - 1$ and $w(C_i) = w(C_{i-1})$. Each projection adds a transitivity constraint. For example, the first elimination adds $e^{b_1}_{n,1} \wedge e^{b_2}_{1,2} \implies e^{b_1+b_2}_{n,2}$. In the $(n-1)$th elimination step we are left with a cycle between $v_{n-1}$ and $v_n$ of weight $w(C)$, at which step the projection method replaces the implicant with **false**.

All together, the following conjunction of constraints appears in $F_{trans}$:

$$\begin{array}{rcl}
e_{n,1}^{b_1} \wedge e_{1,2}^{b_2} & \Longrightarrow & e_{n,2}^{b_1+b_2} \\
 & \wedge & \\
e_{n,2}^{b_1+b_2} \wedge e_{2,3}^{b_3} & \Longrightarrow & e_{n,3}^{b_1+b_2+b_3} \\
 & \wedge & \\
 & \vdots & \\
 & \wedge & \\
e_{n,n-1}^{\sum_{i=1}^{n-1} b_i} \wedge e_{n-1,n}^{b_n} & \Longrightarrow & \mathbf{false}
\end{array}$$

This chain of constraints forces at least one of the edges to be **false**. $\square$

**Example 3.3** We illustrate the DIRECT encoding method using the DL formula introduced in Example 3.2, reproduced below:

$$\begin{gathered}
(x_1 \geq x_2 \;\wedge\; x_2 \geq x_3 \;\wedge\; x_3 \geq x_1 + 1) \\
\vee \\
(x_4 \geq x_5 + 100 \;\wedge\; \mathit{ITE}(x_5 \geq x_6 + 50,\, x_6,\, x_7) \geq x_4 - 100)
\end{gathered}$$

The main steps are outlined below:

1. After eliminating the *ITE* expression, we obtain the following DL formula:

$$\begin{gathered}
(x_1 \geq x_2 \;\wedge\; x_2 \geq x_3 \;\wedge\; x_3 \geq x_1 + 1) \\
\vee \\
\big(x_4 \geq x_5 + 100 \;\wedge\; [(x_5 \geq x_6 + 50 \wedge x_6 \geq x_4 - 100) \;\vee\; (\neg x_5 \geq x_6 + 50 \wedge x_7 \geq x_4 - 100)]\big)
\end{gathered}$$

   Next, we obtain the negation-free form $F_{norm}$:

$$\begin{gathered}
(x_1 \geq x_2 \;\wedge\; x_2 \geq x_3 \;\wedge\; x_3 \geq x_1 + 1) \\
\vee \\
\big(x_4 \geq x_5 + 100 \;\wedge\; [(x_5 \geq x_6 + 50 \wedge x_6 \geq x_4 - 100) \;\vee\; (x_6 \geq x_5 - 49 \wedge x_7 \geq x_4 - 100)]\big)
\end{gathered}$$

2. The Boolean skeleton $F_{bvar}$ is:

$$\begin{gathered}
(e_{1,2}^{0} \;\wedge\; e_{2,3}^{0} \;\wedge\; e_{3,1}^{1}) \\
\vee \\
\big(e_{4,5}^{100} \;\wedge\; [(e_{5,6}^{50} \wedge e_{6,4}^{-100}) \;\vee\; (e_{6,5}^{-49} \wedge e_{7,4}^{-100})]\big)
\end{gathered}$$

3. The constraint graph $\mathcal{G}_{norm}$ corresponding to $F_{norm}$ is the graph depicted in Figure 3.2.

   Suppose we perform Fourier-Motzkin elimination using the heuristic of picking the vertex for which the product of in-degree and out-degree is minimum. One order generated by this heuristic is $v_2 < v_3 < v_1 < v_7 < v_4 < v_5 < v6$. The resulting graph is shown in Figure 3.3.

   The formula $F_{trans}$ comprising of the generated transitivity constraints is

$$
\begin{array}{rcl}
e^0_{1,2} \wedge e^0_{2,3} & \Longrightarrow & e^0_{1,3} \\
& \wedge & \\
e^0_{1,3} \wedge e^1_{3,1} & \Longrightarrow & \mathbf{false} \\
& \wedge & \\
e^{-100}_{6,4} \wedge e^{100}_{4,5} & \Longrightarrow & e^0_{6,5} \\
& \wedge & \\
e^0_{6,5} \wedge e^{50}_{5,6} & \Longrightarrow & \mathbf{false} \\
& \wedge & \\
e^{-49}_{6,5} \wedge e^{50}_{5,6} & \Longrightarrow & \mathbf{false}
\end{array}
$$

□

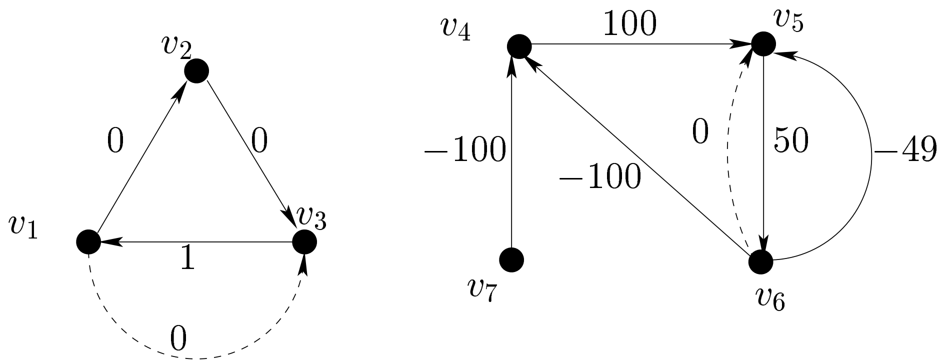



Figure 3.3: **Illustration of** DIRECT **encoding.** The final state of the constraint graph is shown, with original edges indicated by solid lines and new edges indicated by dashed lines.

### Complexity

In the worst case, the DIRECT encoding can generate exponentially many transitivity constraints in the problem size. Here is an example that demonstrates this worst-case behavior.

**Example 3.4** Consider the constraint graph in Figure 3.4. It is cyclic on $n$ vertices $v_1, v_2, \dots, v_n$. There are $n$ edges going from $v_i$ to $v_{i+1}$ for $1 \leq i \leq n-1$ and also from $v_n$ to $v_1$ to close the cycles. The weights on the edges are chosen as follows. For $1 \leq i \leq n-1$, the weights on edges going from $v_i$ to $v_{i+1}$ are $0, n^{i-1}, 2n^{i-1}, \ldots, (n-1)n^{i-1}$. The weights on edges going from $v_n$ to $v_1$ are $0, n^{n-1}, 2n^{n-1}, \ldots, (n-1)n^{n-1}$.

Observe that there are $n^n$ distinct simple cycles in this graph, each with a different cumulative weight in the range $[0, n^n - 1]$.

Thus, no matter what order of vertex elimination we select, in the $(n-2)$th vertex elimination step, there will be $n^{n-1}$ new edges added. Each of these edges will form one edge of a two-edge cycle of cumulative weight in the range $[0, n^n - 1]$.

Since every two-edge cycle yields a corresponding transitivity constraint, $\mathcal{O}(n^n)$ transitivity constraints will be generated on this example.

The weight of each edge in the starting graph, encoded in binary, requires $\mathcal{O}(n \log n)$ space and there are $n^2$ edges to start with. Thus, this example illustrates the worst-case scenario. □

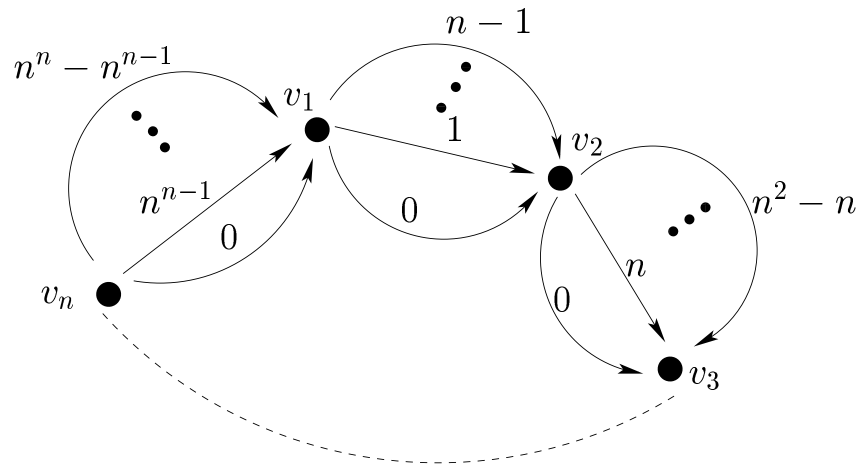



Figure 3.4: **Example demonstrating exponential blow-up of** DIRECT **encoding**

## 3.4 Related Work

The small-domain and direct encoding algorithms were originally proposed for deciding equality logic (and uninterpreted functions) via translation to SAT. Pnueli et al. [122] and Bryant et al. [28] proposed different small-domain encoding algorithms. The former is based on *range allocation*, where the structure of the formulas is analyzed so as to generate a set of values (not necessarily in a contiguous range) for each variable over which it suffices to search for satisfying solutions. The latter approach is based on the notion of *positive equality*, where the polarity of equalities in the formula is analyzed to reduce the small-domain size for certain variables to singleton sets.

The origins of the direct encoding algorithm are in a paper by Goel et al. [62], where the Boolean reasoning is BDD-based. Bryant and Velev [32] later proposed the direct encoding algorithm for equality logic based on generating transitivity constraints; the encoding algorithm for difference logic described in this chapter is an extension of their work.

Recently, Talupur et al. [153] have proposed an extension of Pnueli et al.'s range allocation method for difference logic. While the domains computed using their method can be far more compact than the one derived in this chapter, the algorithm for computing those domains is currently a performance bottleneck.

## 3.5 Discussion

The small-domain encoding method can be viewed as a "model checking approach" to deciding the satisfiability of DL, since it searches for a model for the formula over a finite domain. On the other hand, the direct encoding method can be viewed as a "theorem proving approach," since it is based on creating enough Boolean instances of the axiom of transitivity so as to preserve satisfiability.

An experimental comparison of the SD and DIRECT encoding methods will be made in Chapter 6.

In the remainder of this thesis, we will extend the SD and DIRECT encoding methods to apply to richer logics.

# Part I

# SAT-Based Decision Procedures

# Chapter 4

# Generalized 2SAT Constraints

*Generalized 2SAT constraints* are a special class of linear constraints over integer variables. A generalized 2SAT (G2SAT) constraint (also called a *unit two variable per inequality* or UTVPI constraint) has at most two variables, and variable coefficients are in $\{-1, 1\}$. The variables are not required to have finite upper or lower bounds. Useful optimization problems, such as the minimum vertex cover and the maximum independent set problems, can be modeled using generalized 2SAT constraints, and several applications of constraint logic programming and automated theorem proving also generate G2SAT constraints (e.g., see [10, 81]).

A *G2SAT formula* is a Boolean combination of G2SAT constraints. In this chapter, we consider the problem of checking the satisfiability of G2SAT formulas. It is easily seen that this problem is NP-complete. However, the special case of checking satisfiability of a conjunction of G2SAT constraints (i.e., finding a feasible integer point in a G2SAT polyhedron) can be solved in polynomial time; for example, a modified version of Fourier-Motzkin elimination (reviewed in Section 4.2) runs in $\mathcal{O}(n^3)$ time.

Current approaches (e.g., [10]) to checking the satisfiability of a G2SAT formula employ a combination of Boolean satisfiability solving and linear constraint solving. Truth values are assigned to linear constraints so that the G2SAT formula is satisfied. Each such truth assignment corresponds to a G2SAT polyhedron. If this polyhedron has a feasible integer point, that point satisfies the original G2SAT formula as well. If not, another truth assignment must be found. Given a G2SAT formula $F_{2sat}$ with $m$ constraints and $n$ variables, and assuming that integer feasibility is checked using the afore-mentioned modified Fourier-Motzkin elimination algorithm, the current techniques have a worst-case running time of $\mathcal{O}(2^m \cdot n^3)$.[1]

In this chapter, we prove that a satisfying solution exists for a G2SAT formula $F_{2sat}$ if and only if there is a solution to $F_{2sat}$ with each variable taking values in the finite range $[-n \cdot (b_{\max} + 1), n \cdot$

[1] Assuming the trivial worst-case bound of $\mathcal{O}(2^N)$ for checking satisfiability of a Boolean formula in $N$ variables.

$(b_{\max}+1)]$, where $n$ is the number of variables in $F_{2sat}$, and $b_{\max}$ is the maximum over the absolute values of constant terms in the constraints. That such a bounded solution exists is not surprising, since satisfiability solving of G2SAT formulas is in NP. However, the previously best known solution bounds [22, 84, 118, 160] are $\Omega(n^2 \cdot (b_{\max}+1) \cdot 2^n)$. In particular, our result eliminates the $2^n$ term, thereby exponentially reducing the solution bound.

Our result can be used to implement a small-domain encoding based decision procedure for G2SAT formulas. Such a procedure checks satisfiability of G2SAT formulas in worst-case time $\mathcal{O}(2^{n \log d})$ where $d = 2 \cdot n \cdot (b_{\max} + 1)$, by encoding each integer variable with $\log d$ Boolean variables. This yields a more efficient satisfiability checker for highly over-constrained formulas, where $m = \Omega(n \cdot \log d)$.[2] In our experience, the latter is often the case for theorem proving applications in program analysis and hardware verification.

A key step in our proof is to show that for a G2SAT polyhedron, if a feasible integer point exists, then one exists within a unit hypercube centered at any minimal face solution (extreme point). As a corollary of this result, we obtain a polynomial-time algorithm for approximating optima to an additive factor in generalized 2SAT integer programs.

Our theoretical results are validated by an experimental evaluation (in Section 4.4) on randomly generated G2SAT formulas, which shows that a decision procedure based on our approach can greatly outperform other procedures.

## 4.1 Previous Work

There has been much previous work on integer programming with two variables per inequality (see, e.g., the work by Hochbaum et al. [73–75]). The main differences between this work (applied to G2SAT constraints) and ours are threefold. First, our focus is on satisfiability solving of arbitrary G2SAT formulas and not linear optimization over G2SAT polyhedra. Second, we do not require variables to be bounded. Finally, for our approximation result, the objective function can be an arbitrary linear function, without any restriction on the sign of cost coefficients.

Previous results on bounding solutions have been derived in the context of showing that integer linear programming is in NP [22, 84, 118, 160]. Even when specialized for G2SAT integer programs, these bounds are $\Omega(n^2 \cdot (b_{\max} + 1) \cdot 2^n)$. Our result is therefore an exponential reduction in the solution bound for G2SAT integer programs, and, to the best of our knowledge, has not been obtained before.

Our results rely on the modified version of Fourier-Motzkin elimination for checking integer feasi-

[2]For a conjunction of G2SAT constraints, $m$ is $\mathcal{O}(n^2)$, since one can eliminate redundant constraints. However, for an arbitrary Boolean combination, this is not the case.

bility of a G2SAT polyhedron; this algorithm is described by Subramani [149], and an incremental version has been given by Harvey and Stuckey [66].

Theorem provers that can check G2SAT formulas, such as CVC-Lite [48], are essentially a combination of a SAT solver and a solver for a system of linear constraints. In the case of CVC-Lite, this solver is the Omega test [127], which for G2SAT constraints is identical to the modified Fourier-Motzkin elimination algorithm referenced above.

## 4.2 Background

We state here, in brief, some definitions and theorems used in the remainder of the chapter. Further details can be found in standard textbooks on polyhedral theory and integer linear programming (e.g., [112, 131]).

Following standard linear programming notation, we denote the number of variables by $n$ and number of constraints by $m$. We assume that a linear constraint is specified in the form $\mathbf{a} \cdot \mathbf{x} \geq b$, where $\mathbf{a}$ is a $n$-dimensional integer vector $[a_1,\ a_2,\ \ldots, a_n]$, $\mathbf{x}$ is a $n$-dimensional vector of integer-valued variables $[x_1, x_2, \ldots, x_n]$, and $b$ is an integer. A system of constraints is specified as $A \cdot \mathbf{x} \geq \mathbf{b}$, where $A$ is a $m \times n$ matrix with integral entries, $\mathbf{b}$ is a $m \times 1$ integer vector $[b_1, b_2, \ldots, b_m]^{\mathrm{T}}$, and $\mathbf{x}$ is a $n \times 1$ vector of integer-valued variables. We use $b_{\max}$ to denote the $L_\infty$ norm of $\mathbf{b}$; i.e., $b_{\max} = \max_i |b_i|$.

The terms *feasible* and *satisfiable* are used interchangeably, as also are *lattice point* and *integer point*.

#### G2SAT Formulas

**Definition 4.1** *A constraint* $\mathbf{a} \cdot \mathbf{x} \geq b$ *is said to be an* absolute *constraint if exactly one of the* $a_i s$ *is non-zero, a* pure difference *constraint if exactly two of the* $a_i s$ *are non-zero with one being* $+1$ *and the other* $-1$*, and a* sum *constraint if exactly two of the* $a_i s$ *are non-zero with both* $+1$ *or both* $-1$.

$\mathbf{a} \cdot \mathbf{x} \geq b$ *is said to be a* G2SAT constraint *if it is either an absolute, a pure difference or a sum constraint.*

Note that difference constraints are either absolute or pure difference constraints.

A G2SAT formula is generated by the following grammar:

$$\begin{array}{rcl} F_{2sat} & ::= & \mathbf{true} \mid \mathbf{false} \mid x_1 + x_2 \geq b \mid x_1 - x_2 \geq b \mid x \geq b \\ & \mid & \neg F_{2sat} \mid F_{2sat\,1} \wedge F_{2sat\,2} \mid F_{2sat\,1} \vee F_{2sat\,2} \end{array}$$

Notice that a negation on a G2SAT constraint can be eliminated by rewriting the constraint. A G2SAT constraint remains G2SAT under such rewriting. The only change is to the sign of variable coefficients, and to the constant term, which can increase in absolute value by at most 1.

**Example 4.1** Consider the following G2SAT formula

$$(\neg\, x_1 + x_2 \geq -1) \ \wedge\ \big(x_2 - x_3 \geq 0 \ \vee\ x_4 \geq 1\big)$$

The constraint $x_1 + x_2 \geq -1$ is a sum constraint, $x_2 - x_3 \geq 0$ is a pure difference constraint, and $x_4 \geq 1$ is an absolute constraint. The negation can be eliminated to obtain an equivalent G2SAT formula

$$-x_1 - x_2 \geq 2 \ \wedge\ \big(x_2 - x_3 \geq 0 \ \vee\ x_4 \geq 1\big)$$

Note that the value of $b_{\max}$ has increased from 1 to 2 after eliminating the negation.

Not all families of linear constraints are closed under eliminating negations. For example, the class of Horn-SAT constraints, which comprises all constraints with at most one variable with a positive coefficient, are not closed under eliminating negations.

**Definition 4.2** *Given a G2SAT formula $F_{2sat}$, an* enumeration bound *is an integer $d$ such that $F_{2sat}$ is lattice point feasible if and only if it contains a lattice point in the $n$-dimensional hypercube $\prod_{i=1}^{n}[-d, d]$. The interval $[-d, d]$ is termed as an* enumeration domain.

**Polyhedral Theory**

**Definition 4.3** *A* minimal face *of a polyhedron is a face that does not contain any other face of the polyhedron. A point lying on a minimal face is called a* minimal face solution (MFS).

When the minimal face is an extreme point (a vertex), a MFS is a *basic feasible solution*.

We write $(A',\ \mathbf{b}') \subseteq (A,\ \mathbf{b})$ to indicate that the polyhedral system $A' \cdot \mathbf{x} \geq \mathbf{b}'$ is a subsystem of the polyhedral system $A \cdot \mathbf{x} \geq \mathbf{b}$. Also, for a matrix $A$, let $r(A)$ denote the rank of $A$. We have the following characterization of a minimal face.

**Theorem 4.1 ([131])** *Let $\mathbf{P} = \{\mathbf{x} : A \cdot \mathbf{x} \geq \mathbf{b}\}$ denote a polyhedron. A non-empty subset $\mathbf{F} \subseteq \mathbf{P}$ is a minimal face of $\mathbf{P}$, if and only if $\mathbf{F} = \{\mathbf{x} : A' \cdot \mathbf{x} = \mathbf{b}'\}$, for some system $A' \cdot \mathbf{x} \geq \mathbf{b}'$, where $(A',\ \mathbf{b}') \subseteq (A,\ \mathbf{b})$, and $r(A',\ \mathbf{b}') = r(A,\ \mathbf{b})$.*

Suppose we apply Fourier-Motzkin (FM) elimination to project a variable $x_j$ from a G2SAT polyhedron $\mathbf{P} : A \cdot \mathbf{x} \geq \mathbf{b}$. Denote the resulting polyhedron by $\tilde{\mathbf{P}} : \tilde{A} \cdot \tilde{\mathbf{x}} \geq \tilde{\mathbf{b}}$. In general, $\tilde{\mathbf{P}}$ is not G2SAT. This is because adding a sum constraint involving $x_i$ and $x_j$ with a difference constraint involving those variables can result in a non-G2SAT constraint either of the form $2x_i \geq b$ or $-2x_i \geq b$.

However, it is possible to modify the basic FM elimination procedure by adding a *coefficient normalization* step, so that the resulting polyhedron remains G2SAT, and moreover, is lattice point feasible iff $\mathbf{P}$ is. The modification hinges on the observation that the only non-G2SAT constraints in $\tilde{\mathbf{P}}$ are of the form $2x_i \geq b$ or $-2x_i \geq b$. By dividing both sides of a newly created non-G2SAT constraint by 2, and rounding up the RHS if it is an odd multiple of $\frac{1}{2}$, we obtain a G2SAT constraint with the same integral solutions as the original. In this way, we replace each non-G2SAT constraint in $\tilde{\mathbf{P}}$ with a corresponding G2SAT constraint to obtain a G2SAT polyhedron $\mathbf{P}' : A' \cdot \mathbf{x}' \geq \mathbf{b}'$.

We will refer to the modified FM elimination procedure as *Fourier-Motzkin elimination with coefficient normalization* (FM-CN). It is easy to see that FM-CN preserves integral solutions, i.e., $\mathbf{P}$ is lattice point feasible iff $\mathbf{P}'$ is. One can use FM-CN to check the feasibility of G2SAT polyhedra in time $\mathcal{O}(n^3)$, by successively eliminating variables, checking at each step that we do not generate a trivially false constraint. At any step, we are guaranteed to have a system of no more than $\mathcal{O}(n^2)$ constraints, since there are only $4 \cdot \binom{n}{2}$ possible non-redundant G2SAT constraints on $n$ variables.

## 4.3 Theoretical Results

Our theoretical results are organized as follows. We begin, in Section 4.3.1, by showing that if a G2SAT polyhedron has a minimal face solution (MFS), then there exists a MFS with each component half-integral and in $[-n \cdot b_{\max},\ n \cdot b_{\max}]$. The main theorem, presented in Section 4.3.3, enables us to go from bounding a MFS to bounding integer solutions. This theorem states that if a G2SAT polyhedron is integer feasible, then it is possible to find a integral solution within a unit box centered at any MFS; i.e., by "rounding" a MFS. In this section, we also describe how to extend results for G2SAT polyhedra to arbitrary G2SAT formulas. Section 4.3.2 presents auxiliary results on rounding that are used to prove the main theorem. Finally, in Section 4.3.4, we show that the main theorem can be used to obtain an additive approximation result for optimizing an arbitrary linear constraint over a G2SAT polyhedron.

### 4.3.1 Minimal Face Solutions of G2SAT Polyhedra

We begin with a useful lemma.

**Lemma 4.1** *Let* $\mathbf{P} : A \cdot \mathbf{x} \geq \mathbf{b}$ *represent a system of* $m$ *pure difference constraints on* $n$ *variables. Then,* $\mathbf{P}$ *has a feasible integer solution if and only if it has an integer solution in the hypercube* $\prod_{i=1}^{n}[0,\ (n-1) \cdot b_{\max}]$.

*Proof:* Follows from Theorem 3.1. □

The following lemma considers bounding a MFS of a G2SAT polyhedron in the non-negative orthant.

**Lemma 4.2** *Let* $\mathbf{P} : A \cdot \mathbf{x} \geq \mathbf{b},\ \mathbf{x} \geq \mathbf{0}$ *denote an arbitrary G2SAT polyhedron in the non-negative orthant with* $m$ *constraints and* $n$ *variables. Then, if a MFS exists, then there is a MFS with each component half-integral and at most* $n \cdot b_{\max}$.

*Proof:* Suppose polyhedron $\mathbf{P}$ has a minimal face solution. Hochbaum et al. [75] have shown that this MFS must be half-integral. We focus here on showing the $n \cdot b_{\max}$ bound.

By definition, the minimal face corresponding to this MFS satisfies a system $A' \cdot \mathbf{x} = \mathbf{b}'$, where $(A'\ \mathbf{b}') \subseteq (A\ \mathbf{b})$, and $r(A') = r(A) = k$ for some $1 \leq k \leq n$ (assuming, w.l.o.g., that $m \leq n$). Accordingly, there are $k$ independent variables and $n - k$ dependent variables in the system; without loss of generality, we assume that the first $k$ variables are independent and set the dependent variables to 0. This results in a system $\mathbf{P_1} : A'' \cdot \mathbf{x}'' = \mathbf{b}'',\ \mathbf{x}'' \geq \mathbf{0}$, where the components of $\mathbf{b}''$ are also components of $\mathbf{b}$, and $\mathbf{x}'' = [x_1, x_2, \ldots, x_k]^T$.

The system $\mathbf{P_1}$ contains 3 types of constraints (equations), viz., absolute, pure difference, and sum. We consider each of these types in turn:

1. An absolute constraint is of the form $x_i = b$. Since $\mathbf{x}'' \geq \mathbf{0}$, the value of $x_i$ must be in $[0, b_{\max}]$.

2. A sum constraint can be written in the form $x_i + x_j = b$, where $b \geq 0$. Since $\mathbf{x}'' \geq \mathbf{0}$, it follows that $0 \leq x_i, x_j \leq b \leq b_{\max}$.

3. From the two cases above, we conclude that the value of any variable appearing in an absolute or sum constraint must lie in $[0, b_{\max}]$ (and moreover, there exists such a half-integral value).

   W.l.o.g, let $x_1, x_2, \ldots, x_l$, $l \leq k$, be variables appearing in the absolute and sum constraints, and let $x_1^*, x_2^*, \ldots, x_l^*$ be the corresponding half-integral values in $[0, b_{\max}]$ satisfying these constraints. Substituting these values into the pure difference constraints might create new absolute constraints, but no new pure difference or sum constraints. The constant term in new absolute constraints generated thus is half-integral and of absolute value at most $2b_{\max}$. The substitution process can be iterated at most $k - 1$ times leading to absolute constraints with half-integral constant terms at most $k \cdot b_{\max}$. Thus, a variable appearing in any of the absolute constraints generated in this iterative process takes half-integral values in $[0, k \cdot b_{\max}]$.

   When the above iterative substitution process terminates, the only constraints possibly left are some of the original pure difference constraints, each with an integral constant term of absolute value at most $b_{\max}$. Since these constraints are satisfiable, we can apply Lemma 4.1 to conclude that there exists a solution to these constraints with each variable taking integral values in $[0, (k - 1) \cdot b_{\max}]$ (since at most $k$ variables appear in these constraints).

Since $k \leq n$, we conclude that there exists a solution to $\mathbf{P_1}$ with each component at most $n \cdot b_{\max}$. $\square$

We now generalize the result to an arbitrary G2SAT polyhedron.

**Theorem 4.2** *Let $\mathbf{P} : A \cdot \mathbf{x} \geq \mathbf{b}$ denote an arbitrary G2SAT polyhedron with $m$ constraints and $n$ variables. If a MFS exists, there exists a MFS with each component half-integral and in the interval $[-n \cdot b_{\max},\ n \cdot b_{\max}]$.*

*Proof:* Suppose $\mathbf{x}^*$ is a MFS of $\mathbf{P}$. Let $j_1, j_2, \ldots, j_k$ be the set of all column indices, $1 \leq j_1, j_2, \ldots, j_k \leq n$, such that $x^*_{j_l} < 0$ for all $l$, $1 \leq l \leq k$. Construct a matrix $A'$ by multiplying the $j_l$th column of $A$ by $-1$ for all $l$, leaving other columns unchanged. We observe that:

1. The polyhedron $\mathbf{P}' : A' \cdot \mathbf{x} \geq \mathbf{b},\ \mathbf{x} \geq \mathbf{0}$ is also G2SAT.
2. If we construct $\mathbf{x}'^*$ from $\mathbf{x}^*$ by negating $x^*_{j_l}$ for all $l$, $1 \leq l \leq k$, $\mathbf{x}'^*$ satisfies $\mathbf{P}'$. Moreover, we argue that it is a MFS of $\mathbf{P}'$ as follows:
   Let $(\tilde{A}, \tilde{\mathbf{b}}) \subseteq (A, \mathbf{b})$ be the constraints satisfied with equality at $\mathbf{x}^*$, and $(\tilde{A}', \tilde{\mathbf{b}'}) \subseteq (A', \mathbf{b}')$ be the constraints satisfied with equality at $\mathbf{x}'^*$. Then, $r(\tilde{A}) = r(\tilde{A}')$, since $\tilde{A}$ and $\tilde{A}'$ correspond to the same rows (of $A$ and $A'$ respectively). Also, note that $r(A) = r(A')$. Finally, since $(\tilde{A}, \tilde{\mathbf{b}})$ define a minimal face of $\mathbf{P}$, $r(\tilde{A}) = r(A)$ [131].
   Thus, $r(\tilde{A}') = r(A')$, and so $\mathbf{x}'^*$ is a MFS of $\mathbf{P}'$.

Using an identical argument, we conclude that, from a MFS of $\mathbf{P}'$, we can construct a MFS of $\mathbf{P}$ by negating values to $x_{j_1}, x_{j_2}, \ldots, x_{j_k}$.

Since $\mathbf{P}'$ has a MFS, by Lemma (4.2) it must have a MFS with each component half-integral and in $[0, n \cdot b_{\max}]$. It follows that $\mathbf{P}$ has a MFS with each component half-integral and in $[-n \cdot b_{\max},\ n \cdot b_{\max}]$. $\square$

**Remark 4.1** Note that the enumeration bound stated in Theorem 4.2 is tight.

First, notice that if $b_{\max} = 0$, then the origin is a MFS, and the bound is tight.

Even if $b_{\max} > 0$, the enumeration domain is still tight in that one of its end points can be attained. For example, suppose that the system of constraints comprises the following $n$ equalities:

$$
\begin{array}{rcll}
x_1 & = & b_{\max} & \\
x_i - x_{i-1} & = & b_{\max} & \quad 2 \leq i \leq n-1 \\
x_n + x_{n-1} & = & -b_{\max} &
\end{array}
$$

It is easy to see that the solution set spans the interval $[-n \cdot b_{\max},\ (n-1) \cdot b_{\max}]$. $\square$

### 4.3.2 Rounding and Semi-Rounding

**Definition 4.4** *A rational number $x$ is said to be* odd half-integral *if it is an odd multiple of $\frac{1}{2}$.*

**Definition 4.5** *A vector $\mathbf{z}$ is said to be a* rounding *of a vector $\mathbf{x}$ if $\mathbf{z}$ is integral and $||\mathbf{z}-\mathbf{x}||_\infty \leq \frac{1}{2}$.*

**Definition 4.6** *A vector $\mathbf{z}$ is said to be a* semi-rounding *of a vector $\mathbf{x}$ if all of the following conditions hold: (1) $||\mathbf{z}-\mathbf{x}||_\infty \leq \frac{1}{2}$; (2) all components of $\mathbf{z}$ are half-integral; and (3) if a component of $\mathbf{x}$ is integral, so is the corresponding component of $\mathbf{z}$.*

**Lemma 4.3** *Let $\mathbf{a}\cdot\mathbf{x} \geq b$ be a G2SAT constraint. Let $\mathbf{x}^*$ be a half-integral vector such that $\mathbf{a}\cdot\mathbf{x}^* > b$, and let $\mathbf{w}^*$ be an arbitrary semi-rounding of $\mathbf{x}^*$. Then, $\mathbf{a}\cdot\mathbf{w}^* \geq b$.*

*Proof:* The proof proceeds by case splitting on the number of variables in the constraint.

1. Suppose the constraint involves only one variable. Then, it is either of the form $x_i \geq b$ or $-x_i \geq b$. Correspondingly, we either have $x_i^* > b$ or $-x_i^* > b$. Since $x_i^*$ is half-integral, in both cases the LHS exceeds $b$ by at least $\frac{1}{2}$. Thus, any semi-rounding $w_i^*$ of $x_i^*$ satisfies the constraint.

2. Suppose the constraint has two variables, $x_i$ and $x_j$. Then, since $x_i^*$ and $x_j^*$ are both half-integral, one of the following two cases must hold:

   (a) The LHS is integral, and exceeds $b$ by at least 1. But any semi-rounding of $x_i^*$ and $x_j^*$ can decrease the LHS by at most 1, and hence satisfies the constraint.

   (b) The LHS is odd half-integral, i.e., one of $x_i^*$ and $x_j^*$ is integral and the other odd half-integral. Thus, the LHS exceeds $b$ by at least $\frac{1}{2}$. In this case, any semi-rounding of $x_i^*$ and $x_j^*$ can decrease the LHS by at most $\frac{1}{2}$, and will satisfy the constraint.

□ Since every rounding $\mathbf{z}$ of $\mathbf{x}^*$ is also a semi-rounding of $\mathbf{x}^*$, we obtain the following corollary:

**Corollary 4.1** *Let $\mathbf{a}\cdot\mathbf{x} \geq b$ be a G2SAT constraint. Let $\mathbf{x}^*$ be a half-integral vector such that $\mathbf{a}\cdot\mathbf{x}^* > b$, and let $\mathbf{z}$ be an arbitrary rounding of $\mathbf{x}^*$. Then, $\mathbf{a}\cdot\mathbf{z} \geq b$.*

We now state a useful property of Fourier-Motzkin elimination with coefficient normalization.

**Proposition 4.1** *Let $\mathbf{P} : A\cdot\mathbf{x} \geq \mathbf{b}$ denote a G2SAT polyhedron in $\mathbb{R}^{n+1}$ and $\mathbf{x}^* = (x_1^*, x_2^*, \ldots, x_{n+1}^*)$ denote a half-integral feasible solution to $\mathbf{P}$. Further, suppose that $\mathbf{P}$ is lattice point feasible.*

*Let $\mathbf{P}' : A'\cdot\mathbf{x}' \geq \mathbf{b}'$ be obtained from $\mathbf{P}$ by projecting out variable $x_{n+1}$ using Fourier-Motzkin elimination with coefficient normalization and denote $(x_1^*, x_2^*, \ldots, x_n^*)$ by $\mathbf{x}'^*$. Then, there exists a semi-rounding $\mathbf{w}'^*$ of $\mathbf{x}'^*$ such that $\mathbf{w}'^*$ is a solution to $\mathbf{P}'$.*

*Proof:* First, note that since $\mathbf{P}$ is lattice point feasible, so is $\mathbf{P}'$.

If $\mathbf{x}'^*$ is already a solution to $\mathbf{P}'$ then the theorem holds trivially.

So suppose that $\mathbf{x}'^*$ does not satisfy $\mathbf{P}'$. The only reason this occurs is because $\mathbf{x}'^*$ is cut off by coefficient normalization, i.e., due to the presence of one or both of the following situations:

1. There exists at least one variable $x_i$, $i \in I$, such that $\mathbf{P}$ has constraints of the form:

$$x_i - x_{n+1} \geq b_i \tag{4.1}$$

$$x_i + x_{n+1} \geq b'_i \tag{4.2}$$

   which result in the following constraint in $\mathbf{P}'$:

$$x_i \geq \left\lceil \frac{b_i + b'_i}{2} \right\rceil \tag{4.3}$$

   where, $b_i + b'_i$ is odd.

   Since $\mathbf{x}'^*$ does not satisfy $\mathbf{P}'$, the following equality also holds:

$$x_i^* = \frac{b_i + b'_i}{2} \tag{4.4}$$

2. There exists at least one variable $x_j$, $j \in J$, such that $\mathbf{P}$ has constraints of the form:

$$-x_j + x_{n+1} \geq b_j \tag{4.5}$$

$$-x_j - x_{n+1} \geq b'_j \tag{4.6}$$

   which result in the following constraint in $\mathbf{P}'$:

$$x_j \leq \left\lfloor \frac{-b_j - b'_j}{2} \right\rfloor \tag{4.7}$$

   where, $b_j + b'_j$ is odd.

   Since $\mathbf{x}'^*$ does not satisfy $\mathbf{P}'$, the following equality also holds:

$$x_j^* = \frac{-b_j - b'_j}{2} \tag{4.8}$$

Note that for some $i \in I$, and $j \in J$, if $i = j$, then we must have $\frac{b_i + b'_i}{2} = \frac{-b_j - b'_j}{2}$. But that would mean that $\mathbf{P}'$ is infeasible, since constraints (4.3) and (4.7) would contradict each other. Hence, we can assume hereafter that the two index sets $I$ and $J$ are disjoint.

We now give a rounding algorithm that generates a semi-rounding $\mathbf{w}'^*$ of $\mathbf{x}'^*$ that satisfies $\mathbf{P}'$. The rounding algorithm is as follows:

1. Initialize the set of variables to be rounded up, $\mathcal{U}$, to be $\{x_i | i \in I\}$. Similarly, initialize the set of variables to be rounded down, $\mathcal{D}$ as $\{x_j | j \in J\}$.

2. $\mathcal{U}_0 := \mathcal{U}, \mathcal{D}_0 := \mathcal{D}, t := 0$.

3. Compute $\mathcal{U}_{t+1}$ and $\mathcal{D}_{t+1}$ as follows. For every $x_i \in \mathcal{U}_t$ and $x_j \in \mathcal{D}_t$,

   (a) Include in $\mathcal{U}_{t+1}$ any variable $x_k$ such that the following constraints in $\mathbf{P}'$, which are valid for $\mathbf{P}$, hold with equality at $\mathbf{x}'^*$:

$$x_k - x_i \geq b_{ki} \tag{4.9}$$
$$x_j + x_k \geq b_{jk} \tag{4.10}$$

   (b) Include in $\mathcal{D}_{t+1}$ any variable $x_k$ such that the following constraints in $\mathbf{P}'$, which are valid for $\mathbf{P}$, hold with equality at $\mathbf{x}'^*$:

$$-x_k - x_i \geq b'_{ki} \tag{4.11}$$
$$x_j - x_k \geq b'_{jk} \tag{4.12}$$

4. If $\mathcal{U}_{t+1} \subseteq \mathcal{U}$ and $\mathcal{D}_{t+1} \subseteq \mathcal{D}$, stop.

   Otherwise, perform the assignments $\mathcal{U} := \mathcal{U} \cup \mathcal{U}_{t+1}$, $\mathcal{D} := \mathcal{D} \cup \mathcal{D}_{t+1}$, $t := t + 1$, and go to step (3).

It is easy to prove by induction on $t$, that for any $x_k \in \mathcal{U}$, $k \notin I$, there either exists $i \in I$ and an integer $b_{ki}$ such that

$$x_k^* - x_i^* = b_{ki} \tag{4.13}$$

or a $j \in J$ and an integer $b_{jk}$ such that

$$x_j^* + x_k^* = b_{jk} \tag{4.14}$$

Similarly, for each $x_k \in \mathcal{D}$, $k \notin J$, there either exists $i \in I$ and an integer $b'_{ki}$ such that

$$-x_k^* - x_i^* = b'_{ki} \tag{4.15}$$

or a $j \in J$ and an integer $b'_{jk}$ such that

$$x_j^* - x_k^* = b'_{jk} \tag{4.16}$$

Suppose the two sets $\mathcal{U}$ and $\mathcal{D}$ are disjoint. Then, to obtain a semi-rounding $\mathbf{w}'^*$ of $\mathbf{x}'^*$, we round up every variable in $\mathcal{U}$ and round down every variable in $\mathcal{D}$.

To complete the proof, the following two sub-goals remain to be established:

1. $\mathcal{U} \cap \mathcal{D} = \emptyset$.

2. $\mathbf{w}'^*$ satisfies $\mathbf{P}'$.

Assuming the first sub-goal, consider the second sub-goal first. We observe that:

- By Lemma 4.3, any constraints in $\mathbf{P}'$ that are not satisfied with equality at $\mathbf{x}'^*$ will continue to be satisfied by $\mathbf{w}'^*$.

- From Equations (4.13)–(4.16), we note that for all $x_k \in \mathcal{U} \cup \mathcal{D}$, $x_k^*$ is odd half-integral, since it is an integral offset from $x_i^*$ or $x_j^*$ for some $i \in I$ or $j \in J$.

  Thus, for all $x_k \in \mathcal{U} \cup \mathcal{D}$, there cannot be any absolute constraint involving $x_k$ in $\mathbf{P}'$ that holds with equality at $\mathbf{x}'^*$. Thus, by Lemma 4.3, the semi-rounding produced by the above algorithm satisfies these absolute constraints.

- Steps 3(a) and 3(b) of the rounding algorithm ensure that all two-variable constraints of $\mathbf{P}'$ satisfied with equality at $\mathbf{x}'^*$ continue to be satisfied by the generated semi-rounding. For example, if $x_k - x_i \geq b_{ki}$ is satisfied with equality at $\mathbf{x}'^*$, and $x_i^*$ is rounded up, so is $x_k^*$, so the constraint continues to be satisfied.

Thus, if the two sets $\mathcal{U}$ and $\mathcal{D}$ are disjoint, we can conclude that $\mathbf{w}'^*$ satisfies $\mathbf{P}'$. We will now show that the former is indeed the case.

The proof is by contradiction. Suppose $\mathcal{U} \cap \mathcal{D} \neq \emptyset$. Let $x_k$ be a variable present in both sets. As we noted before, for any $i \in I$ and $j \in J$, $i \neq j$, so we can assume that $k$ is neither in $I$ nor in $J$. We have the following cases, each of which leads to a contradiction:

1. Equations (4.13) and (4.16) hold. Then, for some integer $b_{ji}$, we have

$$x_j^* - x_i^* = b_{ji} \tag{4.17}$$

   The above equation corresponds to the following inequality derived by adding Inequalities (4.9) and (4.12), which is valid for both $\mathbf{P}$ and $\mathbf{P}'$:

$$x_j - x_i \geq b_{ji} \tag{4.18}$$

   Further, from Equation (4.17) and Inequalities (4.1), (4.2), (4.5), and (4.6), we can conclude that

$$-b_{ji} = x_i^* - x_j^* \quad \geq \quad b_i + b_j \tag{4.19}$$

$$-b_{ji} = x_i^* - x_j^* \quad \geq \quad b_i' + b_j' \tag{4.20}$$

Also from Equations (4.4) and (4.8), we know that

$$-b_{ji} = x_i^* - x_j^* = \frac{b_i + b_j + b'_i + b'_j}{2} \tag{4.21}$$

From (4.19), (4.20), and (4.21) above, we infer that $b_i + b_j = b'_i + b'_j = -b_{ji}$.

Thus, the inequalities in (4.19) and (4.20) hold with equality. Also, from Inequalities (4.1) and (4.5), $x_i - x_j \geq b_i + b_j$ is valid for $\mathbf{P}$. Thus, we can conclude that Inequality (4.18) holds with equality for $\mathbf{P}$. This further implies that Inequalities (4.1), (4.2), (4.5), and (4.6) hold with equality for $\mathbf{P}$.

Since there is a unique solution to Constraints (4.1), (4.2), (4.5), (4.6) and (4.18) that satisfies them with equality, in every feasible solution of $\mathbf{P}$, $x_i = x_i^*$, $x_j = x_j^*$, and $x_{n+1} = x_{n+1}^*$. Since at least one of $x_i^*$ and $x_j^*$ is odd half-integral, this contradicts the premise that $\mathbf{P}$ has a lattice point solution.

2. Equations (4.14) and (4.15) hold. This case is identical to Case (1) above.

3. Equations (4.14) and (4.16) hold. Then, we have

$$x_j^* = \frac{b_{jk} + b'_{jk}}{2} \tag{4.22}$$

This implies that $\frac{b_{jk}+b'_{jk}}{2} = \frac{-b_j - b'_j}{2}$.

Further, Equation (4.22) corresponds to the following valid cut for $\mathbf{P}'$ (i.e., it preserves lattice point solutions), obtained by adding (4.10) and (4.12):

$$x_j \geq \left\lceil \frac{b_{jk} + b'_{jk}}{2} \right\rceil \tag{4.23}$$

However, Constraints (4.7) and (4.23) contradict each other, implying that $\mathbf{P}'$ is not lattice point feasible, which contradicts the theorem's premise.

4. Equations (4.13) and (4.15) hold. This case is identical to Case (3) above.

Thus, $\mathcal{U} \cap \mathcal{D} = \emptyset$ and we obtain a semi-rounding $\mathbf{w}'^*$ of $\mathbf{x}'^*$ as required. This completes the proof.

□

### 4.3.3 Main Theorems

We now arrive at the key result of this chapter.

**Theorem 4.3** *Let* $\mathbf{P} : A \cdot \mathbf{x} \geq \mathbf{b}$ *denote a G2SAT polyhedron and* $\mathbf{x}^*$ *denote a half-integral MFS. If* $\mathbf{P}$ *is lattice point feasible, then it contains a lattice point* $\mathbf{z}$ *such that* $||\mathbf{z} - \mathbf{x}^*||_\infty \leq \frac{1}{2}$, *i.e.,* $\mathbf{z}$ *is a rounding of* $\mathbf{x}^*$.

*Proof:* We prove the theorem by induction on the length of $\mathbf{x}$.

*Base Case:* Let $\mathbf{x} = x \in \mathbb{R}$. If $x^*$ is a MFS, there exists a constraint $x \geq b$ that holds with equality for $x^*$. Thus, the theorem holds trivially for $\mathbf{z} = x^*$.

*Induction Step:* Let us assume that the theorem holds for all vectors $\mathbf{x}$ of length up to $n$.

Consider the case when $\mathbf{x} \in \mathbb{R}^{n+1}$. Since $\mathbf{P}$ has a MFS, by Theorem (4.2), it has one with half-integral entries. Let $\mathbf{x}^* = (x_1^*, x_2^*, \ldots, x_{n+1}^*)$ be one such MFS of $\mathbf{P}$. If $\mathbf{x}^*$ is integral, we set $\mathbf{z}$ to $\mathbf{x}^*$ and we are done. So, let us assume that $\mathbf{x}^*$ has some odd half-integral entries. Note that if two variables $x_i$ and $x_j$ appear together in a constraint of $\mathbf{P}$ that holds with equality, either both $x_i^*$ and $x_j^*$ are integral or both are odd half-integral.

Project variable $x_{n+1}$ out of $\mathbf{P}$ using Fourier-Motzkin elimination with coefficient normalization (FM-CN). Let $\mathbf{P}' : A' \cdot \mathbf{x}' \geq \mathbf{b}'$ be the resulting system, where $\mathbf{x}' \in \mathbb{R}^n$.

Suppose there exists a lattice point solution $\mathbf{y} = (y_1, y_2, \ldots, y_{n+1})$ of $\mathbf{P}$. Thus, $\mathbf{y}' = (y_1, y_2, \ldots, y_n)$ is a lattice point solution of $\mathbf{P}'$.

Consider $\mathbf{x}'^* = (x_1^*, x_2^*, \ldots, x_n^*)$. We will show that there exists a rounding $\mathbf{z}' = (z_1, z_2, \ldots, z_n)$ of $\mathbf{x}'^*$ which satisfies $\mathbf{P}'$. We consider the following three cases:

*Case 1:* $\mathbf{x}'^*$ is in the interior of $\mathbf{P}'$, i.e., none of the constraints in $A' \cdot \mathbf{x}' \geq \mathbf{b}'$ hold with equality. By Corollary 4.1, any rounding of $\mathbf{x}'^*$ yields a lattice point solution $\mathbf{z}'$ of $\mathbf{P}'$.

*Case 2:* Suppose that $\mathbf{x}'^*$ is a solution of $\mathbf{P}'$ that satisfies some constraints with equality. Suppose that for some $(A'', \mathbf{b}'') \subseteq (A', \mathbf{b}')$, $A'' \cdot \mathbf{x}'^* = \mathbf{b}''$, and the remaining constraints are strict, i.e., not satisfied with equality. Since $\mathbf{x}'^*$ is a MFS of $A'' \cdot \mathbf{x}' \geq \mathbf{b}''$, by the induction hypothesis, we can conclude that there exists a lattice point rounding $\mathbf{z}'$ of $\mathbf{x}'^*$, such that $\mathbf{z}'$ is a solution of $A'' \cdot \mathbf{x}' \geq \mathbf{b}''$. Since, by Corollary 4.1, any rounding of $\mathbf{x}'^*$ satisfies the strict constraints, $\mathbf{z}'$ is also a lattice point solution of $\mathbf{P}'$.

*Case 3:* It is possible that after coefficient normalization, $\mathbf{x}'^*$ does not satisfy $\mathbf{P}'$. By Proposition 4.1, there exists a semi-rounding $\mathbf{w}'^*$ of $\mathbf{x}'^*$ that satisfies $\mathbf{P}'$. Thus, either Case (1) or Case (2) applies with $\mathbf{x}'^*$ replaced by $\mathbf{w}'^*$, and we can obtain a rounding $\mathbf{z}'$ of $\mathbf{w}'^*$ that is a lattice point solution of $\mathbf{P}'$. Finally, note that a rounding of $\mathbf{w}'^*$ is also a rounding of $\mathbf{x}'^*$, since integral components of $\mathbf{x}'^*$ are preserved in $\mathbf{w}'^*$. This completes Case (3).

Thus, we can obtain a lattice point solution $\mathbf{z}'$ of $\mathbf{P}'$ that is a rounding of $\mathbf{x}'^*$.

Since $\mathbf{P}$ is G2SAT, and $\mathbf{P}'$ is obtained from $\mathbf{P}$ using FM-CN, a lattice point solution of $\mathbf{P}'$ can be extended to one of $\mathbf{P}$. Thus, there exists an integral $z_{n+1}$ such that $\mathbf{z} = (z_1, z_2, \ldots, z_n, z_{n+1})$ is a solution of $\mathbf{P}$.

To complete the proof, we show that there exists such an integral $z_{n+1}$ that is moreover a rounding of $x^*_{n+1}$. Since $\mathbf{x}^*$ is a MFS of $\mathbf{P}$, there exists a subset of constraints $(\tilde{A}, \tilde{\mathbf{b}})$ of $(A, \mathbf{b})$ that hold with equality at $\mathbf{x}^*$. The value of $x_{n+1}$ is constrained only by the values of other variables $x_j$ such that there exists an equation in $\tilde{A}\mathbf{x} = \tilde{\mathbf{b}}$ in which $x_{n+1}$ and $x_j$ appear together. Let $J$ be the index set of all such variables $x_j$. We now show that there exists a rounding $z_{n+1}$ of $x^*_{n+1}$ that satisfies $\mathbf{P_1} : \tilde{A}\mathbf{x} \geq \tilde{\mathbf{b}}$. There are two cases:

1. If $x^*_{n+1}$ is integral, so is $x^*_j$ for all $j \in J$. Thus, $z_{n+1} = x^*_{n+1}$ satisfies $\mathbf{P_1}$, and we are done.
2. If $x^*_{n+1}$ is odd half-integral, so is $x^*_j$ for all $j \in J$. In this case, we claim that there exists a consistent way to round $x^*_{n+1}$, either up or down, so that the result satisfies $\mathbf{P_1}$. Suppose not, i.e., there exists constraints that force $x^*_{n+1}$ to be rounded up as well as down. There are four instances in which this might occur:
   (a) There exist constraints $x_{n+1} - x_i \geq b$ and $x_j - x_{n+1} \geq b'$ in $\mathbf{P}$ that hold with equality at $\mathbf{x}^*$; furthermore, $z_j = \lfloor x^*_j \rfloor$ and $z_i = \lceil x^*_i \rceil$. Thus, we have $x^*_j - x^*_i = b + b'$, but $z_j - z_i < b + b'$. Since $x_j - x_i \geq b + b'$ is a valid inequality for $\mathbf{P}$, this means that $\mathbf{z}$ does not lie in $\mathbf{P}$, a contradiction.
   (b) There exist constraints $-x_{n+1} - x_i \geq b$ and $x_j + x_{n+1} \geq b'$ in $\mathbf{P}$ that hold with equality at $\mathbf{x}^*$; furthermore, $z_j = \lfloor x^*_j \rfloor$ and $z_i = \lceil x^*_i \rceil$. This case is identical to Case (2a) above.
   (c) There exist constraints $x_j - x_{n+1} \geq b$ and $x_j + x_{n+1} \geq b'$ in $\mathbf{P}$ that hold with equality at $\mathbf{x}^*$, with $z_j = \lfloor x^*_j \rfloor$. Thus, $2x^*_j = b + b'$. Since, $x^*_j$ is odd half-integral, $b + b'$ must be an odd integer. Moreover, $2z_j < b + b'$. However, since $2x_j \geq b + b'$ is a valid inequality for $\mathbf{P}$, this means that $\mathbf{z}$ does not lie in $\mathbf{P}$, a contradiction.
   (d) There exist constraints $x_{n+1} - x_i \geq b$ and $-x_{n+1} - x_i \geq b'$ in $\mathbf{P}$ that hold with equality at $\mathbf{x}^*$, with $z_i = \lceil x^*_i \rceil$. This case is identical to Case (2c) above.

   Thus, there exists a consistent way to round $x^*_{n+1}$ either up or down and satisfy every constraint in $\mathbf{P_1}$. Let $z_{n+1}$ be this rounding.

Applying Corollary 4.1, any rounding of $x^*$ satisfies the constraints in $(A, \mathbf{b}) \setminus (\tilde{A}, \tilde{\mathbf{b}})$.

Thus, we can obtain a rounding $\mathbf{z}$ of $\mathbf{x}^*$ that is a lattice point solution of $\mathbf{P}$.

□

From Theorem (4.2) and Theorem (4.3), we can conclude the following theorem.

**Theorem 4.4** *Let* $\mathbf{P} : A \cdot \mathbf{x} \geq \mathbf{b}$ *denote a G2SAT polyhedron with* $m$ *constraints and* $n$ *variables. Then,* $\mathbf{P}$ *has enumeration bound* $n \cdot b_{\max}$.

The above result is easily generalized for arbitrary G2SAT formulas.

**Theorem 4.5** *Let $F_{2sat}$ denote a G2SAT formula with $m$ constraints, $n$ variables, and let $b_{\max}$ be the maximum over the absolute values of constant terms appearing in $F_{2sat}$. Then, $F_{2sat}$ has enumeration bound $n \cdot (b_{\max} + 1)$.*

*Proof:* If $F_{2sat}$ has a satisfying integer solution, that solution must satisfy one of the terms in the disjunctive normal form (DNF) of $F_{2sat}$. Each term in the DNF representation of $F_{2sat}$ is a G2SAT polyhedron in which the constant term in any constraint has absolute value at most $b_{\max} + 1$ (we use $b_{\max} + 1$ in place of $b_{\max}$ to account for eliminating negations on constraints). It follows that there is a solution to $F_{2sat}$ in $[-n \cdot (b_{\max} + 1),\ n \cdot (b_{\max} + 1)]$. □

### 4.3.4 Approximation Results for Optimization

Consider the problem of optimizing an arbitrary linear function over a G2SAT polyhedron $\mathbf{P}$. This problem is NP-hard (minimum vertex cover is a special case). As a corollary of Theorem (4.3), we obtain the following theorem showing that one can approximate the optimal value to within an additive factor.

**Theorem 4.6** *Let $\mathbf{P} = \{\mathbf{x} : A \cdot \mathbf{x} \geq \mathbf{b}\}$ denote a G2SAT polyhedron that contains a lattice point. Let the integer linear program be $\max\{\mathbf{c} \cdot \mathbf{x} : \mathbf{x} \in \mathbf{P}\}$.*

*If the optimum value is finite, solving the LP-relaxation and rounding the solution can yield a feasible lattice point that approximates the optimum to within an additive factor of $\pm\frac{\sum_{j=1}^{n}|c_j|}{2}$. If the LP-relaxation is unbounded, so is the integer program.*

*Proof:* If the optimum value $v^*$ of the LP-relaxation is finite, it is attained at a MFS $\mathbf{x}^*$. Since $\mathbf{P}$ is lattice point feasible, by Theorem 4.3, there exists a lattice point $\mathbf{z}$ in $\mathbf{P}$ such that such that $||\mathbf{z} - \mathbf{x}^*||_\infty \leq \frac{1}{2}$. It follows that $\mathbf{c} \cdot \mathbf{z}$ is within $\pm\frac{\sum_{j=1}^{n}|c_j|}{2}$ of $v^*$, and hence of the integer optimum.

If the LP-relaxation is unbounded, so must the integer program, since $\mathbf{P}$ is lattice point feasible [112]. □

Moreover, an approximate solution can be obtained in polynomial time in the following three steps:

1. Check whether $\mathbf{P}$ is lattice point feasible using Fourier-Motzkin elimination with coefficient normalization. If $\mathbf{P}$ is lattice point infeasible, stop.
2. If $\mathbf{P}$ is lattice point feasible, solve its LP-relaxation. If it is unbounded, we conclude that the original IP is also unbounded. Otherwise, the optimum is attained at a MFS $\mathbf{x}^*$.
3. Round $\mathbf{x}^*$ to obtain an integer solution that is within $\pm\frac{\sum_{j=1}^{n}|c_j|}{2}$ of the optimum. The rounding is performed as follows. For each variable $x_i$ that has an odd half-integral value $x_i^*$, we check whether adding the constraint $x_i = \lceil x_i^* \rceil$ to $\mathbf{P}$ preserves lattice point feasibility. If not, we set $x_i$

to $\lfloor x_i^* \rfloor$ and iterate, picking another variable to round, until we have obtained a feasible integer solution.

It is easy to see that each step can be performed in polynomial time. Notice that if lattice point feasibility is preserved by setting $x_i$ either to $\lceil x_i^* \rceil$ or to $\lfloor x_i^* \rfloor$, the direction of rounding can be chosen heuristically to obtain a tighter approximation.

Our approximation theorem is general, in that it applies to any generalized 2SAT integer program, including non 0-1 programs with *arbitrary coefficients* in the objective function. However, the approximation factor is additive, and the result is more likely to be useful for non 0-1 programs. In contrast, the results of Hochbaum et al. [75] guarantee a 2-approximation for G2SAT integer programs expressed as a minimization problem where the objective function is required to have non-negative coefficients.

## 4.4 Experimental Evaluation

We now present experimental results demonstrating that a decision procedure based on the solution bound derived herein can outperform other state-of-the-art procedures.

### 4.4.1 Implementation

We implemented a decision procedure that operates in three steps. First, given a G2SAT formula $F_{2sat}$, it computes the enumeration bound $n \cdot (b_{\max} + 1)$. Second, it translates the input G2SAT formula to a Boolean formula by replacing each integer variable by a finite-precision, signed bit-vector that can take any value in the range $[-n \cdot (b_{\max}+1),\ n \cdot (b_{\max}+1)]$. Arithmetic and relational operators are then encoded as arithmetic circuits and comparators. Let $F_{bool}$ denote the resulting Boolean formula. Clearly, $F_{bool}$ is satisfiable if and only if $F_{2sat}$ is satisfiable. Thus, the final step consists of invoking a Boolean satisfiability (SAT) solver on $F_{bool}$. Notice that the translation to SAT takes polynomial time and that the size of $F_{bool}$ is polynomial in that of $F_{2sat}$.

The main reason for using a translation to SAT, as opposed to a non-SAT-based procedure, is that our benchmarks possess a non-trivial Boolean structure. Also, by this approach, we can leverage the recent advances in SAT solving (e.g., [63, 104]). For our experiments, we employed the zChaff satisfiability solver [104]; however, any other SAT solver can be employed instead just as easily.

### 4.4.2 Setup

A set of randomly generated G2SAT formulas was used for the experimental evaluation. A G2SAT formula can be viewed as a Boolean circuit where the inputs to the circuit are G2SAT constraints rather than being Boolean variables. Each formula was generated based on 3 parameters: the maximum number of variables, an upper bound on the size of the constant term, and the maximum depth of the circuit. We varied the maximum number of variables over the set $\{40, 80, 160, 320, 640\}$, the constant term upper bound over the set $\{16, 256, 4096, 65536, 1048576\}$, and the maximum circuit depth over $\{6, 7, 8, 9, 10\}$. For each choice of these three parameters, we generated a formula using one of three different random seeds; the seed was used in generating, at each level in the circuit, either a randomly chosen Boolean operator or a G2SAT constraint. The variables and constant term in each G2SAT constraint were randomly generated as well. Finally, the resulting G2SAT formula was conjoined with a set of upper and lower bound constraints on each variable, where the bounds were randomly selected to be between 0 and the upper bound on the constant term. This last operation was performed in order to generate a mix of both satisfiable and unsatisfiable formulas. Thus, in total, the benchmark suite comprises 375 formulas, of which 202 are unsatisfiable.

We compared our procedure against two other decision procedures. Both are based on a combination of a SAT solver with a solver for a system of integer linear constraints. The first is a publicly available theorem prover called CVC-Lite [48] (the version available as of December 2004). CVC-Lite uses a SAT solver for finding Boolean assignments to the formula, treating G2SAT constraints as Boolean literals. For every such assignment, it decides the feasibility of the corresponding conjunction of G2SAT constraints by using the FM-CN procedure (it actually uses the Omega test [127], which specializes to FM-CN for G2SAT constraints). Details about CVC-Lite's operation can be found in the papers by Barrett et al. and Ganesh et al. [13, 17]. The SAT solver used by CVC-Lite is a modified version of the zChaff solver used by our procedure. The second decision procedure, written by Daniel Kroening (currently at ETH Zürich), works on similar principles to CVC-Lite, except that it uses the CPLEX commercial optimization software [46] (version 9.0) instead of the FM-CN procedure. This procedure also uses the zChaff solver as its SAT solving engine.

Experiments were run on a Linux workstation with a 2 GHz Pentium 4 processor and 1 GB of RAM. Our decision procedure, called UCLID, is written mostly in Moscow ML, a dialect of Standard ML. A timeout of 600 seconds was imposed on each run.

### 4.4.3 Comparison

Figures 4.1 and 4.2 compare UCLID's total time (time for both encoding and SAT solving) to that taken by CVC-Lite and the CPLEX-based solver respectively. In each plot, the y-coordinate of a point is the time taken by UCLID, and the x-coordinate is the time taken by the decision procedure we compare it against. UCLID's total time is dominated by the SAT solving time. Note that the X and Y axes are on different scales. This is because UCLID finishes within 30 seconds on all benchmarks whereas the run-times for the other solvers are spread out over the entire range $[0, 600]$.

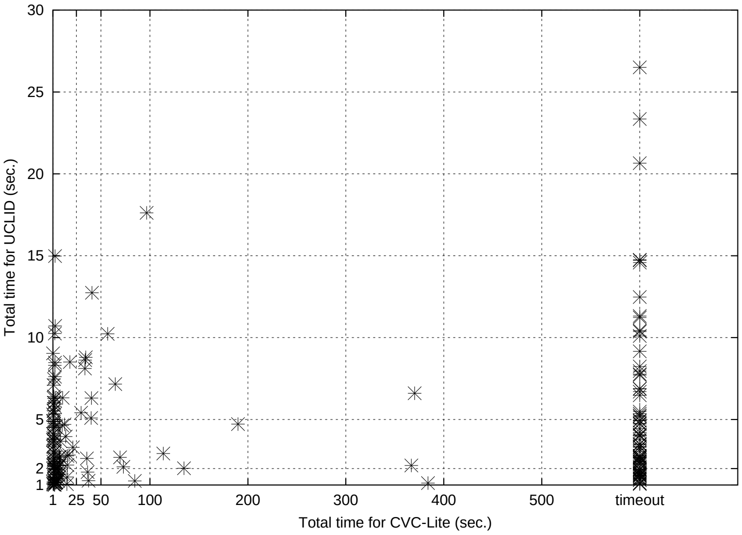



Figure 4.1: **Experimental comparison of UCLID versus CVC-Lite for G2SAT formulas.** Note that the scale on the Y-axis is about 20 times that of the X-axis.

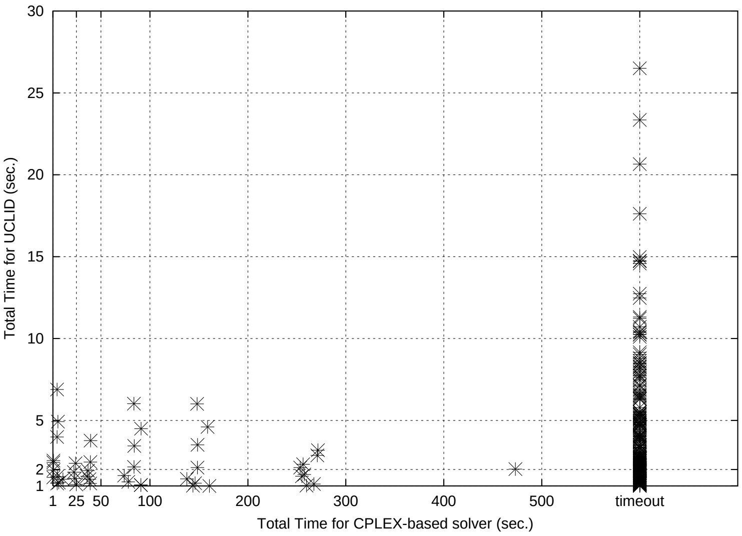



Figure 4.2: **Experimental comparison of UCLID versus CPLEX-based solver for G2SAT formulas.** Note that the scale on the Y-axis is about 20 times that of the X-axis.

First, consider the comparison with CVC-Lite. We observe from Figure 4.1 that CVC-Lite performs worse than UCLID overall, timing out on 95 of the 375 benchmarks. However, note that there are 171 benchmarks on which CVC-Lite outperforms UCLID. UCLID completes within 15 seconds on all of these benchmarks, and within 5 seconds on all but 22 of them.

The comparison with the CPLEX-based solver yields similar results, as one can observe in Figure 4.2. In fact, the CPLEX-based solver even performs worse than CVC-Lite, timing out on 246 of the 375 benchmarks. UCLID is outperformed on only 16 benchmarks, on all of which it terminates within 7 seconds.

We further analyzed our results by dividing the benchmarks into 4 categories, with each category comprising benchmarks on which UCLID's time falls within a certain range. For each category, we computed the percentage of benchmarks on which UCLID outperforms the other two solvers. This data is displayed in Table 4.1. We note that the benchmarks on which UCLID is outperformed are those on which both it and the competing solver finish within a few seconds. Note also that UCLID finishes within 5 seconds on over 80% of the benchmarks.

| UCLID time range (time in seconds) | Number of benchmarks | % of benchmarks on which UCLID runs faster | |
|---|---|---|---|
| | | CVC-Lite prover | CPLEX-based solver |
| [0, 5] | 315 | 52.38 | 95.24 |
| (5, 10] | 42 | 54.76 | 97.62 |
| (10, 20] | 15 | 80.00 | 100.00 |
| (20, 30) | 3 | 100.00 | 100.00 |

Table 4.1: **Comparing UCLID with other solvers using a time-wise break-up of benchmarks.** The second column indicates the number of benchmarks on which UCLID's run-time is within the indicated range.

Thus, one can conclude that the enumerative approach presented herein can greatly outperform a more traditional approach based on combining a SAT solver with a constraint solver. The main reason for this seems to be that solvers based on the latter approach enumerate several SAT assignments that, while satisfying the Boolean skeleton of the formula, correspond to infeasible systems of G2SAT constraints. On the other hand, UCLID's encoding adds in all the "G2SAT information" necessary for the SAT solver to significantly prune its search space.

## 4.5 Summary

We have proposed a new approach to deciding the satisfiability of Boolean combinations of generalized 2SAT constraints. The central insight is that it is sufficient to search for bounded solutions, where each variable is restricted within the finite range $[-n \cdot (b_{\max} + 1), n \cdot (b_{\max} + 1)]$. The solution bound we derive improves over previous results by an exponential factor. The key step in our derivation is a novel result for G2SAT polyhedra on finding integer solutions by rounding minimal face solutions. Experiments demonstrate the efficacy of a SAT-based decision procedure based on our theoretical results.

# Chapter 5

# Quantifier-Free Presburger Arithmetic

*Presburger arithmetic* [125] is defined as the first-order theory of the structure $\langle \mathbb{N}, 0, 1, \leqslant, + \rangle$, where $\mathbb{N}$ denotes the set of natural numbers. The satisfiability problem for Presburger arithmetic is decidable, but of super-exponential worst-case complexity [54]. Fortunately, for many applications, such as in program analysis (e.g., [127]) and hardware verification (e.g., [26]), the quantifier-free fragment suffices. We are concerned, in this chapter, with the satisfiability problem for this fragment.

A formula $F_{qfp}$ in quantifier-free Presburger arithmetic (QFP) is constructed by combining linear constraints with Boolean operators ($\wedge$, $\vee$, $\neg$). Formally, the $i^{\text{th}}$ constraint is of the form

$$\sum_{j=1}^{n} a_{i,j} x_j \geq b_i$$

where the coefficients and the constant terms are integer constants and the variables $x_1, x_2, \dots, x_n$ are integer-valued[1]. An integer linear program is a conjunction of linear constraints, and hence is a special kind of QFP formula.

The satisfiability problem for QFP is NP-complete. The NP-hardness follows from a straightforward encoding of the 3SAT problem as a 0-1 integer linear program. That it is moreover in NP can be concluded from the result that integer linear programming is in NP [22, 84, 118, 160].

Thus, if there is a satisfying solution to a QFP formula, there is one whose size, measured in bits, is polynomially bounded in the problem size. Problem size is traditionally measured in terms of the parameters $m$, $n$, $\log a_{\max}$, and $\log b_{\max}$. Recall that $m$ is the total number of constraints in the formula, $n$ is the number of variables, and $a_{\max} = \max_{(i,j)} |a_{i,j}|$ and $b_{\max} = \max_i |b_i|$ are the maximums of the absolute values of coefficients and constant terms respectively.

[1] While Presburger arithmetic is defined over $\mathbb{N}$, we interpret the variables over $\mathbb{Z}$ as it is general and more suitable for applications. It is straightforward to translate a formula with integer variables to one where variables are interpreted over $\mathbb{N}$, and vice-versa, by adding (linearly many) additional variables or constraints.

| Project | Maximum Fraction of Non-Difference Constraints | Maximum Width of a Non-Difference Constraint |
|---|---|---|
| Blast | 0.0255 | 6 |
| Magic | 0.0032 | 2 |
| MIT | 0.0087 | 3 |
| WiSA | 0.0091 | 4 |

Table 5.1: **Linear arithmetic constraints in software verification are mostly difference constraints.** For each software verification project, the maximum fraction of non-difference constraints is shown, as well as the maximum width of a non-difference constraint, where the maximum is taken over all formulas in the set. The Blast formulas were generated from device drivers written in C, the Magic formulas from an implementation of `openssl` written in C, the MIT formulas from Java programs, and the WiSA formulas were generated in the checking of format string vulnerabilities.

The above result implies that we can use a small-domain (SD) encoding approach to checking the satisfiability of a QFP formula $F_{qfp}$. To recapitulate, we first compute the polynomial bound $S$ on solution size, and then search for a satisfying solution to $F_{qfp}$ in the bounded space $\{0, 1, \dots, 2^S - 1\}^n$. However, a naïve implementation of a SD-based decision procedure fails for QFP formulas encountered in practice. The problem is that the bound on solution size, $S$, is $\mathcal{O}(\log m + \log b_{\max} + m[\log m + \log a_{\max}])$. In particular, the presence of the $m \log m$ term means that, for practical problems involving hundreds of linear constraints, the Boolean formulas generated are likely to be too large to be decided by present-day SAT solvers.

In this chapter, we explore the small-domain encoding approach to deciding QFP formulas, but with a focus on formulas generated in software verification. It has been observed, by us and others, that formulas from this domain have:

1. *Mainly Difference Constraints:* Of the $m$ constraints, $m - k$ are *difference* constraints, where $k \ll m$.

2. *Sparse Structure:* The $k$ non-difference constraints are sparse, with at most $w$ variables per constraint, where $w$ is "small". We will refer to $w$ as the *width* of the constraint.

Pratt [124] observed that most inequalities generated in program verification are difference constraints. More recently, the authors of the theorem prover Simplify observed in the context of the Extended Static Checker for Java (ESC/Java) project that "the inequalities that occur in program checking rarely involve more than two or three terms" [53]. We have performed a study of formulas generated in various recent software verification projects: the Blast project at Berkeley [69], the Magic project at CMU [36], the Wisconsin Safety Analyzer (WiSA) project [164], and the software upgrade checking project at MIT [97]. The results of this study, indicated in Table 5.1, support the afore-mentioned observations regarding the "sparse, mostly difference" nature of constraints in QFP formulas. To our knowledge, no previous decision procedure for QFP has attempted to exploit this problem structure.

The following novel contributions are made in this chapter:

- We derive bounds on solutions for QFP formulas, not only in terms of the traditional parameters $m$, $n$, $a_{\max}$, and $b_{\max}$, but also in terms of $k$ and $w$. In particular, we show that the worst-case number of bits required per integer variable is linear in $k$, but only logarithmic in $w$. Unlike previously derived bounds, ours is not in terms of the total number of constraints $m$.
- We use the derived bounds in a sound and complete decision procedure for QFP based on small-domain encoding, and present empirical evidence that our method can greatly outperform other decision procedures.

The rest of this chapter is organized as follows. We begin with a discussion of related work (Section 5.1) and some background material (Section 5.2). Our main theoretical results on computing solution bounds are presented in Section 5.3. Techniques for improving the bound in practice are discussed in Section 5.4. An experimental evaluation is presented in Section 5.5, followed by a discussion in Section 5.6.

## 5.1 Related Work

There has been much work on deciding quantifier-free Presburger arithmetic; we present a brief discussion here and refer the reader to a recent survey [59] for more details. Recent techniques fall into four categories.

### Enumerating DNF terms

The first class comprises procedures targeted towards solving conjunctions of constraints, with disjunctions handled by enumerating terms in a disjunctive normal form (DNF). Examples include the Omega test [127] (which is an extension of Fourier-Motzkin elimination for integers) and solvers based on other integer linear programming techniques. The drawback of these methods is the need to enumerate the potentially exponentially many terms in the DNF representation. Our work is targeted towards solving formulas with a complicated Boolean structure, which often arise in verification applications.

### Lazy translation to SAT

The second set of methods attempt to remedy the above problem by instead relying on modern SAT solving strategies. The approach works as follows. A Boolean abstraction of the QFP formula $F_{qfp}$ is generated by replacing each linear constraint with a corresponding Boolean variable. If the abstraction is unsatisfiable, then so is $F_{qfp}$. If not, the satisfying assignment (model) is checked for consistency with the theory of quantifier-free Presburger arithmetic, using a ground decision procedure for conjunctions of linear constraints (a procedure for checking feasibility of integer linear programs). Assignments that are inconsistent are excluded from later consideration by adding a "lemma" to the Boolean abstraction. The process continues until either a consistent assignment is found, or all (exponentially many) assignments have been explored. Examples of decision procedures in this class that have some support for QFP include CVC [13, 17] and ICS [51]. (The general idea for combining a SAT solver with a linear programming engine originates in a paper by Wolfman and Weld [165].) The ground decision procedures used by provers in this class employ a combination framework such as the Nelson-Oppen architecture for cooperating decision procedures [109] or a Shostak-like combination method [139, 141]. These methods are only defined for combining disjoint theories. In order to exploit the mostly-difference structure of a formula, one approach could be to combine a decision procedure for a theory of difference constraints with one for a theory of non-difference constraints, but this needs an extension of the combination methods that applies to these non-disjoint theories.

### Eager translation to SAT

Strichman [146] presents SAT-based decision procedures for linear arithmetic (over the rationals) and QFP. The translation to SAT is a generalization of DIRECT encoding for arbitrary linear constraints. For QFP, the basic idea is to create a Boolean encoding of all the possible variable projection steps performed by the Omega test. Since Fourier-Motzkin elimination (and therefore, the Omega test) has worst-case double-exponential complexity in both time and space [37], this approach leads to a SAT problem that, in the worst-case, is doubly-exponential in the size of the original formula and takes doubly-exponential time to generate.

Our approach also falls in this category. However, in contrast to Strichman's translation, our encoding algorithm generates SAT problems that are polynomial in the size of the original formulas, and runs in polynomial time.

#### Automata theory-based methods

The final class of methods are based on finite automata theory (e.g., [59, 166]). The basic idea is to construct a finite automaton corresponding to the input QFP formula $F_{qfp}$, such that language accepted by the automaton consists of the binary encodings of satisfying solutions of $F_{qfp}$. According to a recent experimental evaluation with other methods [59], these techniques are better than others at solving formulas with very large coefficients, but do not scale well with the number of variables and constraints.

Note that automata-based techniques can handle full Presburger arithmetic, not just the quantifier-free fragment.

#### Unique features of our approach

The approach we present in this chapter is distinct from the categories mentioned above. In particular, the following unique features differentiate it from previous methods:

- It is the first small-domain encoding method and the first tractable procedure for translating a QFP formula to SAT in a single step. The clear separation between the translation and the SAT solving allows us to leverage future advances in SAT solving far more easily than other SAT-based procedures.
- It is the first technique, to the best of our knowledge, that formally exploits the structure of formulas commonly encountered in software verification.

In addition to the above, the bounds we derive in this chapter are also of independent theoretical interest. For instance, they indicate that the solution bound is independent of the number of difference constraints.

## 5.2 Background

We define useful notation and state the previous results on bounding satisfying solutions of ILPs.

### 5.2.1 Preliminaries

Consider a system of $m$ linear constraints in $n$ integer-valued variables:

$$A\mathbf{x} \geq \mathbf{b} \qquad (5.1)$$

Here $A$ is an $m \times n$ matrix with integral entries, $\mathbf{b}$ is a $m \times 1$ vector of integral entries, and $\mathbf{x}$ is a $n \times 1$ vector of integer-valued variables. A satisfying solution to system (5.1) is an evaluation of $\mathbf{x}$ that satisfies (5.1).

As outlined in Section 2.1, the variables can be constrained to be non-negative by adding a zero variable $x_0$, replacing each original variable $x_i$ by $x'_i - x_0$, and then adjusting the coefficients in the matrix $A$ to get a new constraint matrix $A'$ and the following system:[2]

$$\begin{aligned} A'\mathbf{x}' &\geq \mathbf{b} \\ \mathbf{x}' &\geq \mathbf{0} \end{aligned} \tag{5.2}$$

Here the system has $n' = n + 1$ variables, and $\mathbf{x}' = [x'_1, x'_2, \dots, x'_n, x_0]^T$. $A'$ has the structure that $a'_{i,j} = a_{i,j}$ for $j = 1, 2, \dots, n$ and $a'_{i,n+1} = -\sum_{j=1}^{n} a_{i,j}$. Note that the last column of $A'$ is a linear combination of the previous $n$ columns. Proposition 2.1 shows that system (5.1) has a solution if and only if system (5.2) has one.

Finally, adding surplus variables to the system, we can rewrite system (5.2) as follows:

$$\begin{aligned} A''\mathbf{x}'' &= \mathbf{b} \\ \mathbf{x}'' &\geq \mathbf{0} \end{aligned} \tag{5.3}$$

where $A'' = [A| - I_m]$ is an $m \times (n' + m)$ integer matrix formed by concatenating $A$ with the negation of the $m \times m$ identity matrix $I_m$.

For convenience we will drop the primes, referring to $A''$ and $\mathbf{x}''$ simply as $\mathbf{A}$ and $\mathbf{x}$. Rewriting system (5.3) thus, we get

$$\begin{aligned} A\mathbf{x} &= \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned} \tag{5.4}$$

**Remark 5.1** A solution to system (5.4) also satisfies system (5.2).

We formally define the terms *solution bound* and *enumeration bound* for QFP formulas.

**Definition 5.1** *Given a QFP formula $F_{qfp}$, a* solution bound *is an integer $d$ such that $F_{qfp}$ has an integer solution if and only if it has an integer solution in the $n$-dimensional hypercube $\prod_{i=1}^{n}[0, d]$.*

**Definition 5.2** *Given a QFP formula $F_{qfp}$, an* enumeration bound *is an integer $d$ such that $F_{qfp}$ has an integer solution if and only if it has an integer solution in the $n$-dimensional hypercube $\prod_{i=1}^{n}[-d, d]$. The interval $[-d, d]$ is termed as an* enumeration domain.

[2]Note that this procedure can increase the width of a constraint by 1. The statistics in Table 5.1 shows the width before this procedure is applied, computed from constraints as they appear in the original formulas.

The following proposition is easily obtained.

**Proposition 5.1** *A solution bound $d \geq 0$ for system (5.2) is an enumeration bound for system (5.1).*

*Proof:* Given a solution $\mathbf{x}'^*$ to system (5.2), we construct a solution $\mathbf{x}^*$ to system (5.1) by setting $x_j^* = {x'_j}^* - x_0^*$. Since each ${x'_j}^*$ and $x_0^*$ are in $[0, d]$, $x_j^* \in [-d, d]$ for all $j$. □

Similarly, if $d$ is an enumeration bound for system (5.1), then $2d$ is a solution bound for system (5.2).

### 5.2.2 Previous Results

The bounds for QFP follow directly from those for integer linear programs. In particular, the results of this chapter build on a result obtained by Borosh, Treybig, and Flahive [21, 22] on bounding the solution of systems of the form (5.4). We state their result in the following theorem:

**Theorem 5.1** *Consider the augmented matrix $[A|\mathbf{b}]$ of dimension $m \times (n' + m + 1)$. Let $\Delta$ be the maximum of the absolute values of all minors of this augmented matrix. Then, the system (5.4) has a satisfying solution if and only if it has one with all entries bounded by $(n + 2)\Delta$.*

Note that the determinant of a matrix can be more than exponential in the dimension of the matrix [25]. In the case of the Borosh-Flahive-Treybig result, it means that $\Delta$ can be as large as $\frac{\mu^m (m+1)^{(m+1)/2}}{2^m}$, where $\mu = \max(a_{\max}, b_{\max})$.

Papadimitriou [118, 120] also gives a bound of similar size, stated in the following theorem:

**Theorem 5.2** *If the ILP of (5.4) has a satisfying solution, then it has a satisfying solution where all entries in the solution vector are bounded by $(n' + m)(1 + b_{\max})(m\, a_{\max})^{2m+3}$.*

Papadimitriou's bound implies that we need $\mathcal{O}(\log m + \log b_{\max} + m[\log m + \log a_{\max}])$ bits to encode each variable (assuming $n' = \mathcal{O}(m)$). The Borosh-Flahive-Treybig bound implies needing $\mathcal{O}(m[\log m + \log \mu])$ bits per variable, which is of the same order.

## 5.3 Main Theoretical Results

We begin in Section 5.3.1 by deriving bounds for ILPs for the case of $k = 0$, when all constraints are difference constraints. Then, in Section 5.3.2, we compute a bound for ILPs for arbitrary $k$. Finally, in Section 5.3.3, we show how our results extend to arbitrary QFP formulas.

### 5.3.1 Bounds for a System of Difference Constraints

Let us first consider computing solution bounds for an ILP for the case where $k = 0$, i.e., system (5.4) comprises only of difference constraints.

In this case, the left-hand side of each equation comprises exactly three variables: two variables $x_i$ and $x_j$ where $0 \leq i, j \leq n$ and one surplus variable $x_l$ where $n + 1 \leq l \leq n + m$. The $t^{\text{th}}$ equation in the system is of the form $x_i - x_j - x_l = b_t$.

As we noted in Section 5.2.1, the matrix $A$ can be written as $[A_o| - I_m]$ where $A_o$ comprises the first $n' = n + 1$ columns, and $I_m$ is the $m \times m$ identity matrix.

The important property of $A_o$ is that each row has exactly one $+1$ entry and exactly one $-1$ entry, with all other entries 0. Thus, $A_o^T$ can be interpreted as the node-arc incidence matrix of a directed graph. Therefore, $A_o^T$ is *totally unimodular* (TUM), i.e., every square submatrix of $A_o^T$ has determinant in $\{0, -1, +1\}$ [120]. Therefore, $A_o$ is TUM, and so is $A = [A_o| - I_m]$.

Now, let us consider using the Borosh-Flahive-Treybig bound stated in Theorem 5.1. This bound is stated in terms of the minors of the matrix $[A|\mathbf{b}]$. For the special case of this section, we have the following bound on the size of any minor:

**Theorem 5.3** *The absolute value of any minor of* $[A|\mathbf{b}]$ *is bounded above by* $s\, b_{\max}$*, where* $s = \min(n + 1, m)$.

*Proof:*

Consider any minor $M$ of $[A|\mathbf{b}]$. Let $r$ be the order of $M$.

If the minor is obtained by deleting the last column (corresponding to $\mathbf{b}$), then it is a minor of $A$, and its value is in $\{0, -1, +1\}$ since $A$ is TUM. Thus, the bound of $s\, b_{\max}$ is attained for any non-trivial minor with $s \geq 1$ and $b_{\max} \geq 1$.

Suppose the $\mathbf{b}$ column is not deleted.

First, note that the matrix $A$ is of the form $[A_o| - I_m]$ where the rank of $A_o$ is at most $s' = \min(n, m)$. This is because $A_o$ has dimensions $m \times n + 1$, and the last column of $A_o$, corresponding to the variable $x_0$, is a linear combination of the previous $n$ columns. (Refer to the construction of system (5.2) from system (5.1).)

Next, suppose the sub-matrix corresponding to $M$ comprises $p$ columns from the $-I_m$ part, $r - p - 1$ columns from the $A_o$ part, and the column corresponding to $\mathbf{b}$. Since permuting the rows and columns of $M$ does not change its absolute value, we can permute the rows of $M$ and the columns corresponding to the $-I_m$ part to get the corresponding sub-matrix in the following form:

$$\left[\begin{array}{c|cccc|c} & 0 & \dots & 0 & -1 & b_{t_1} \\ & 0 & \dots & -1 & 0 & b_{t_2} \\ A_o & \vdots & \dots & \vdots & \vdots & \vdots \\ \text{part} & -1 & \dots & 0 & 0 & b_{t_p} \\ & 0 & \dots & 0 & 0 & b_{t_{p+1}} \\ & \vdots & \dots & \vdots & \vdots & \vdots \\ & 0 & \dots & 0 & 0 & b_{t_r} \end{array}\right]$$

Expanding $M$ along the last column, we get

$$|M| = |b_{t_1} M_1 - b_{t_2} M_2 + b_{t_3} M_3 - \dots (-1)^{r-1} b_{t_r} M_r|$$

where each $M_i$ is a minor corresponding to a submatrix of $A$.

However, notice that $M_i = 0$ for all $1 \leq i \leq p$, since each of those minors have an entire column (from the $-I_m$ part) equal to $0$. Therefore, we can reduce the right-hand side to the sum of $r - p$ terms:

$$|M| \leq |b_{t_{p+1}} M_{p+1}| + |b_{t_{p+2}} M_{p+2}| + \dots |b_{t_r} M_r|$$

Notice that, so far, we have not made use of the special structure of $A$.

Now, observing that $A$ is TUM, $|M_i| \leq 1$ for all $i$.

$$|M| \leq |b_{t_{p+1}}| + |b_{t_{p+2}}| + \dots + |b_{t_r}|$$

For all $i$, $|b_{t_i}| \leq b_{\max}$. Further, since each non-zero $M_i$ can be of order at most $s'$, $r - p \leq s = min(s' + 1, m)$.[3] Therefore, we get

$$|M| \leq s\, b_{\max}$$

□

Using the terminology of Theorem 5.1, we have $\Delta \leq s\, b_{\max}$. Thus, the solution bound $d$ in this case is $(n + 2)\, s\, b_{\max}$.

Thus, $S$, the bound on the number of bits per variable, is

$$\lceil \log(n + 2) + \log s + \log b_{\max} \rceil$$

Formulas generated from verification problems tend to be overconstrained, so we assume $n < m$. Thus, $s = n + 1$, and the bound reduces to $\mathcal{O}(\log n + \log b_{\max})$ bits per variable.

We close this section with the following two observations about Theorem 5.3.

[3] We use $s' + 1$ and not $s'$ to account for the case where $p = 0$. The minimum with $m$ is taken because $s' + 1$ can exceed $m$ but $\mathbf{b}$ has only $m$ elements.

**Remark 5.2** The derived solution bound is conservative. From Theorem 3.1, we know that a tighter solution bound is $n \cdot b_{\max}$. This indicates that there might be room for improving the bound in Theorem 5.1.

**Remark 5.3** The only property of the $A$ matrix that the proof of Theorem 5.3 relies on is the totally unimodular (TUM) property. Thus, Theorem 5.3 would also apply to any system of linear constraints whose coefficient matrix is TUM. Examples of such matrices include *interval* matrices, or more generally *network* matrices. Note that the TUM property can be tested for in polynomial time [131].

### 5.3.2 Bounds for a Sparse System of Mainly Difference Constraints

We now consider the general case for ILPs, where we have $k$ non-difference constraints, each referring to at most $w$ variables.

Without loss of generality, we can reorder the rows of matrix $A$ so that the $k$ non-difference constraints are the top $k$ rows, and the difference constraints are the bottom $m-k$ rows. Reordering the rows of $A$ can only change the sign of any minor of $[A|\mathbf{b}]$, not the absolute value. Thus, the matrix $[A|\mathbf{b}]$ can be put into the following form:

$$\left[\begin{array}{c|c|c} A_1 & & b_1 \\ \cline{1-1} & -I_m & b_2 \\ A_2 & & \vdots \\ & & b_m \end{array}\right]$$

Here, $A_1$ is a $k \times n+1$ dimensional matrix corresponding to the non-difference constraints, $A_2$ is a $m-k \times n+1$ dimensional matrix with the difference constraints, $I_m$ is the $m \times m$ identity corresponding to the surplus variables, and the last column is the vector $\mathbf{b}$.

For ease of presentation, we will assume in the rest of Sections 5.3.2 and 5.3.3 that $k \leq n+1$. We will revisit this assumption at the end of Section 5.3.

The matrix composed of $A_1$ and $A_2$ will be referred to, as before, as $A_o$. Note that each row of $A_1$ has at most $w$ non-zero entries, and each row of $A_2$ has exactly one $+1$ and one $-1$ with the remaining entries 0. Thus, $A_2$ is TUM.

We prove the following theorem:

**Theorem 5.4** *The absolute value of any minor of* $[A|\mathbf{b}]$ *is bounded above by* $s\, b_{\max}\, (a_{\max}\, w)^k$, *where* $s = \min(n+1, m)$.

*Proof:*

Consider any minor $M$ of $[A|\mathbf{b}]$, and let $r$ be its order.

As in Theorem 5.3, if $M$ includes $p$ columns from the $-I_m$ part of $A$, then we can infer that $r - p \leq s$. (Our proof of this property in Theorem 5.3 made no assumptions on the form of $A_o$.)

If $M$ includes the last column $\mathbf{b}$, then as in the proof of Theorem 5.3, we can conclude that

$$|M| \;\leq\; (r-p)\, b_{\max}\, [\max_{j=1}^{r} |M_j|] \tag{5.5}$$

where $M_j$ is a minor of $A_o$.

If $M$ does not include $\mathbf{b}$, then it is a minor of $A$. Without loss of generality, we can assume that $M$ does not include a column from the $-I_m$ part of $A$, since such columns only contribute to the sign of the determinant.

So, let us consider bounding a minor $M_j$ of $A_o$ of order $r$ (or $r-1$, if $M$ includes the $\mathbf{b}$ column). Since $A_o = \left[\frac{A_1}{A_2}\right]$, consider expanding $M_j$, using the standard determinant expansion by minors along the top $k$ rows corresponding to non-difference constraints. Each term in the expansion is (up to a sign) the product of at most $k$ entries from the $A_1$ portion, one from each row, and a minor from $A_2$. Since $A_2$ is TUM, each product term is bounded in absolute value by ${a_{\max}}^k$. Furthermore, there can be at most $w^k$ non-zero terms in the expansion, since each non-zero product term is obtained by choosing one non-zero element from each of the rows of the $A_1$ portion of $M_j$, and this can be done in at most $w^k$ ways.

Therefore, $|M_j|$ is bounded by $(a_{\max}\, w)^k$. Combining this with the inequality (5.5), and since $r - p \leq s$, we get

$$|M| \;\leq\; s\, b_{\max}\, (a_{\max}\, w)^k$$

which is what we set out to prove. $\square$

Thus, we conclude that $\Delta \;\leq\; s\, b_{\max} (a_{\max}\, w)^k$, where $s \;=\; \min(n+1, m)$. From Theorems 5.1 and 5.4, and Remark 5.1, we obtain the following theorem:

**Theorem 5.5** *A solution bound for the system (5.2) is*

$$(n+2)\Delta = (n+2) \cdot s \cdot b_{\max} \cdot (a_{\max}\, w)^k$$

Thus, the solution size $S$ is

$$\lceil \log(n+2) + \log s + \log b_{\max} + k(\log a_{\max} + \log w) \rceil$$

**Remark 5.4** We make the following observations about the bound derived above, assuming as before, that $n < m$, and so $s = n + 1$:

- *Dependence on Parameters:* We observe that the bound is linear in $k$, logarithmic in $a_{\max}$, $w$, $n$, and $b_{\max}$. In particular, the bound is not in terms of the total number of linear constraints, $m$.

- *Worst-case Asymptotic Growth*: In the worst case, $k = m$, $w = n + 1$, and $n = \mathcal{O}(m)$, and we get the $\mathcal{O}(\log m + \log b_{\max} + m[\log m + \log a_{\max}])$ bound of Papadimitriou.

- *Typical-case Asymptotic Growth:* As observed in our study of formulas from software verification, $w$ is typically a small constant, so the number of bits needed per variable is $\mathcal{O}(\log n + \log b_{\max} + k \log a_{\max} + k)$. In many cases, $a_{\max}$ and $k$ are also bounded by a small constant. Thus, $S$ is typically $\mathcal{O}(\log n + \log b_{\max})$. This reduces the search space by an exponential factor over using the bound expressed in terms of $m$.

- *Representing Non-difference Constraints:* There are many ways to represent non-difference constraints and these have an impact on the bound we derive. In particular, it is possible to transform a system of non-difference constraints to one with at most three variables per constraint. For example, the linear constraint $x_1 + x_2 + x_3 + x_4 = x_5$ can be rewritten as:

  $$\begin{aligned} x_1 + x'_1 &= x_5 \\ x_2 + x'_2 &= x'_1 \\ x_3 + x_4 &= x'_2 \end{aligned}$$

  For the original representation, $k = 1$ and $w = 5$, while for the new representation $k = 3$ and $w = 3$. Since our bound is linear in $k$ and logarithmic in $w$, the original representation would yield a tighter bound.

  Similarly, one can eliminate variables with coefficients greater than 1 in absolute value by introducing new variables; e.g., $2x$ is represented as $x + x'$ with an additional difference constraint $x = x'$. This can be used to adjust $w$, $a_{\max}$, and $n$ so that the overall bound is reduced.

The derived bound only yields benefits in the case when the system has few non-difference constraints which themselves are sparse. In this case, we can instantiate variables over a finite domain that is much smaller than that obtained without making any assumptions on the structure of the system.

Finally, from Proposition 5.1 and Theorem 5.5, we obtain an enumeration bound for system (5.1):

**Theorem 5.6** *An enumeration bound for system (5.1) is*

$$(n + 2) \cdot s \cdot b_{\max} \cdot (a_{\max}\, w)^k$$

Note that the values of $a_{\max}$ and $w$ in the statement of Theorem 5.6 are those for system (5.2).

### 5.3.3 Bounds for Arbitrary Quantifier-Free Presburger Formulas

We now return to our original goal, that of finding a solution bound for an arbitrary QFP formula $F_{qfp}$.

Suppose that $F_{qfp}$ has $m$ linear constraints $\phi_1, \phi_2, \ldots, \phi_m$, of which $m - k$ are difference constraints, and $n$ variables $x_1, x_2, \ldots, x_n$. As before, we assume that each non-difference constraint has at most $w$ variables, $a_{\max}$ is the maximum over the absolute values of coefficients $a_{i,j}$ of variables, and $b_{\max}$ is the maximum over the absolute values of constants $b_i$ appearing in the constraints. Furthermore, let us assume that the zero variable (used in transforming system 5.1 to system 5.2) have already been introduced into the constraints, and that $a_{\max}$ and $w$ have been computed after this introduction.

We prove the following theorem.

**Theorem 5.7** $(n+2) \cdot \Delta$ *is a solution bound for* $F_{qfp}$ *where*

$$\Delta = s\,(b_{\max} + 1)\,(a_{\max}\,w)^k$$

*and* $s = \min(n+1, m)$.

*Proof:* Let $\sigma$ be an arbitrary satisfying assignment to $F_{qfp}$. Let $m'$ constraints, $\phi_{i_1}, \phi_{i_2}, \ldots, \phi_{i_{m'}}$, evaluate to $\mathtt{true}$ under $\sigma$, the rest evaluating to **false**. Let $A' = [a_{i,j}]$ be a $m' \times n$ matrix in which each row comprises the coefficients of variables $x_1, x_2, \ldots, x_n$ in a constraint $\phi_{i_k}$, $1 \le k \le m'$. Thus, $A' = [a_{i,j}]$ where $i \in \{i_1, \ldots, i_{m'}\}$.

Now consider a constraint $\phi_{i_k}$ where $k > m'$, that evaluates to **false** under $\sigma$. $\phi_{i_k}$ is the inequality

$$\sum_{j=1}^{n} a_{i_k,j} x_j \ge b_{i_k}$$

Then $\sigma$ satisfies $\neg\phi_{i_k}$ which is the inequality

$$\sum_{j=1}^{n} a_{i_k,j} x_j < b_{i_k}$$

or equivalently,

$$\sum_{j=1}^{n} -a_{i_k,j} x_j \ge -b_{i_k} + 1$$

Let $A''$ be a $(m - m') \times n$ matrix corresponding to the coefficients of variables in constraints $\neg\phi_{i_{m'+1}}, \neg\phi_{i_{m'+2}}, \ldots, \neg\phi_{i_m}$. Thus, $A'' = [-a_{i,j}]$ where $i \in \{i_{m'+1}, \ldots, i_m\}$.

Finally, let $\mathbf{b} = [b_{i_1}, b_{i_2}, \ldots, b_{i_{m'}}, -b_{i_{m'+1}} + 1, -b_{i_{m'+2}} + 1, \ldots, -b_{i_m} + 1]^T$

Clearly, $\sigma$ is a satisfying solution to the ILP given by

$$\left[\begin{array}{c} A' \\ \hline A'' \end{array}\right] \mathbf{x} \geq \mathbf{b} \tag{5.6}$$

Also, if the system (5.6) has a satisfying solution then $F_{qfp}$ is satisfied by that solution. Thus, $F_{qfp}$ and the system (5.6) are equi-satisfiable, for every possible system (5.6) we construct in the manner described above.

By Theorems 5.1 and 5.4, we can conclude that if system (5.6) has a satisfying solution, it has one bounded by $(n+2)\Delta$ where

$$\Delta = s\,(b_{\max}+1)\,(a_{\max}\,w)^k$$

and $s = \min(n+1, m)$. Moreover, this bound works for every possible system (5.6).

Therefore, if $F_{qfp}$ has a satisfying solution, it has one bounded by $(n+2)\Delta$. $\Box$

Thus, to generate the Boolean encoding of the starting QFP formula, we must encode each integer variable as a symbolic bit-vector of length $S$ given by

$$S = \lceil \log[(n+2)\Delta] \rceil = \lceil \log(n+2) + \log s + \log(b_{\max}+1) + k(\log a_{\max} + \log w) \rceil$$

**Remark 5.5** If the zero variable is not introduced into the formula $F_{qfp}$, we can search for solutions in $\prod_{i=1}^{n}[-d, d]$, where $d = (n+2)\Delta$. As noted earlier, values of $a_{\max}$ and $w$ used in computing $\Delta$ are those obtained after introducing the zero variable.

**Remark 5.6** In Section 5.3.2, we assumed, for ease of presentation, that $k \leq n+1$. If this does not hold, we can simply replace $k$ in the results of Sections 5.3.2 and 5.3.3 by $\min(k, n+1)$. This is because the dimension of the minor $M_j$ of $A_o$ (mentioned in the proof of Theorem 5.4) is limited by $n+1$.

**Remark 5.7** Let us specialize the derived solution bound for G2SAT formulas. Since, $w \leq 2$, $a_{\max} = 1$, the bound specializes to $(n+2)\,s\,(b_{\max}+1)\,2^k$. This indicates that the derived bound is conservative.

### Summary of notation

We conclude this section by summarizing the symbols used to represent formula parameters and the quantities derived therefrom. For easy reference, they are listed in Table 5.2.

| *Symbol* | *Meaning* |
|---|---|
| $n$ | Number of variables |
| $m$ | Number of constraints |
| $b_{\max}$ | Maximum constant term |
| $a_{\max}$ | Maximum variable coefficient |
| $k$ | Number of non-difference constraints |
| $w$ | Maximum number of non-zero coefficients in any constraint |
| $s$ | $\min(n+1, m)$ |
| $\Delta$ | $s \cdot (b_{\max}+1) \cdot (a_{\max}\, w)^k$ |
| $d$ | Solution bound, $(n+2)\Delta$ |
| $S$ | Solution size, $\lceil \log(d+1) \rceil$ |

Table 5.2: **Parameters and derived quantities**

## 5.4 Improvements

The bounds we derived in the preceding section are conservative. For a particular problem instance, the size of minors can be far smaller than the bound we computed. However, this cannot be directly exploited by enumerating minors, since the number of minors grows exponentially with the dimensions of the constraint matrix. Also, there is a special case under which one can improve the $(n+2)\Delta$ bound. If all the constraints are originally linear equalities and the system of constraints has full rank, a bound of $\Delta$ suffices [20]. However, in our experience, even if the linear constraints are all equalities, they still tend to be linearly dependent. Thus, we have not been able to make use of this special case result.

Fortunately, there are other techniques for improving the solution bound that we have found to be fairly useful in practice. These include theoretical improvements as well as heuristics that are useful in practice. We describe these methods in this section.

### 5.4.1 Variable Classes

Recall the notion of a variable class introduced in Section 2.2. The variables and constraints in a QFP formula can usually be partitioned into several classes. Parameters $n$, $k$, $b_{\max}$, $a_{\max}$, and $w$ can be separately computed for each variable class, resulting in a separately computed solution bound for each class.

The correctness of this optimization follows from a reduction to ILP as performed in the proof of Theorem 5.7, and the observation that a satisfying solution to a system of ILPs, no two of which share a variable, can be obtained by solving them independently and concatenating the solutions.

Moreover, if all the constraints in a variable class are difference constraints (or G2SAT constraints), one can use the tighter solution bounds derived in Chapters 3 and 4.

### 5.4.2 Large Coefficients and Widths

In the expression for $S$, the term involving $a_{\max}$ (and $w$) is multiplied by a factor of $k$. Thus, any increase in $\log a_{\max}$ gets amplified by a factor of $k$. It is therefore useful to more carefully model the dependence of $S$ on coefficients. We present two techniques to alleviate the problem of dealing with large coefficients. These techniques also apply to dealing with large constraint widths.

#### An $n^k$-fold reduction

The coefficient of the zero variable $x_0$ has, so far, been used in computing $a_{\max}$. We will now show that we can ignore this coefficient, and also ignore any contribution of $x_0$ to the width $w$. This optimization can result in a reduction of up to a factor of $n^k$ in the solution bound $d$.

The largest reduction occurs when, in the original formula, we have a constraint of the form $\sum_j a_i x_j \geq b_i$, where $a_i$ is the largest coefficient in absolute value. After adding the zero variable, this constraint is transformed to $(\sum_j a_i x_j) - (n \cdot a_i)x_0 \geq b_i$. Thus, $a_{\max}$ now equals $n \cdot a_i$, a factor of $n$ times greater than in the original formula.

Let us revisit the transformation performed in Section 5.2.1 to convert system (5.1) to system (5.2). A different, commonly-used transformation to non-negative variables is to write each $x_j$ as $x_j^+ - x_j^-$, where $x_j^+, x_j^- \geq 0$ for all $j$. Let the resulting system be referred to as system (5.2'). Let us assume that this different transformation is used in place of the original one that generates system (5.2), leaving all successive transformations the same.

Now, consider the form of the matrix $[A|\mathbf{b}]$, as used in Section 5.3.2, reproduced below:

$$\left[\begin{array}{c|c|c} A_1 & & b_1 \\ \cline{1-1} & -I_m & b_2 \\ A_2 & & \vdots \\ & & b_m \end{array}\right]$$

With the new transformation method, $A_1$ is a $k \times 2n$ dimensional matrix corresponding to the non-difference constraints, $A_2$ is a $(m-k) \times 2n$ dimensional matrix with the difference constraints, $I_m$ is the $m \times m$ identity corresponding to the surplus variables, and the last column is the vector $\mathbf{b}$.

Importantly, note that $A_2$ is *still* totally unimodular and the ranks of $A_1$ and $A_2$ are the same as they were with the use of the single zero variable $x_0$. This is because any non-singular sub-matrix of $A_o$

must include exactly one of the columns corresponding to $x_j^+$ and $x_j^-$, since they are negations of each other. Therefore, the values of $w$ and $a_{\max}$ used in the proof of Theorem 5.4 are those for the system (5.1).

Thus, if we use the transformation method of replacing $x_j$ with $x_j^+ - x_j^-$, the values of $w$ and $a_{\max}$ used in the statement of Theorem 5.4 are those for the system (5.1).

Note, however, that by replacing $x_j$ with $x_j^+ - x_j^-$, the number of variables in the problem doubles, and in particular, the number of input variables in the SAT-encoding is doubled. This is rather undesirable.

Fortunately, there are two solutions that avoid the doubling of variables at the minor cost of only 1 extra bit per variable.

1. The first solution is based on the following proposition that mirrors Proposition 5.1.

   **Proposition 5.2** *A solution bound $d \geq 0$ for system (5.2') is an enumeration bound for system (5.1).*

   *Proof:* Given a solution $\mathbf{x}'^*$ within the solution bound $d$ to system (5.2'), we construct a solution $\mathbf{x}^*$ to system (5.1) by setting $x_j^* = x_j^{+*} - x_j^{-*}$. Clearly, $x_j^* \in [-d, d]$ for all $j$. □

   Thus, we can restrict our search to the hypercube $\prod_{i=1}^{n}[-d, d]$, where the solution bound $d$ is computed using the values of $w$ and $a_{\max}$ for the system (5.1).

2. The second solution uses the following proposition showing that we can use the technique of adding a zero variable $x_0$ and the values of $w$ and $a_{\max}$ for the system (5.1), while paying only a minor penalty of 1 extra bit per variable.

   **Proposition 5.3** *Suppose $d \geq 0$ is a solution bound such that system (5.2') has a solution in $[0, d]$ iff system (5.1) is feasible. Then, system (5.2) has a solution in $[0, 2d]$ iff system (5.2') has a solution in $[0, d]$.*

   *Proof:*

   *(if part)*: Suppose system (5.2') has a solution in $[0, d]$; i.e., $x_j^+, x_j^- \in [0, d]$ for all $j$. Then, we construct a satisfying assignment to system (5.2) as follows:

   - $x_0$ is assigned the value $\max_j x_j^-$.
   - $x_j$, for $j > 0$, is assigned the value $x_j^+ + (x_0 - x_j^-)$.

   Since $0 \leq (x_0 - x_j^-) \leq d$, we can conclude that $0 \leq x_j \leq 2d$ for all $j$. It is easy to see that the resulting assignment satisfies system (5.2).

*(only if part)*: Suppose system (5.2) has a solution in $[0, 2d]$. This means that the original system (5.1) is feasible. It follows that system (5.2') has a solution in $[0, d]$.

□

In both solutions, we must search $2d + 1$ values for each variable $x_j$, $1 \leq j \leq n$. However, the former avoids the need to add $x_0$, and hence will have fewer input variables in the SAT-encoding. Hence, the former solution is preferable.

The reader must note, though, that this optimization is only relevant when the introduction of the zero variable (significantly) affects the value of $a_{\max}$. (The impact on $w$ is minor.) If the value of $a_{\max}$ is unaffected by the introduction of the zero variable $x_0$, using $x_0$ can result in a more compact SAT-encoding than using an enumeration domain of $[-d, d]$ for each variable. If one uses the $x_0$ variable, one introduces $\log d$ input Boolean variables for $x_0$ in the SAT-encoding. On the other hand, without the $x_0$ variable, one introduces $n$ additional Boolean variables to encode sign bits. The relative size of the SAT-encoding, and hence the decision to introduce $x_0$, would depend on whether $n$ exceeds $\log d$.

#### Product of $k$ largest coefficients and widths

There is a simpler optimization which we have found to be useful in practice.

In the proof of Theorem 5.4, in deriving the $(a_{\max} \cdot w)^k$ term, we have assumed the worst-case scenario of each term in the determinant expansion equaling ${a_{\max}}^k$ and there being $w$ terms to choose from in each row.

In fact, we can replace ${a_{\max}}^k$ with $\prod_{i=1}^{k} a_{\max i}$, where $a_{\max i}$ denotes the largest coefficient in row $i$, in absolute value. Similarly, $w^k$ can be replaced with $\prod_i w_i$, where $w_i$ is the width of constraint $i$.

### 5.4.3 Large Constant Terms

For some formulas, the value of $b_{\max}$ is very large due to the presence of a single large constant (or very few of them). In such cases, a less conservative analysis or other problem transformations are useful. We present two such techniques here.

#### Product of $s$ largest constants

It is easy to see that, in the proof of Theorem 5.4, the $s\, b_{\max}$ term can be replaced by $\sum_{j=1}^{s} |b_{i_j}|$, where $b_{i_1}, b_{i_2}, \ldots, b_{i_s}$ are the $s$ largest elements of $b$ in absolute value. Similarly, the expression for

$\Delta$ derived in Theorem 5.7 gets modified to

$$\Delta = \left(\sum_{j=1}^{s}(|b_{i_j}| + 1)\right) \cdot (a_{\max}\, w)^k$$

This optimization, like that of Section 5.4.2, has also proved fairly useful in practice.

**Shift of origin**

Another transformation that can be useful for dealing with large constant terms is to replace a variable $x_j$ by $x_j - \alpha_j$; this corresponds to shifting the origin in $\mathbb{R}^n$ by $\alpha_j$ along the $x_j$-axis.

The $i^{\text{th}}$ constraint is then transformed into $\sum_j a_{i,j}(x_j - \alpha_j) \geq b_i$. Rewriting this, we obtain the form $\sum_j a_{i,j}x_j \geq b'_i$, where $b'_i = b_i + (\sum_j a_{i,j}\alpha_j)$.

The new value of $b_{\max}$, after the transformation, is $\max_i |b'_i|$. Therefore, we wish to find values of $\alpha_j$s so as to minimize the value of $\max_i |b'_i|$.

This problem can be phrased as the following integer linear program:

$$\begin{array}{lcll}
 & \min z & & \\
\text{subject to} & & & \\
z & \geq & b_i + (\sum_j a_{i,j}\alpha_j) & 1 \leq i \leq m \\
z & \geq & -b_i - (\sum_j a_{i,j}\alpha_j) & 1 \leq i \leq m \\
z & \geq & 0 & \\
 & & z \in \mathbb{Z}, \quad \alpha_j \in \mathbb{Z} \text{ for } 1 \leq j \leq n &
\end{array}$$

The above ILP has $n + 1$ variables and $2m + 1$ constraints (including the non-negativity constraint on $z$).

In fact, one can write one such ILP for each variable class, since they do not share any variables or constraints. Then, the optimum value for each class will indicate the new value of $b_{\max}$ to use for that class.

## 5.5 Experimental Evaluation

We used the bound derived in the previous section to implement a decision procedure based on small-domain encoding. We describe the implementation decisions in Section 5.5.1 and present a detailed experimental evaluation in Section 5.5.2.

### 5.5.1 Implementation

The decision procedure starts by analyzing the formula to obtain parameters, and computes the solution bound. We found that the optimizations of Section 5.4.1 and the first half of Section 5.4.2 are always useful, especially since formulas tend to contain many variables classes comprising of only difference constraints. Hence, our base-line implementation always includes these optimizations. The impact of other optimizations is studied in Section 5.5.2.

Given the solution bound defining a finite range of values, integer variables in the QFP formula are encoded as symbolic bit-vectors (in twos complement encoding) large enough to express any integer value within that range. Arithmetic operators are implemented as arbitrary-precision bit-vector arithmetic operations. In our implementation, we used a ripple-carry adder circuit for encoding the "$+$" and "$-$" operators, a shift-and-add circuit to encode multiplication by a constant. Equalities and inequalities over integer expressions are translated to comparator circuits over bit-vector expressions. The resulting Boolean formula is passed as input to a SAT solver.

We implemented our procedure as part of the UCLID verifier [156], which is written in Moscow ML [103]. In our implementation we used the zChaff SAT solver [169] version 2003.7.22. In the sequel, we will refer to our decision procedure as the "UCLID" procedure.

### 5.5.2 Experimental Results

We report here on a series of experiments we performed to evaluate our decision procedure against other theorem provers, as well as to assess the impact of the various optimizations discussed in Section 5.4.

All experiments were performed on a Pentium-IV 2 GHz machine with 1 GB of RAM running Linux. A timeout of 3600 seconds (1 hour) was imposed on each run.

#### Benchmarks

For benchmarks, we used 10 formulas from the Wisconsin Safety Analyzer (WiSA) project on checking format string vulnerabilities, and 3 generated by the Blast software model checker. The benchmarks include both satisfiable and unsatisfiable formulas in an extension of QFP with uninterpreted functions. Uninterpreted functions were first eliminated using Ackermann's technique [2],[4] and the decision procedures were run on the resulting QFP formula.

[4] Ackermann's function elimination method replaces each function application by a fresh variable, and then instantiates the congruence axiom for those applications. For instance, the formula $f(x) = f(y)$ is translated to the function-free formula $v_{f_1} = v_{f_2} \wedge (x = y \implies v_{f_1} = v_{f_2})$.

Some characteristics of the formulas are displayed in Table 5.3. For each formula, we indicate whether it is satisfiable or not. We give the values of parameters $n$, $m$, $k$, $w$, $a_{\max}$ and $b_{\max}$ corresponding to the variable class for which $S = \lceil \log[(n+2)\Delta] \rceil$ is largest, i.e, for which we need the largest number of bits per variable. The values of the parameters for the overall formula are also given (although these are not used in computing $S$ for any variable class); thus, the values of $m$ and $n$ in these columns are the *total* numbers of variables and constraints for the entire formula.

The top 10 formulas listed in the table are from the WiSA project. One key characteristic of these formulas is that they involve a significant number of Boolean operators ($\wedge$, $\vee$, $\neg$), and in particular there is a lot of alternation of $\wedge$ and $\vee$. The other important characteristic of these benchmarks is that, although they vary in $n$, $m$, and $b_{\max}$, the values of $k$, $w$, and $a_{\max}$ are fixed at a small value.

Three formulas from the Blast suite are listed at the bottom of Table 5.3. All these formulas are unsatisfiable. Each formula is a conjunction of two sub-formulae: a large conjunction of linear constraints, and a conjunction of congruence constraints generated by Ackermann's function elimination method. Thus, there is only one alternation of $\wedge$ and $\vee$ in these formulas.

| Formula | Ans. | Parameters corr. to max. $S$ | | | | | | | Max. parameters overall | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $n$ | $m$ | $k$ | $w$ | $a_{\max}$ | $b_{\max}$ | $S$ | $n$ | $m$ | $k$ | $w$ | $a_{\max}$ | $b_{\max}$ |
| s-20-20 | SAT | 28 | 263 | 5 | 4 | 4 | 21 | 36 | 64 | 550 | 5 | 4 | 4 | 255 |
| s-20-30 | SAT | 28 | 263 | 5 | 4 | 4 | 30 | 36 | 64 | 550 | 5 | 4 | 4 | 255 |
| s-20-40 | UNS | 28 | 263 | 5 | 4 | 4 | 40 | 37 | 64 | 550 | 5 | 4 | 4 | 255 |
| s-30-30 | SAT | 38 | 383 | 5 | 4 | 4 | 31 | 37 | 82 | 800 | 5 | 4 | 4 | 255 |
| s-30-40 | SAT | 38 | 383 | 5 | 4 | 4 | 40 | 37 | 82 | 800 | 5 | 4 | 4 | 255 |
| xs-20-20 | SAT | 49 | 323 | 5 | 4 | 4 | 21 | 37 | 84 | 632 | 5 | 4 | 4 | 255 |
| xs-20-30 | SAT | 49 | 323 | 5 | 4 | 4 | 30 | 38 | 84 | 632 | 5 | 4 | 4 | 255 |
| xs-20-40 | UNS | 49 | 323 | 5 | 4 | 4 | 40 | 38 | 84 | 632 | 5 | 4 | 4 | 255 |
| xs-30-30 | SAT | 69 | 473 | 5 | 4 | 4 | 31 | 39 | 114 | 922 | 5 | 4 | 4 | 255 |
| xs-30-40 | SAT | 69 | 473 | 5 | 4 | 4 | 40 | 39 | 114 | 922 | 5 | 4 | 4 | 255 |
| blast-tl2 | UNS | 54 | 67 | 7 | 3 | 1 | 0 | 24 | 145 | 274 | 7 | 3 | 1 | 128 |
| blast-tl3 | UNS | 201 | 2669 | 19 | 6 | 1 | 15 | 70 | 260 | 2986 | 19 | 6 | 1 | 128 |
| blast-f8 | UNS | 255 | 6087 | 0 | 2 | 1 | 2560 | 20 | 321 | 7224 | 0 | 2 | 1 | 2560 |

Table 5.3: **Benchmark characteristics.** The top half of the table consists of the WiSA benchmarks and the bottom three are generated by the Blast software verifier.

#### Impact of optimizations

In this section, we discuss the impact of optimizations discussed in Sections 5.4.2 and 5.4.3.

Table 5.4 compares the following $4$ different encoding options based on different ways of computing the solution bound:

**Base**: The base-line method of computing the solution bound.

**Coeff**: Using the optimization of Section 5.4.2 alone.

**Const**: Using the optimization of Section 5.4.3 alone.

**All**: Using optimization methods of both Sections 5.4.2 and 5.4.3.

The comparison is made with respect to the largest number of bits needed for any variable class, and the run-times for both generating the SAT-encoding and for SAT solving.

First, we note that Coeff and Const both generate more compact encodings than Base; on the WiSA benchmarks, they use about 5-10 fewer bits per variable in the largest variable class. The reduction in the total number of bits, summed over all variables in all variable classes, is similar, since most variables fall into a single class.

The encoding times decrease with reduction in number of bits; this is just as one would expect.

However, the comparison of SAT solving times is more mixed; on a few benchmarks Coeff and Const outperform Base, and on others, they do worse. The latter behavior is observed especially on satisfiable formulas. The reason for this might be the relative ease in finding larger solutions for those formulas than finding smaller solutions.

When Coeff and Const are both used (indicated as "All"), we find that not only are encoding times smaller than the Base technique, but SAT solving times are also smaller in all cases except one, where the difference is only minor. This seems to indicate that a reduction in SAT-encoding size beyond a certain limit overcomes any negative effects of restricting the search to smaller solutions.

We also performed an experiment to explore the use of the *shift-of-origin* optimization described in Section 5.4.3. UCLID automatically formulated the ILP and invoked CPLEX [46], an integer linear programming solver (version 8.1), to solve it. Since none of the benchmarks listed in Table 5.3 have especially large constants, we used a different, unsatisfiable formula from the Blast suite which has only difference constraints, but with large constants.

Table 5.5 summarizes the key characteristics of this formula as well as the results obtained by comparing versions of the base-line (Base) implementation with and without the optimization enabled. We list the values of parameters, with and without the shift-of-origin optimization enabled, for the variable classes that yield the two largest values of $S$ when the optimization is disabled.

With the optimization turned on, the largest constant in the *entire* formula falls from $261133242$ to $432539$, a 600-fold reduction. However, if we restrict our attention to the largest variable class,

| Formula | Ans. | Max. #bits/var. | | | | Encoding Time (sec.) | | | | SAT Time (sec.) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Base | Coeff | Const | All | Base | Coeff | Const | All | Base | Coeff | Const | All |
| s-20-20 | SAT | 36 | 26 | 31 | 21 | 1.66 | 1.25 | 1.27 | 1.00 | 5.41 | 9.28 | 8.34 | 0.48 |
| s-20-30 | SAT | 36 | 26 | 31 | 22 | 1.72 | 1.24 | 1.32 | 1.02 | 3.99 | 2.28 | 4.82 | 0.50 |
| s-20-40 | UNS | 37 | 27 | 32 | 22 | 1.72 | 1.28 | 1.30 | 1.03 | 1.37 | 1.35 | 0.92 | 0.87 |
| s-30-30 | SAT | 37 | 27 | 32 | 22 | 2.27 | 1.90 | 1.99 | 1.57 | 17.22 | 0.88 | 14.31 | 9.57 |
| s-30-40 | SAT | 37 | 28 | 32 | 23 | 2.39 | 1.96 | 2.03 | 1.55 | 20.17 | 8.22 | 4.80 | 11.99 |
| xs-20-20 | SAT | 37 | 28 | 32 | 22 | 2.29 | 1.88 | 1.93 | 1.55 | 17.67 | 21.62 | 11.67 | 7.15 |
| xs-20-30 | SAT | 38 | 28 | 32 | 23 | 2.29 | 1.95 | 2.00 | 1.61 | 23.21 | 18.19 | 1.50 | 7.18 |
| xs-20-40 | UNS | 38 | 29 | 33 | 23 | 2.41 | 1.99 | 2.04 | 1.59 | 7.32 | 8.60 | 10.55 | 8.01 |
| xs-30-30 | SAT | 39 | 29 | 33 | 23 | 3.84 | 2.71 | 2.89 | 2.17 | 79.10 | 18.40 | 20.16 | 27.92 |
| xs-30-40 | SAT | 39 | 30 | 33 | 24 | 3.76 | 2.83 | 2.67 | 2.12 | 27.60 | 45.63 | 13.36 | 13.45 |
| blast-tl2 | UNS | 24 | 24 | 19 | 19 | 1.54 | 1.48 | 1.10 | 1.08 | 0.05 | 0.04 | 0.03 | 0.03 |
| blast-tl3 | UNS | 70 | 53 | 62 | 46 | 29.98 | 19.34 | 22.50 | 17.57 | 0.78 | 0.54 | 0.66 | 0.46 |
| blast-f8 | UNS | 20 | 20 | 12 | 12 | 18.37 | 17.99 | 10.71 | 10.68 | 6.22 | 6.15 | 2.63 | 2.29 |

Table 5.4: **An experimental evaluation of encoding optimizations.** We compare the $4$ different UCLID encoding options with respect to the maximum number of bits needed for any integer variable (“Max. #bits/var.”), the time taken to generate the Boolean encoding, and the time taken by the SAT solver.

comprising $230$ variables, the reduction in $b_{\max}$ is more modest, about a factor of $4$. This yields a saving of 2 bits per variable for that variable class. The saving in the total number of bits, summed over all variable classes, is $677$. This is, however, not large enough to reduce either the encoding time or the SAT time. In fact, the encoding time increases by about a second; this is the time required to run CPLEX and for the processing overhead of creating the ILP.

Even though the shift-of-origin optimization has not resulted in faster run-times in our experiments, it clearly has the potential to greatly reduce the number of bits, and might prove useful on other benchmarks.

### Comparison with other theorem provers

We compared UCLID’s performance with that of the SAT-based provers ICS [80] (version 2.0) and CVC-Lite [48] (the new implementation of CVC, version 1.1.0), as well as the automata-based procedure LASH [92]. While CVC-Lite and LASH are sound and complete for QFP, ICS 2.0 is incomplete; i.e., it can report a formula to be satisfiable when it is not. The ground decision procedure ICS uses is the Simplex linear programming algorithm with some additional heuristics to deal with integer variables. However, in our experiments, both UCLID and ICS returned the same

| Shift-of-origin enabled? | Param. for largest $S$ | | | | Param. for $2^{\text{nd}}$ largest $S$ | | | | Total #bits | Time (sec.) | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | $n$ | $m$ | $b_{\max}$ | $S$ | $n$ | $m$ | $b_{\max}$ | $S$ | | Enc. | SAT |
| No | 230 | 6417 | 2162688 | 29 | 2 | 2 | 261133242 | 28 | 7510 | 24.68 | 0.70 |
| Yes | 230 | 6417 | 432539 | 27 | 2 | 2 | 0 | 1 | 6833 | 25.78 | 0.71 |

Table 5.5: **Evaluating the shift-of-origin optimization.** We list the values of parameters corresponding to variable classes with the two largest values of $S$, as computed *without* the shift-of-origin optimization. "Total #bits" indicates the number of bits needed to encode all integer variables. Encoding time is indicated as "Enc." and SAT solving time as "SAT".

answer whenever ICS terminated within the timeout. The ground decision procedure for CVC-Lite is a proof-producing variant of the Omega test [17].

LASH was unable to complete on any benchmark within the timeout since it was unable to construct the corresponding automata; we attribute this to the relatively large number of variables and constraints in our formulas, and note that Ganesh et al. obtained similar results in their study [59].

A comparison of UCLID versus ICS and CVC-Lite is displayed in Table 5.6. From Table 5.6, we observe that UCLID outperforms ICS on all the WiSA benchmarks, terminating well within a minute on each one. However, ICS performs best on the Blast formulas, finishing within a fraction of a second on all. CVC-Lite does not outperform the other procedures on any formula, and was unable to complete on any of the WiSA benchmarks. We suspect that this time is being mainly spent in the ground decision procedure based on the Omega test, but have been unable to obtain detailed statistics.

Let us consider the WiSA benchmarks first. These formulas have a complicated Boolean structure that requires ICS to enumerate many inconsistent Boolean assignments before being able to decide the formula. The ICS run-time is dominated by the time taken by the ground decision procedure. We observe that the number of inconsistent Boolean assignments alone is not a precise indicator of total run-time, which also depends on the time taken by the ground decision procedure in ruling out a single Boolean assignment.

The reason for UCLID's superior performance is the formula structure, where $k$, $w$, and $a_{\max}$ remain fixed at a low value while $m$, $n$, and $b_{\max}$ increase. Thus, the maximum number of bits per variable stays about the same even as $m$ increases substantially, and the resulting SAT problem is within the capacity of zChaff. Also, for these benchmarks, the SAT time is almost always the larger portion of UCLID's run-time; this is not surprising since Boolean structure of the original formula is non-trivial, and moreover, the time to compute the parameter values and generate the SAT-encoding is polynomial in the input size.

Next, consider the results on the Blast formulas. The reason for ICS's superior performance on

| Formula | Ans. | UCLID Time (sec.) | | | ICS | | | CVC-Lite |
|---|---|---|---|---|---|---|---|---|
| | | | | | #(Inc. assn.) | Time (sec.) | | Total Time (sec.) |
| | | Enc. | SAT | Total | | Ground | Total | |
| s-20-20 | SAT | 1.13 | 1.02 | 2.15 | 904 | 23.32 | 23.76 | * |
| s-20-30 | SAT | 1.17 | 1.02 | 2.19 | 1887 | 51.68 | 52.29 | * |
| s-20-40 | UNS | 1.16 | 1.35 | 2.51 | 25776 | 658.01 | 669.99 | * |
| s-30-30 | SAT | 1.73 | 11.12 | 12.85 | 2286 | 268.21 | 269.42 | * |
| s-30-40 | SAT | 1.77 | 13.81 | 15.58 | 14604 | 1621.27 | 1625.15 | * |
| xs-20-20 | SAT | 1.63 | 8.38 | 10.01 | 2307 | 97.21 | 98.32 | * |
| xs-20-30 | SAT | 1.50 | 7.22 | 8.72 | 33103 | 1519.77 | 1540.27 | * |
| xs-20-40 | UNS | 1.65 | 8.84 | 10.49 | 97427 | 3468.91 | * | * |
| xs-30-30 | SAT | 2.26 | 29.73 | 31.99 | 72585 | 3287.47 | * | * |
| xs-30-40 | SAT | 2.32 | 15.65 | 17.97 | 33754 | 3082.34 | * | * |
| blast-tl2 | UNS | 1.08 | 0.03 | 1.11 | 1 | 0.01 | 0.01 | 1.38 |
| blast-tl3 | UNS | 17.57 | 0.46 | 18.03 | 0 | 0.00 | 0.01 | 37.77 |
| blast-f8 | UNS | 10.68 | 2.29 | 12.97 | 1 | 0.01 | 0.05 | 179.43 |

Table 5.6: **Experimental comparison with other theorem provers.** The UCLID version is the one with all optimizations turned on (“All”). For ICS, we give the total time, the number of inconsistent Boolean assignments analyzed by the ground decision procedure (“#(Inc. assn.)”), as well as the overall time taken by the ground decision procedure (“Ground”). For CVC-Lite, we indicate the total run-time. A “$*$” indicates that the decision procedure timed out after 3600 sec. LASH did not complete within the timeout on any formula.

these can be gauged by the number of inconsistent Boolean assignments it has to enumerate. On the formula named “blast-tl3”, purely Boolean reasoning suffices to decide unsatisfiability. For the other two formulas, the reason for unsatisfiability is a mutually-inconsistent subset amongst all the linear constraints that are conjoined together, and a single call to ICS’s ground decision procedure suffices to infer the inconsistency.

On the other hand, UCLID’s run-time is dominated by the encoding time. Once the encoding is generated, the SAT solver decides unsatisfiability easily.

To summarize, it appears that decision procedures based on a lazy translation to SAT, such as ICS, are effective when the formula structure is such that only a few calls to the ground decision procedure are required. UCLID performs better on formulas with complicated Boolean structure and comprising linear constraints with the sparse structure formalized in this chapter.

## 5.6 Discussion

We have presented a formal approach to exploiting the "sparse, mainly difference constraint" nature of quantifier-free Presburger formulas encountered in software verification. Our approach is based on formalizing this sparse structure using new parameters, and deriving a new parameterized bound on satisfying solutions to QFP formulas. We have also proposed several ways in which the bound can be reduced in practice. Experimental results show the benefits of using the derived bound in a SAT-based decision procedure based on small-domain encoding.

Table 5.7 summarizes the value of $d$ for all the classes of linear constraints explored in this thesis. We can clearly see that the bound derived in this chapter for quantifier-free Presburger arithmetic is

| Class of Linear Constraints | Solution Bound $d$ |
|---|---|
| Difference constraints | $n \cdot (b_{\max} + 1)$ |
| Generalized 2SAT constraints | $2 \cdot n \cdot (b_{\max} + 1)$ |
| Arbitrary linear constraints | $(n + 2) \cdot \min(n + 1, m) \cdot (b_{\max} + 1) \cdot (w \cdot a_{\max})^k$ |

Table 5.7: **Solution bounds for classes of linear constraints.** The classes are listed top to bottom in increasing order of expressiveness.

conservative. For example, if all constraints are difference constraints, the expression for $d$ derived in this chapter simplifies to $(n + 2) \cdot \min(n + 1, m) \cdot (b_{\max} + 1)$. This is $n + 2$ times as big as the bound derived in Chapter 3; note that the looseness in the bound is a carry-over from the result of Borosh, Treybig, and Flahive. For generalized 2SAT constraints, the bound derived for arbitrary QFP is much looser. In the worst case, it is looser by an exponential factor: if $k$ is $\mathcal{O}(m)$, $a_{\max}$ is 1, and $w$ is 2, then the bound is $\mathcal{O}((n + 2) \cdot \min(n + 1, m) \cdot (b_{\max} + 1) \cdot 2^m)$, whereas the results of Chapter 4 tell us that the solution bound $d = 2 \cdot n \cdot (b_{\max} + 1)$ suffices (since $n \cdot (b_{\max} + 1)$ is an enumeration bound).

Due to the conservative nature of the bound derived in this chapter, and in spite of the many optimizations discussed, the computed solution bound can generate a SAT problem beyond the reach of current solvers. The latter situation can also arise for problem domains that do not generate sparse linear constraints. There is therefore a need for an efficient algorithm to compute a tighter solution bound.

Recent work by the author and colleagues [87], implemented in UCLID, presents one approach towards computing a tighter solution bound. The central idea is to compute the solution bound incrementally, starting with a small bound and increasing it "on demand". Figure 5.1 outlines this *lazy* approach to computing the solution bound.

Given a QFP formula $F_{qfp}$, we start with an encoding size for each integer variable that is smaller

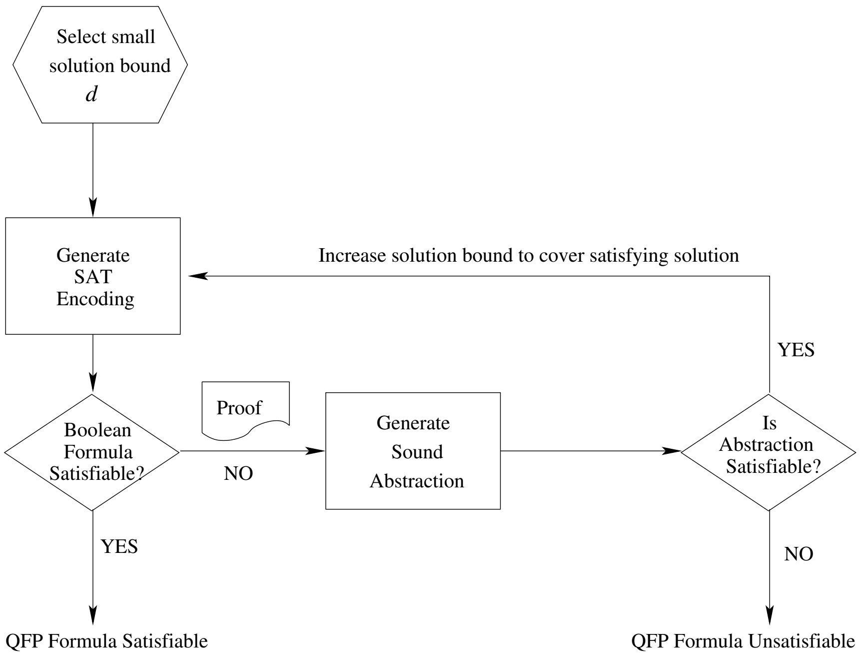



Figure 5.1: **Lazy approach to computing solution bound**

than that prescribed by the conservative bound (say, for example, 1 bit per variable).

If the resulting Boolean formula is satisfiable, so is $F_{qfp}$. If not, the proof of unsatisfiability generated by the SAT solver is used to generate a *sound abstraction* $F'_{qfp}$ of $F_{qfp}$. A sound abstraction is a formula, usually much smaller than the original, such that if it is unsatisfiable, so is the original formula. A sound and complete decision procedure for QFP (such as the one proposed in this chapter) is then used on $F'_{qfp}$. If this decision procedure concludes that $F'_{qfp}$ is unsatisfiable, so is $F_{qfp}$. If not, it provides a counterexample which indicates the necessary increase in the encoding size. A new SAT-encoding is generated, and the procedure repeats.

The bound $S$ on solution size that we derive in this chapter implies an upper bound $nS$ on the number of iterations of this lazy encoding procedure; thus the lazy encoding procedure needs only polynomially many iterations before it terminates with the correct answer. Of course, each iteration involves a call to a SAT solver as well as to a decision procedure for QFP.

A key component of this lazy approach is the generation of the sound abstraction. While the details are outside the scope of this thesis, we sketch one approach here. (Details can be found in [87].) Assume that $F_{qfp}$ is in conjunctive normal form (CNF); thus, $F_{qfp}$ can be viewed as a set of clauses, each of which is a disjunction of linear constraints and Boolean literals. A subset of this set of clauses is a sound abstraction of $F_{qfp}$. This subset is computed by retaining only those clauses from the original set that contribute to the proof of unsatisfiability of the SAT-encoding.

The potential advantage of this lazy approach is twofold: (1) It avoids using the conservative bounds we have derived in this chapter, and (2) if the generated abstractions are small, the sound and complete decision procedure used by this approach will run much faster than if it were fed the original formula.

For the WiSA benchmarks discussed in Section 5.5, we found that a solution bound of $2^8 - 1$, i.e., $8$ bits per variable, is sufficient to decide satisfiability. However, the time required to derive this bound using the method of [87] is much greater than the run-times we report in Section 5.5. Still, the lazy approach can prove especially useful in cases in which $S$ is so large that the SAT problem is outside the reach of current SAT solvers. Among other things, there is potential to improve its efficiency by using an incremental SAT solver in the loop.

# Chapter 6

# Automated Selection of Boolean Encoding

Chapter 3 introduced two very distinct methods of deciding a difference logic formula via translation to SAT. This naturally gives rise to the following question: Given a difference logic formula, which encoding technique should one use to decide *that* formula the fastest?

In this chapter, we first present evidence that this question cannot be resolved entirely in favor of either method. We then show that one can select an encoding method based on formula characteristics using a rule generated by machine learning from past examples (formulas). Moreover, parts of a single formula corresponding to different variable classes can be encoded using different encoding methods. The resulting hybrid encoding algorithm is more robust to variation in formula characteristics than either of the two techniques of Chapter 3.

## 6.1 The Need for Algorithm Selection

An experimental study comparing the small-domain (SD) and DIRECT encoding methods over a range of benchmarks indicates that neither method dominates the other in run-time performance. Section 6.1.1 presents the results of this study. These findings motivate the use of automated algorithm selection, described in Section 6.1.2.

### 6.1.1 Comparing the SD and DIRECT Methods

We compare the space and time complexity of the SD and DIRECT decision procedures with respect to both encoding and SAT-solving steps.

Let us first compare the encoding steps of the two decision procedures. The SD encoding algorithm runs in polynomial time and generates a SAT problem that is polynomial-size in the original formula. On the other hand, the DIRECT encoding can generate, in the worst case, a SAT problem that is exponential in the size of the original formula (this is due to a worst-case exponential number of transitivity constraints; see Example 3.4).

The above comparison suggests always favoring SD over DIRECT, since the SD encoding phase is polynomial-time and the SAT instance is polynomial-size in the input. Unfortunately, such a simple judgement cannot be made. First, theoretical worst-case results do not always reflect practice. Second, the run-times of SAT solvers do not always increase monotonically with the size of the SAT instance. In this section, we present experimental evidence supporting the latter behavior, which has also been observed in other contexts (e.g., [77, 126]). We will also formally characterize the structure of the SAT instances generated by the DIRECT encoding method, showing that even when they are bigger than those generated by the SD method, the special structure of the DIRECT encoding method makes the SAT time only polynomially-dependent on the number of transitivity constraints.

Note also that both encoding methods can generate arbitrary SAT instances. For example, when the starting formula is purely Boolean, both methods generate identical output.

**Experimental setup**

All experiments reported in this chapter were based on a set of 49 difference logic formulas,[1] all but 4 of which are unsatisfiable. These formulas are drawn from problems encountered in both hardware and software design verification. The hardware designs include the load-store unit of an industrial microprocessor, an out-of-order microprocessor design [89], a cache coherence protocol [61], and a 5-stage DLX pipeline. The software benchmarks are generated in the verification of safety properties of device driver code [68], and in translation validation [123].

Experiments were run on a Pentium-IV 2 GHz machine with 1 GB of RAM. A timeout of 3600 seconds (one hour) was imposed on each run. For the SAT solving phase, we used the zChaff SAT solver (version 2003.7) with the default options.

**Analysis of results**

Figure 6.1 shows a scatterplot comparing the total run-time (encoding time and SAT time) for both encoding methods. In the plot, the x-coordinate of each point is the time taken by DIRECT, and the y-coordinate is the time taken by SD. We also plot the diagonal line $y = x$ in each plot.

[1] The formulas originally also included applications of uninterpreted functions, but these are first eliminated using the method proposed by Bryant et al. [29]. This method is reviewed in Chapter 7.

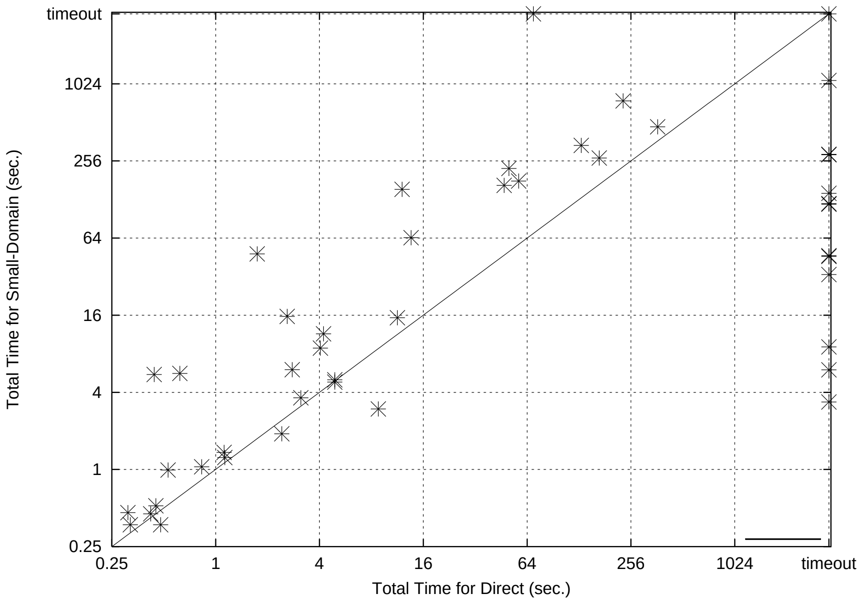



Figure 6.1: **Comparing SD and** DIRECT **encoding methods.** Note the log scales on both axes.

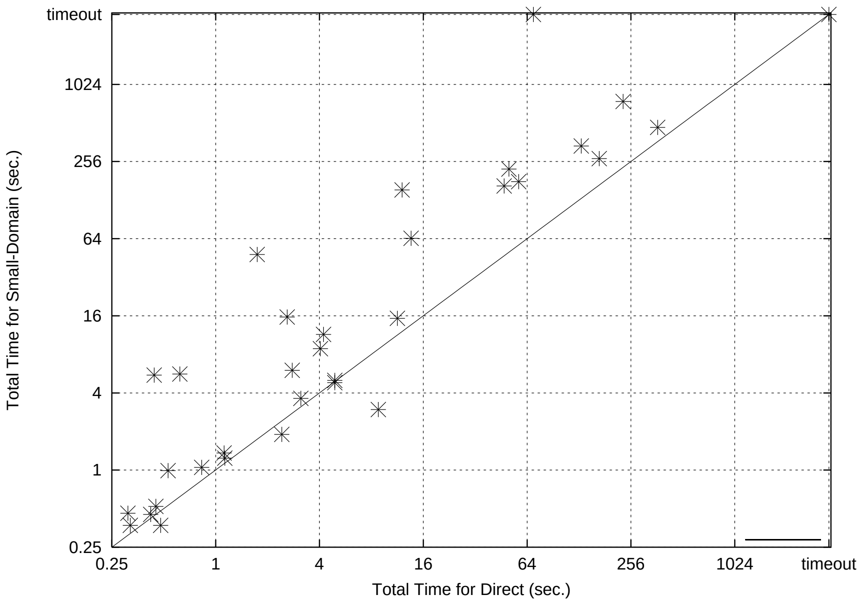



Figure 6.2: **Comparing SD and** DIRECT **methods when** DIRECT **encoding phase completes.** Note the log scales on both axes.

Thus, points above the diagonal correspond to benchmarks on which DIRECT outperforms SD, and vice-versa for the points below the diagonal. Note that some points are spaced close enough to appear superimposed on each other.

The SD method times out on two benchmarks, whereas the DIRECT method does not complete on 12. As expected, the SAT solving phase is the reason that the SD method fails to finish. On the other hand, the DIRECT method fails to reach the SAT solving phase on 11 of the 12 formulas it does not finish on.

Figure 6.2 shows the scatterplot restricted to the 38 benchmarks for which the DIRECT method reaches the SAT solving stage. It is evident that the DIRECT outperforms SD on almost all of these formulas.

A closer look at the data in Figure 6.2 reveals the non-monotonic behavior of the zChaff SAT solver. Table 6.1 shows the effect of the encoding method on zChaff's performance on a representative sample of benchmarks from out-of-order processor verification [89]. Even though the SD method generates smaller SAT instances, zChaff does more backtracking and runs slower on SD-instances as compared to DIRECT-instancs.

| Benchmark | # of CNF Variables | | # of CNF Clauses | | # of Conflict Clauses | | SAT Time (sec.) | |
|---|---|---|---|---|---|---|---|---|
| | SD | DIRECT | SD | DIRECT | SD | DIRECT | SD | DIRECT |
| OOO.rf9 | 14744 | 15898 | 43741 | 47786 | 84748 | 7849 | 152.49 | 8.61 |
| OOO.tag14 | 48825 | 53910 | 145570 | 167308 | 65012 | 8934 | 220.38 | 34.59 |

Table 6.1: **Effect of encoding on zChaff performance.** "Conflict Clauses" denotes the conflict clauses added by zChaff on backtracking.

We have also observed the same behavior for other solvers based on the Davis-Putnam-Logemann-Loveland (DPLL) method, such as BerkMin [63] and Siege [142].

The structure of SAT instances generated by the DIRECT method can be characterized formally. Recall from Section 3.3 that a transitivity constraint generated in the DIRECT encoding algorithm either has the form $e^{b_1}_{j,i} \wedge e^{b_2}_{i,k} \implies e^{b_1+b_2}_{j,k}$ or the form $e^{b_1}_{j,i} \wedge e^{b_2}_{i,k} \implies \mathbf{false}$. Rewriting the constraint as a CNF clause, we either get the expression $(\neg e^{b_1}_{j,i}) \vee (\neg e^{b_2}_{i,k}) \vee e^{b_1+b_2}_{j,k}$ or $(\neg e^{b_1}_{j,i}) \vee (\neg e^{b_2}_{i,k})$. In either case, there is at most one positive literal in the generated CNF clause. In other words, each transitivity constraint is a *Horn clause*. Since transitivity constraints are the source of exponential blow-up in the size of SAT problems generated using the DIRECT encoding, one can characterize the DIRECT encoding as generating *mostly-Horn-SAT* problems, in the worst-case.

A SAT instance comprising only Horn clauses (a Horn-SAT instance) is linear time solvable, with unit propagation being the main step [38]. Thus, in the worst-case, the run-time of a SAT solver is $\mathcal{O}(2^m)$, where $m$ is the number of original difference constraints; i.e., the run-time does not grow exponentially in the number of transitivity constraints.

Although current DPLL-based SAT solvers such as zChaff do not explicitly check for Horn structure, they can solve mostly-Horn-SAT instances very fast. This appears to be mainly due to the efficient implementation of unit propagation.

**Discussion**

We conclude this section with a summary of our findings. We note that:

1. The performance of DPLL-based SAT solvers on instances generated using the DIRECT encoding algorithm is superior to their performance on instances generated using the SD encoding, even when the latter instances are larger.
2. The DIRECT encoding algorithm can, in the worst-case, generate a SAT problem that is exponentially large in the original difference logic formula; moreover, this worst-case behavior manifests itself in practice sometimes. In contrast, the SD encoding algorithm always generates a polynomial-size SAT problem.

The bottleneck for the DIRECT encoding, therefore, is the Boolean encoding step. In experiments, we have observed that if this step completes, it is almost always the case that DIRECT outperforms SD.

### 6.1.2 Automated Algorithm Selection

Since neither one of SD and DIRECT encoding methods dominates the other, we are presented with the following questions:

1. Given an input formula in difference logic, can we automatically select the Boolean encoding method that is best for that formula?
2. Can the SD and DIRECT encoding methods be combined for the same formula?

Let us consider the second question first. As we saw in Chapter 3, variables and constraints can be partitioned into equivalence classes. A separate encoding method can be used for each equivalence class, and this decision is independent of those made for other classes. Thus, we can answer the second question in the affirmative.

The first question can be viewed as an instance of a more general problem called the *algorithm selection problem* [129]. This problem is stated as follows:

> Given a portfolio of algorithms for a problem and a specific problem instance, which algorithm must one select to solve that instance in the least amount of time?

The algorithm selection problem has been studied in various contexts, but never before for the specific problem we consider. Algorithm selection arises naturally in the case of NP-hard problems due to the unpredictability of run-times of heuristic-based algorithms. For example, researchers have recently considered the problem of selecting one of several different algorithms for integer linear programming [93]. There has also been work on choosing between different polynomial-time algorithms for a problem, e.g., for selecting between sorting algorithms [88].

The general framework for algorithm selection is as follows:

1. Select features $f_1, f_2, \ldots, f_k$ of the input that characterize the run-time of each alternative algorithm. These features must be computable in (low-degree) polynomial time. Feature selection is typically done manually.

2. Use machine learning techniques to derive a rule $r$ based on the features from a training set of problem instances (formulas, in our case). Mathematically, the rule is a function from the feature space to the set of candidate algorithms.

3. At run-time, compute the values of features for the input, and evaluate the rule: $r(f_1, f_2, \ldots, f_k)$ is the selected algorithm.

In the next section, we present an approach based on the above framework to automatically selecting between the SD and DIRECT Boolean encoding algorithms.

## 6.2 Learning-Based Approach

Applying the above learning-based approach in our specific context requires making two design decisions. First, a suitable set of input features must be selected. This is addressed in Sections 6.2.1 and 6.2.2. Second, a machine learning algorithm must be chosen; this is discussed in Section 6.2.3. In addition, modifications must be made to the Boolean encoding algorithm so as to permit combining both SD and DIRECT methods whilst using automated algorithm selection. We discuss these modifications in Section 6.2.4.

### 6.2.1 Complexity of Counting Transitivity Constraints

Our first task is to pick a feature of the input formula that best characterizes the run-times of the SD and DIRECT algorithms. We observed in Section 6.1.1 that the DIRECT algorithm outperforms SD when the DIRECT encoding phase completes. Thus, a predictor of the run-time of the DIRECT encoding phase is a natural choice of formula feature. The best predictor of the DIRECT encoding time is the number of transitivity constraints.

Unfortunately, the following result shows that the number of transitivity constraints is not a suitable formula feature.

**Theorem 6.1** *A polynomial-time algorithm for counting the number of transitivity constraints cannot exist.*

*Proof:* The proof is by contradiction. Suppose a polynomial-time algorithm $\mathcal{A}$ does exist.

Every transitivity constraint involving three variables involves the addition of an edge to the constraint graph, if it did not already exist. We show that $\mathcal{A}$ must keep track of (i.e., maintain at least one bit of storage for) every new edge added to the constraint graph. As illustrated in Example 3.4, the number of new edges added at a node elimination step can be exponential in the size of the original formula (the starting constraint graph for Example 3.4 is reproduced in Figure 6.3, for convenience). This implies that, in the worst-case, $\mathcal{A}$ performs exponentially many writes, which is a contradiction.

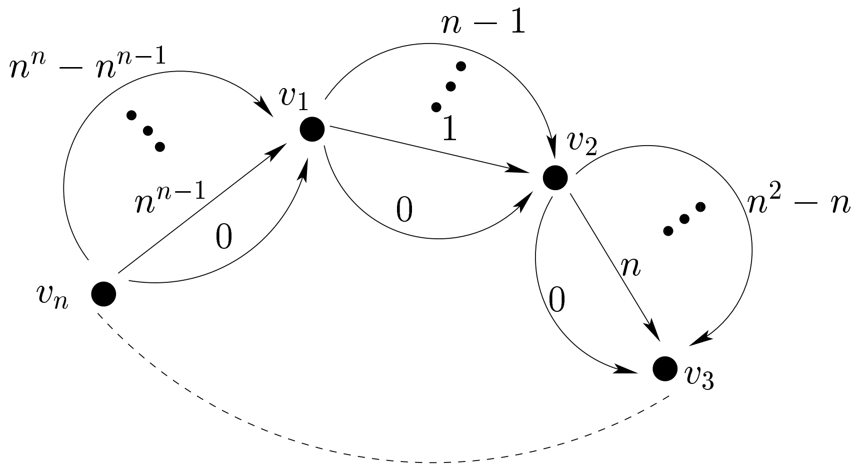



Figure 6.3: **Exponential blow-up of** DIRECT **encoding, revisited**

Suppose $\mathcal{A}$ does not keep track of every newly added edge. Consider the graph in Figure 6.3. We observe that:

1. There are $n^k$ paths of distinct weight between $k$ adjacent vertices.

2. Each new edge that is added in the DIRECT encoding algorithm, as a result of eliminating some vertex $v_i$, accumulates the weights of edges on a distinct path between a subset of adjacent vertices containing $v_i$.

3. Every newly added edge is generated by applying the transitivity rule to a unique pair of previously existing edges, and this procedure continues until only two vertices remain.

Thus, if the edge $e$ missed by $\mathcal{A}$ was added at the $i^{\text{th}}$ node elimination step, $\mathcal{A}$ will fail to record the $n^{n-i-2}$ new edges that are generated transitively from $e$. In other words, the count maintained by $\mathcal{A}$ will fall short of the correct count by at least $n^{n-i-2}$.

Thus, $\mathcal{A}$ must keep track of every newly added edge, implying that a polynomial-time algorithm for counting the number of transitivity constraints cannot exist. $\Box$

### 6.2.2 Feature Selection

The hardness of counting transtivity constraints, expressed in Theorem 6.1, implies that we must look for other formula features to base the algorithm selector on.

Four features were selected for the results reported here. The features and the rationale for selecting them are as follows:

1. $m$, *the number of difference constraints:* Constraint graphs with very few edges (bounded by a small constant) are likely to generate few transitivity constraints.

2. $n$, *the number of variables:* The rationale is similar to that for $m$.

3. $\frac{m}{n}$: This ratio is the average number of edges per vertex. If a vertex has a large number of both incoming and outgoing edges, eliminating it is likely to generate many new edges.

4. $\frac{m}{n^2}$: This ratio is the average number of edges per node-pair, and is a measure of the density of the graph.

Thus, each difference logic formula is represented by a corresponding feature vector $(m, n, \frac{m}{n}, \frac{m}{n^2})$.

Note that all four features, by themselves, are not perfect predictors of the number of transitivity constraints. For example, if the starting constraint graph is a directed acyclic graph (DAG), eliminating vertices in a topologically sorted order will result in no edges being added, even if $m$ is very large. There is also no formal reason why this set of features is adequate. The only justification of our choice is the experimental validation presented in Section 6.3.

A major advantage of our choice of features is that they are computable in low-degree polynomial time. In the absence of *ITE* expressions, all four features are exactly computable in linear time, by performing a scan of the formula. However, if *ITE* expressions are present, it is preferable not to eliminate them since the elimination step itself can lead to an exponential blow-up. Therefore, we instead estimate $m$ by performing a cross-product operation at each relational operator. A detailed description of this operation are deferred to Section 6.2.4.

### 6.2.3 Machine Learning Technique

The choice of a machine learning technique depends on the domain and range of the function to be learnt. In our situation, we wish to learn a binary-valued decision function $r(m, n, \frac{m}{n}, \frac{m}{n^2})$ such that

$$r(m, n, \frac{m}{n}, \frac{m}{n^2}) = \begin{cases} 1 & \text{if DIRECT must be selected} \\ 0 & \text{if SD must be selected} \end{cases} \tag{6.1}$$

The domain of the decision function $r$ is $\mathbb{Z} \times \mathbb{Z} \times \mathbb{R} \times \mathbb{R}$. A particularly suitable technique for learning a binary function of numerical parameters is the *support vector machine* [157]. Given a set of points in $\mathbb{R}^n$ with some points labeled 0 (negative examples) and some labeled 1 (positive examples), a support vector machine (SVM) attempts to find the "best possible" separation of the negative examples and the positive examples. In the simplest case, the examples are linearly separable, and the generated separator is a linear function defining the half-space of $\mathbb{R}^n$ in which the positive examples lie. However, in practice, examples are not usually linearly separable and the data can also be noisy. The real strength of SVMs lies in their ability to learn non-linear separators that optimally separate the examples (for a suitable definition of optimality). The key idea is to project the points into a higher dimensional space in which an optimal linear separator can be found.

Further details on SVMs are outside the scope of this thesis. We refer the interested reader to Christopher Burges' tutorial on the subject [35].

In our context, an SVM learner is used as follows. First, we generate feature vectors for a set of *training examples*, viz., a set of formulas used to learn a decision rule. The SVM learner is applied to the resulting set of feature vectors to obtain a decision rule. Note that this process of learning is off-line. Second, given a new formula to decide, the learned decision rule is used to classify it according to Equation 6.1.

Details on the SVM implementation we used are discussed in Section 6.3.

### 6.2.4 Hybrid Encoding Algorithm

The choice between SD and DIRECT encoding algorithms is *local*, made separately for each variable class. Since a difference logic formula typically corresponds to several variable classes, making local decisions based on the learned decision rule $r$ leads to a *hybrid* encoding algorithm.

Figure 6.4 re-defines difference logic syntax for easy reference in this section.

Given a difference logic formula $F_{diff}$, the hybrid encoding algorithm generates an equi-satisfiable Boolean formula $F_{bool}$ in the following six steps.

1. **Generate variable classes.** Let $V$ denote the set of variables. We start by assigning each

$$\begin{array}{rcl} \textit{bool-expr} & ::= & \mathbf{true} \mid \mathbf{false} \mid \textit{bool-var} \mid \neg\textit{bool-expr} \mid (\textit{bool-expr} \vee \textit{bool-expr}) \\ & & \mid (\textit{bool-expr} \wedge \textit{bool-expr}) \mid (\textit{int-expr} = \textit{int-expr}) \mid (\textit{int-expr} < \textit{int-expr}) \\ \textit{int-expr} & ::= & x_i \mid \textit{int-expr} + b \mid \textit{ITE}(\textit{bool-expr}, \textit{int-expr}, \textit{int-expr}) \end{array}$$

Figure 6.4: **Difference logic syntax, revisited.** $x_i$, $0 \leq i \leq n$, and $b$ denote an integer variable and constant respectively.

variable to its own class. We then compute the *dependency set* for each term in $F_{diff}$, denoting some subset of variables in $V$ to which this term could evaluate. While doing this, some of the classes are merged so that each dependency set is a subset of some class. For term $T \doteq x_i$, its dependency set is $\{x_i\}$. For term $T \doteq T_1 + b$, its dependency set is the same as that of $T_1$. For $T \doteq \textit{ITE}(F, T_1, T_2)$, its dependency set is the union of those of $T_1$ and $T_2$. If the dependency sets of $T_1$ and $T_2$ are subsets of two distinct classes, then we merge those classes. For each equation $T_1 = T_2$ and each inequality $T_1 < T_2$, we perform a similar merging if the dependency sets of $T_1$ and $T_2$ are subsets of distinct classes. Let $V_1, \ldots, V_K$ be the $K$ different variable classes generated by this procedure.

2. **Generate ground terms.** A *ground term* is an expression of the form $x_i + b$, viz., an integer offset from a variable. We transform the formula to generate ground terms by repeatedly applying the following rewrite rules until a fixed point is reached.

$$T + 0 \rightarrow T$$
$$(T + b_1) + b_2 \rightarrow T + (b_1 + b_2)$$
$$\textit{ITE}(F, T_1, T_2) + b \rightarrow \textit{ITE}(F, T_1 + b, T_2 + b)$$

At this point, the terms at the leaves of the formula (viewed as a expression graph) consist only of ground terms.

3. **Compute solution bounds for each variable class.** Recall from Remark 3.1 in Section 3.2 that the solution bound $d_i$ for a variable class $V_i$ with $n_i$ variables is given by

$$d_i = \sum_{j=1}^{n_i-1} |b_{i_j} + 1| \tag{6.2}$$

where $b_{i_1}, b_{i_2}, \ldots, b_{i_{n-1}}$ are the $n_i - 1$ largest constants appearing in constraints corresponding to class $V_i$.

The quantity $n_i$ is easily computable, but computing the constants takes a little more work due to the presence of *ITE* expressions. The constants are computed as follows. For each equation $T_1 = T_2$ and each inequality $T_1 < T_2$ corresponding to class $V_i$, we find the set of ground terms $G(T_1)$ and $G(T_2)$ that $T_1$ and $T_2$ can evaluate to, respectively. This is done by modifying the algorithm for computing the *dependency set* (described above) to include ground terms in addition to variables. For every pair $(x_{k_1}+b_{k_1}, x_{k_2}+b_{k_2})$ in the cross product $G(T_1) \times G(T_2)$, where $k_1 \neq k_2$, we compute the constant term $|b_{k_1} - b_{k_2}| + 1$. The $n_i - 1$ largest such terms are recorded, and used for computing $d_i$.

Given $d_i$, we obtain the length $S_i$ of the bit-vector required to encode each variable in class $V_i$.

4. **Compute an upper bound on the number of difference constraints for each class.** For each of the classes $V_i$, we compute an upper bound $\mathit{diffcnt}_i$ on the number of difference constraints $m_i$. This is done as follows. Initially, $\mathit{diffcnt}_i = 0$, for each class $V_i$. Then, for each equation $T_1 = T_2$ and each inequality $T_1 < T_2$ corresponding to $V_i$, we find the set of ground terms $G(T_1)$ and $G(T_2)$ that $T_1$ and $T_2$ can evaluate to, respectively. For every pair $(t_1, t_2)$ in the cross product $G(T_1) \times G(T_2)$ that has not been encountered yet, and where $t_1$ and $t_2$ are distinct from each other, we increment $\mathit{diffcnt}_i$ by 1.

   Note that $\mathit{diffcnt}_i$ is an upper bound on $m_i$ because we count constraints that disappear after eliminating ITEs, e.g., counting $(x_1, x_2)$ at the node *ITE*$(F,\, x_1,\, x_2) =$ *ITE*$(\neg F,\, x_2,\, x_1)$.

5. **Perform hybrid encoding.** At this point, we have all the information we need to encode the difference logic formula into a Boolean formula.

   The algorithm proceeds by recursing on the formula structure. A Boolean variable retains the same encoding. For a node $f_1 \wedge f_2$ (or $f_1 \vee f_2$), we recursively encode the subexpressions $f_1$, $f_2$ and conjoin (or disjoin) the results. Similarly, $\neg f_1$ is evaluated by encoding $f_1$ and negating the result. The more interesting cases involve equation or inequalities.

   For each equation $T_1 = T_2$ or an inequality $T_1 < T_2$, we find the class $V_k$ which contains the variables that appear in $G(T_1)$ and $G(T_2)$.

   We then evaluate the SVM classifier for $V_k$. The result of the classifier for $V_k$ is

   $$r(\mathit{diffcnt}_k, n_k, \frac{\mathit{diffcnt}_k}{n_k}, \frac{\mathit{diffcnt}_k}{n_k^2})$$

   Note that the classifier has to be evaluated only once for each variable class.

   If the classifier returns 0, then we encode $T_1$, $T_2$ using the SD method. The encodings of $T_1$ and $T_2$ are symbolic bit-vectors of size $S_k$. Bitwise equality or comparison is used to translate a relational operator to a Boolean expression. The arithmetic operations $+$ and $-$ are encoded using binary arithmetic, and *ITE* expressions are encoded as multiplexors.

   Otherwise, if the classifier returns 1, we use the DIRECT method to encode $T_1$ and $T_2$, using the technique proposed by Bryant et al. [29]. Suppose $T_1$ evaluates to a ground term $g_i$ under the condition $c_i^1$, and $T_2$ evaluates to $g_j$ under $c_j^2$. For example, the term *ITE*$(F, x_1, x_2)$ evaluates to $x_2$ under $\neg F$. The encoding of the predicate $T_1 \bowtie T_2$, where $\bowtie \in \{=, <\}$, is given by $\bigvee_{i,j} c_i^1 \wedge c_j^2 \wedge e^{\bowtie}_{g_i,g_j}$, where $e^{\bowtie}_{g_i,g_j}$ is a symbolic Boolean constant to encode the constraint $g_i \bowtie g_j$.

6. **Generate** $F_{bool}$**.** Let $F_{bvar}$ denote the formula obtained by performing the hybrid encoding on $F_{\mathit{diff}}$. We generate the conjunction $F_{trans}$ of all transitivity constraints for predicates in $F_{\mathit{diff}}$ encoded using the DIRECT method. The final Boolean formula $F_{bool}$ is then generated as $(F_{trans} \implies F_{bvar})$.

Hereafter, the hybrid encoding algorithm will be denoted as HYBRID.

## 6.3 Experimental Evaluation

The HYBRID encoding algorithm was implemented in UCLID. We report her on experiments comparing HYBRID with the SD and DIRECT encoding methods. We also report comparisons with CVC-Lite [13, 48] (the latest version at the time of writing), a publicly-available SAT-based decision procedure that is sound and complete for integers. CVC-Lite is the successor to SVC, the Stanford Validity Checker [12].

The experimental setup was identical to that used in Section 6.1.1; we used the same 49 benchmarks, the zChaff SAT solver, and the same platform and timeout (3600 sec.) settings.

#### Implementation of HYBRID

The implementation of HYBRID exactly follows the algorithm described in Section 6.2.4. The only remaining details concern our use of SVM learning.

We used a publicly available package called SVM-Light [83, 151]. About one-third of the formulas (17 out of 49) were used as a training set. For each of these formulas, we ran both SD and DIRECT encoding algorithms. If the DIRECT encoding algorithm ran out of memory on a formula, we marked it as a negative example; if not, we marked it as a positive example. The input to SVM-Light comprised the labeled feature vectors corresponding to these training examples. Note that the $m$ values in the feature vectors were computed exactly, since we ran the DIRECT encoding algorithm which eliminates *ITE* expressions as a first step. The only preprocessing step applied to the feature vectors before running the SVM-Light learner on them was to scale all features to be of the same order by multiplying by a constant (for our case, in and around the range $[0, 1]$), as recommended by Hsu et al. [79]. This is to avoid larger-valued features (such as $m$) dominating smaller ones (such as $\frac{m}{n^2}$), and also to avoid numerical computation errors. We used SVM-Light to learn a non-linear separator by choosing a degree-three polynomial kernel with unit coefficients.

### Results

Figure 6.5 shows a scatterplot comparing the total run-time (encoding + SAT) of the SD method to that for the HYBRID method. The format of this plot is identical to those in Figures 6.1 and 6.2. We observe that HYBRID outperforms SD on almost all benchmarks, including one on which SD times out while HYBRID completes within about 2 minutes. There is one benchmark on which both HYBRID and SD fail to complete within the timeout; this is an example on which neither SD nor DIRECT complete due to the time taken by the SAT solver. There are a few benchmarks on which SD outperforms HYBRID, but both run-times are either very close or on the order of a few seconds.

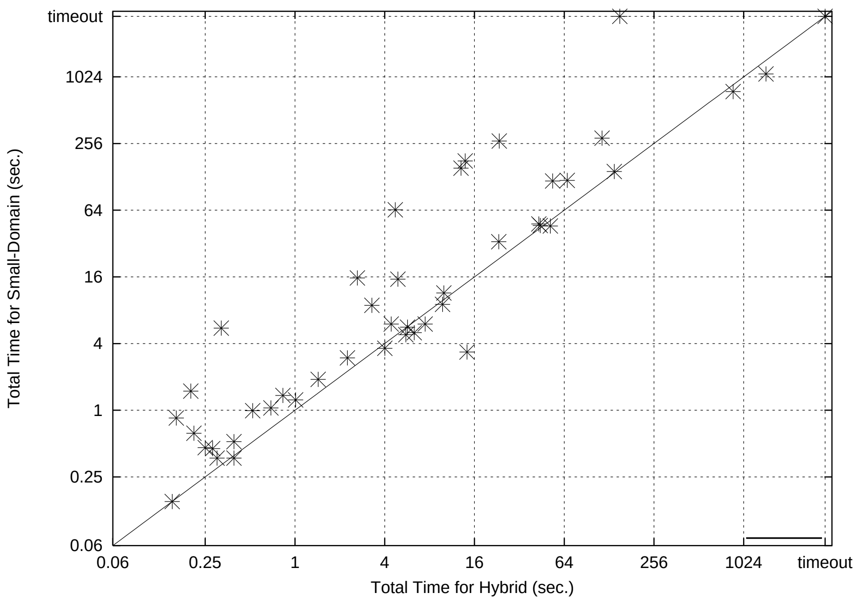



Figure 6.5: **Comparing SD and** HYBRID **encoding methods.** Note the log scales on both axes.

Figure 6.6 shows the comparison of HYBRID with DIRECT. Again we see that HYBRID outperforms DIRECT on the majority of formulas, including 11 on which DIRECT times out while HYBRID finishes. There are also two examples on which DIRECT outperforms HYBRID by about a factor of four and on which both methods take longer than a minute. The reason for DIRECT's superior performance on these benchmarks is to due to misclassification by the SVM learner, which, in turn, is likely because the set of features is inadequate to fully characterize the number of transitivity constraints.

Figure 6.7 compares HYBRID with CVC-Lite. CVC-Lite terminates within the timeout on 19 of the

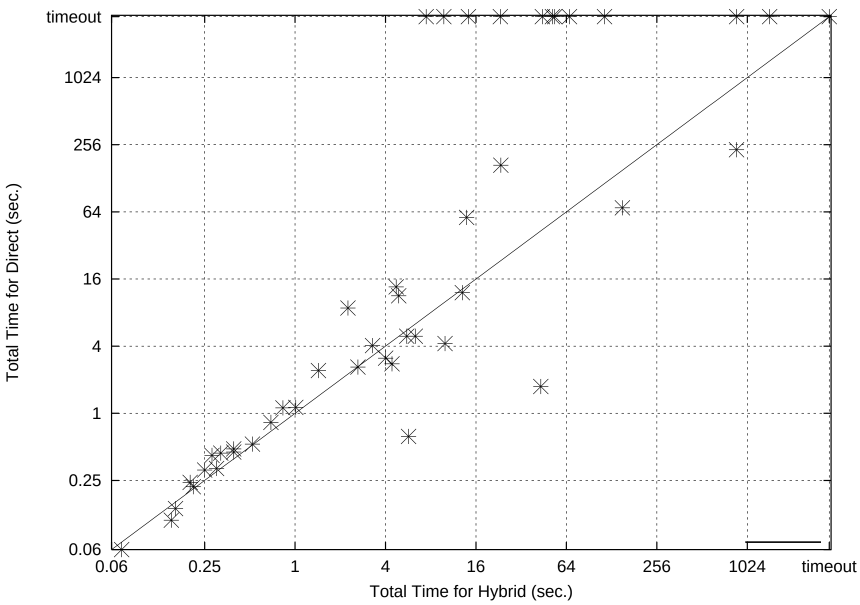



Figure 6.6: **Comparing** DIRECT **and** HYBRID **encoding methods.** Note the log scales on both axes.

49 benchmarks. HYBRID outperforms CVC-Lite on all but 8 benchmarks, on all of which HYBRID terminates within a minute. These 8 benchmarks are all conjunctions of atomic predicates which requires CVC-Lite to only make a single call to its ground decision procedure that solves a system of difference constraints using Fourier-Motzkin elimination. On the remaining 11 benchmarks on which CVC-Lite terminates, we can see that HYBRID sometimes outperforms CVC-Lite by over a factor of 1000.

In summary, the improvement of HYBRID over DIRECT is due to reduction in the number of transitivity constraints, while the improvement over SD is due to reduced SAT time. We have also demonstrated that HYBRID can greatly outperform a state-of-the-art procedure such as CVC-Lite.

## 6.4 Discussion

We presented a novel hybrid Boolean encoding method for difference logic, making two main conceptual contributions. First, we demonstrated the complementary strengths of the SD and DIRECT encodings and showed how they can be combined. Second, we showed how machine learning can be used to automatically select between the two encoding algorithms based on past examples. Experimental results demonstrate the robustness of the resulting HYBRID method to variations in formula characteristics.

The work in this chapter is just a first step towards automated algorithm selection in the context

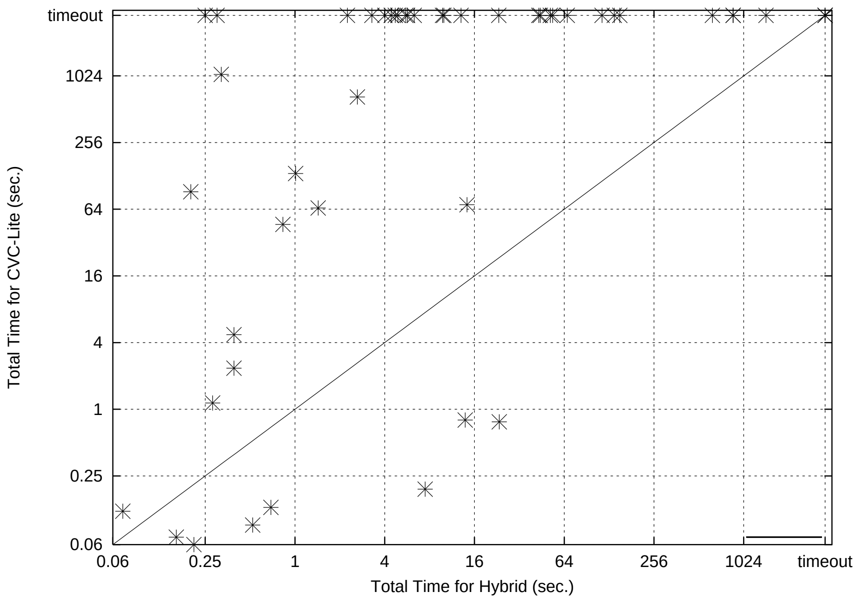



Figure 6.7: **Comparing CVC-Lite and the** HYBRID **encoding method.** Note the log scales on both axes.

of decision procedures, in general, and for SAT-based procedures, in particular. There are many directions for future work.

The problem of misclassification and feature selection need further study. The feature set can be expanded, and techniques for feature subset selection [50] can be employed. Other methods for learning a binary function of numerical inputs, such as *logistic regression* [78], deserve further exploration.

Although the number of transitivity constraints cannot be counted exactly in polynomial time, there is still the possibility of finding an approximation algorithm. A somewhat related problem, that of counting the number of cycles in a directed graph, has been proved to be hard to approximate to a $1 + \epsilon$ factor [82]. This problem appears to be related since the goal of adding transivity constraints is to ensure that the constraint graph corresponding to a satisfying assignment does not contain a positive weight cycle. The implications of the hardness result for counting transivity constraints are, however, unclear.

# Chapter 7

# Extended Logic and Applications

The decision procedures described in this thesis are implemented in a verification system called UCLID. The logic underlying UCLID includes not only linear arithmetic over integers, but also two other logical constructs, viz., *uninterpreted functions* and a restricted form of *lambda expressions*. These additional constructs are very useful in modeling a variety of both hardware and software systems.

The first half of this chapter describes the extensions to the logic and the corresponding extensions to UCLID's decision procedures. In the second half, we describe the verification techniques available in UCLID, for which the decision procedures form the computational engine. We also illustrate how one of these techniques, *bounded model checking*, has proved useful in analyzing software for a class of security bugs known as *format-string vulnerabilities*.

## 7.1 Extended Logic

Figure 7.1 gives the syntax for the extended logic that includes the following three theories:

1. Uninterpreted functions
2. Quantifier-free Presburger arithmetic
3. Restricted lambda expressions (these can be used to express arrays, for example)

Expressions in the extended logic are of four different types. As before, two of the types are *Boolean* and *integer*. Boolean expressions, or formulas, yield **true** or **false**. *Integer* expressions, also referred to as *terms*, yield integer values. *Predicate* expressions denote functions from integers to Boolean values. *Function* expressions, on the other hand, denote functions from integers to integers.

$$
\begin{array}{rcl}
\textit{bool-expr} & ::= & \mathbf{true} \mid \mathbf{false} \mid \neg \textit{bool-expr} \mid (\textit{bool-expr} \wedge \textit{bool-expr}) \mid (\textit{bool-expr} \vee \textit{bool-expr}) \\
& & \mid (\sum_{j=1}^{n} a_j \cdot \textit{int-expr}_j = b) \;\mid\; (\sum_{j=1}^{n} a_j \cdot \textit{int-expr}_j < b) \\
& & \mid \textit{predicate-expr}(\textit{int-expr}, \ldots, \textit{int-expr}) \\
\textit{int-expr} & ::= & \textit{int-var} \mid b \mid \textit{int-expr} + \textit{int-expr} \mid a \cdot \textit{int-expr} \\
& & \mid \textit{ITE}(\textit{bool-expr}, \textit{int-expr}, \textit{int-expr}) \\
& & \mid \textit{function-expr}(\textit{int-expr}, \ldots, \textit{int-expr}) \\
\textit{predicate-expr} & ::= & \textit{predicate-symbol} \;\mid\; \lambda\, \textit{int-var}, \ldots, \textit{int-var} \,.\, \textit{bool-expr} \\
\textit{function-expr} & ::= & \textit{function-symbol} \;\mid\; \lambda\, \textit{int-var}, \ldots, \textit{int-var} \,.\, \textit{int-expr}
\end{array}
$$

Figure 7.1: **Expression syntax for extended UCLID logic.** Expressions can denote computations of Boolean values, integers, or functions of integers yielding Boolean values or integers. $a_j$ and $b$ denote integer constants.

The simplest Boolean expressions are the values **true** and **false**. Boolean expressions can also be formed as a linear equation or inequality over integer expressions, by applying a predicate expression to a list of integer expressions, and by combining Boolean expressions using Boolean connectives. Relational and Boolean operators not shown in the figure can expressed in terms of those employed.

Integer expressions can be integer variables, used only as the formal arguments of lambda expressions, or an integer constant (note the difference here with the syntax used earlier in the thesis). They can also be formed by combining integer expressions with the operators (interpreted functions) for linear arithmetic, by applying a function expression to a set of integer expressions, or by applying the *ITE* ("if-then-else") operator.

Function expressions can be either function symbols, representing uninterpreted functions, or lambda expressions, defining the value of the function as an integer expression containing references to a set of argument variables. Function symbols of arity zero are also called *symbolic constants*. They denote arbitrary integer values, and play the same role in this chapter as integer variables (denoted $x_i$) in previous chapters. Since these symbols are instantiated without any arguments, we will omit the parentheses, writing $f$ instead of $f()$.

Similarly, predicate expressions can be either predicate symbols, representing uninterpreted predicates, or lambda expressions, defining the value of the predicate as a Boolean expression containing references to a set of argument variables. Predicate symbols of arity zero are also called *symbolic Boolean constants*. They denote arbitrary Boolean values, and play the same role as Boolean variables in previous chapters. We will also omit the parentheses following the instantiation of such a predicate.

Notice that we restrict the parameters to a lambda expression to be integers, and not function or predicate expressions. There is no way in our logic to express any form of iteration or recursion.

An integer variable $x$ is said to be *bound* in expression $E$ when it occurs inside a lambda expression for which $x$ is one of the argument variables. We say that an expression is *well-formed* when it contains no unbound variables. The value denoted by a well-formed expression is defined relative to an interpretation $I$ of the function and predicate symbols. Interpretation $I$ assigns to each function symbol of arity $k$ a function from $\mathbb{Z}^k$ to $\mathbb{Z}$, and to each predicate symbol of arity $k$ a function from $\mathbb{Z}^k$ to $\{\mathbf{true}, \mathbf{false}\}$. Given an interpretation $I$ of the function and predicate symbols and a well-formed expression $E$, we can define the *valuation* of $E$ under $I$, denoted $[E]_I$, according to its syntactic structure. The valuation of $E$ is either a Boolean value, an integer, a function from integers to Boolean values, or a function from integers to integers, according to whether $E$ is a Boolean expression, an integer expression, a predicate expression, or a function expression, respectively. We omit the details. A well-formed formula $F$ is *true under interpretation* $I$ if $[F]_I$ is **true**. It is *valid* when it is true under all possible interpretations.

Note that our logic is quantifier-free. It is well-known that adding quantifiers to even the sub-logic of uninterpreted functions and equality results in undecidability [19, 65].

We now show how the newly added logical constructs can be used for modeling a range of hardware and software constructs.

### 7.1.1 Uninterpreted Function Symbols

Uninterpreted functions and predicates satisfy no particular property other than *functional consistency*, viz., that they evaluate to the same value on the same arguments. Functional consistency is formalized in the theory of uninterpreted functions as the *congruence axiom*. This axiom is stated below for an arbitrary uninterpreted function symbol $f$ of arity $k$:

$$\begin{gathered}\forall x_{11}, x_{12}, \ldots, x_{1k}, x_{21}, x_{22}, \ldots, x_{2k} : \\ (x_{11} = x_{21} \wedge x_{12} = x_{22} \wedge \ldots \wedge x_{1k} = x_{2k}) \implies f(x_{11}, x_{12}, \ldots, x_{1k}) = f(x_{21}, x_{22}, \ldots, x_{2k})\end{gathered} \tag{7.1}$$

Uninterpreted functions and predicates are used in hardware verification to abstract word-level values of data and implementation details of functional blocks. Similarly, in software analysis, operators for non-linear functions such as multiplication and division can be abstracted using uninterpreted functions. In addition, uninterpreted functions and predicates are particularly useful in modeling data access functions, such as array and memory operations.

Uninterpreted functions are useful when comparing two systems for behavioral equivalence, such as a specification and its implementation. This is because using the same function symbol in a symmetric way in the two systems ensures that it will return the same values when applied to equal arguments. For example, one successful use of the UCLID system is in the verification of *pipelined microprocessor designs*, where a pipelined implementation is compared with an instruction set architecture (ISA) model [89]. Similarly, in software analysis, uninterpreted functions find use in applications such as *translation validation* [123], where two program fragments are checked for behavioral equivalence.

### 7.1.2 Lambda Expressions

Lambda expressions are extremely useful in modeling data structures. In this section, we give a few representative examples. We use a record notation to represent data structures that are characterized by multiple expressions.

#### Memories

Lambda notation allows us to model the effect of a sequence of *read* and *write* operations on a memory (the *select* and *update* operations on an array). At any point of system operation, a memory is represented by a function expression $M$ denoting a mapping from addresses to values (for an array, the mapping is from indices to values). The initial state of the memory is given by an uninterpreted function symbol $m_0$ indicating an arbitrary memory state. The effect of a write operation with integer expressions $A$ and $D$ denoting the address and data values yields a function expression $M'$:

$$M' \quad = \quad \lambda\, addr\, .\, \mathit{ITE}(addr = A,\ D,\ M(addr))$$

Reading from array $M$ at address $A$ is simply yields the function application $M(A)$.

Multi-dimensional memories or arrays are easily expressed in exactly the same way. Moreover, lambda expressions can express *parallel-update* operations, which express the result of updating multiple memory locations in a single step. This is particularly relevant for hardware, and can also be used in modeling concurrent software. For instance, to express the result of resetting to zero all memory locations that have negative values, we can write

$$M' \quad = \quad \lambda\, a\, .\, \mathit{ITE}(M(a) < 0,\ 0,\ M(a))$$

The ability to model the *select* and *update* array operations raises a natural question about whether the lambda notation introduced in this section is more (or less) expressive than the standard non-extensional theory of arrays [34, 110]. In fact, these two theories are incomparable, for the following reasons:

1. The standard theory of arrays cannot model parallel-update operations. As we have shown, these can be easily expressed with our lambda notation.

2. The standard theory of arrays allows two arrays to be compared for equality. Formally, such a comparison between arrays $M_1$ and $M_2$ can be written as $\forall i.M_1(i) = M_2(i)$. Since our logic is quantifier-free (with implicit universal quantification on all symbols at the top level), such a comparison can only be made when it appears in the formula under an even number of negations, by applying both arrays to a fresh symbol that is universally quantified at the top level.

Other forms of memory can be modeled as well using lambda expressions. For example, we can model a Content Addressable Memory (CAM) that stores associations between keys and data. We represent a CAM $C$ at any point in the system operation by two expressions: a predicate expression $C.\mathit{present}$ such that $C.\mathit{present}(k)$ is true for any key $k$ that is stored in the CAM, and a function expression $C.\mathit{data}$, such that $C.\mathit{data}(k)$ yields the data associated with key $k$, assuming the key is present. As an initial state in invariant checking we can represent a CAM $C$ having an arbitrary state by letting $C.\mathit{present} = p_0$ and $C.\mathit{contents} = c_0$, where $p_0$ (respectively, $c_0$) is an uninterpreted predicate (resp., function).

Insertion into a CAM is expressed by the operation $\mathit{Insert}(C, K, D)$. This operation yields a new CAM $C'$ where:

$$\begin{aligned} C'.\mathit{present} &= \lambda\, \mathit{key}\ .\ \mathit{key}\!=\!K \lor C.\mathit{present}(\mathit{key}) \\ C'.\mathit{data} &= \lambda\, \mathit{key}\ .\ \mathit{ITE}(\mathit{key}\!=\!K,\ D,\ C.\mathit{data}(\mathit{key})) \end{aligned}$$

On the other hand, the effect of deleting the entry associated with key $K$ is expressed by the operation $\mathit{Delete}(C, K)$. This operation yields a new CAM $C'$ where

$$\begin{aligned} C'.\mathit{present} &= \lambda\, \mathit{key}\ .\ \neg(\mathit{key}\!=\!K) \land C.\mathit{present}(\mathit{key}) \\ C'.\mathit{data} &= C.\mathit{data} \end{aligned}$$

**Ordered Data Structures**

We show how an ordered data structure, such as a queue, can be modeled using lambda notation and linear arithmetic.

A queue of arbitrary length can be modeled as a record $Q$ having components $Q.\mathit{contents}$, $Q.\mathit{head}$, and $Q.\mathit{tail}$. Conceptually, the contents of the queue are represented as some subsequence of an infinite sequence, where $Q.\mathit{contents}$ is a function expression mapping an integer index $i$ to the value of sequence element $i$. $Q.\mathit{head}$ is an integer expression indicating the index of the head of the queue, i.e., the position of the oldest element in the queue. $Q.\mathit{tail}$ is an integer expression indicating the index at which to insert the next element. In general, we require $Q.head \leq Q.tail$ as an invariant property. $Q$ is modeled as having an arbitrary state by letting $Q.contents = c_0$, $Q.head = h_0$, and $Q.tail = t_0$, where $c_0$ is an uninterpreted function and $h_0$ and $t_0$ are symbolic constants satisfying the constraint $h_0 \leq t_0$. This constraint is enforced by including it in the antecedent of the formula whose validity we wish to check.

The operation testing if the queue is empty can be expressed quite simply as:

$$isEmpty(Q) = (Q.head = Q.tail)$$

Using this operation we can define the following three operations on the queue:

1. $Pop(Q)$: The pop operation on an non-empty queue returns a new queue $Q'$ with the first element removed; this is modeled by incrementing the head.

$$Q'.head = \mathit{ITE}(isEmpty(Q),\ Q.head,\ Q.head + 1)$$

2. $First(Q)$: This operation returns the element at the head of the queue, provided the queue is non-empty. It is defined as $Q.contents(Q.head)$.

3. $Push(Q, X)$: Pushing data item $X$ into $Q$ returns a new queue $Q'$ where

$$\begin{aligned} Q'.tail &= Q.tail + 1 \\ Q'.contents &= \lambda\, i\ .\ \mathit{ITE}(i = Q.tail,\ X,\ Q.contents(i)) \end{aligned}$$

Assuming we start in a state where $h_0 \leq t_0$, $Q.head$ will never be greater than $Q.tail$ because of the conditions under which we increment the head.

Bounded length queues can be similarly expressed, with an additional constraint in the case of the push operation disallowing a push when the queue is full. In particular, to bound a queue to a maximum length of $k$ (where $k$ is an integer, not a symbolic constant), we add the condition for pushing that $Q.tail$ is incremented only when $Q.tail < Q.head + k$.

### Partially Interpreted Functions

We noted earlier that non-linear arithmetic operations can be abstracted using uninterpreted functions. Lambda expressions allow us to assign a partial interpretation to such operations.

For instance, for integer multiplication, we can express the property that the constant 1 is the multiplicative identity and 0 is the annihilator, by defining multiplication as follows:

$$mul = \lambda\, i, j\ .\ \mathit{ITE}(i = 0 \vee j = 0,\ 0,\ \mathit{ITE}(i = 1,\ j,\ \mathit{ITE}(j = 1,\ i,\ mult(i, j))))$$

Here the uninterpreted function $mult$ is the default in case none of the special cases are matched.

The use of such partially interpreted functions can reduce the imprecision that abstraction of non-linear operators introduces.

## 7.2 Decision Procedure Extensions

Given a formula $F_{ucl}$ in the extended logic of UCLID, we decide its validity by performing a satisfiability-preserving translation to a Boolean formula $F_{bool}$ in a single step, and then invoking a SAT solver on $F_{bool}$. The translation operates in three steps:

1. All lambda expressions are eliminated, resulting in a formula $F_{norm}$.
2. Function and predicate applications of non-zero arity are eliminated to get a formula $F_{arith}$.
3. Formula $F_{arith}$ is in quantifier-free Presburger arithmetic. We translate $F_{arith}$ to an equi-satisfiable Boolean formula $F_{bool}$ using the methods described in Chapters 3–6.

A brief description of the first two steps of translation follows. Details on eliminating function applications are outside the scope of this thesis and can be found in earlier work [2, 29].

### 7.2.1 Elimination of Lambda Expressions

Recall that the extended logic syntax does not permit recursion or iteration. Therefore, each lambda application in $F_{ucl}$ can be expanded by *beta-substitution*, i.e., by replacing each argument variable with the corresponding argument term. Denote the resulting formula by $F_{norm}$.

This step can result in an exponential blow-up in formula size. Suppose that all expressions in our logic are represented as directed acyclic graphs (DAGs) so as to share common sub-expressions. Then, the following example shows how we can get an exponential-sized DAG representation of $F_{norm}$ starting from a linear-sized DAG representation of $F_{ucl}$.

**Example 7.1** Let $F_{ucl}$ be defined recursively by the following set of expressions:

$$\begin{array}{rcl}
F_{ucl} & \doteq & P(L_1(b)) \\
L_1 & \doteq & \lambda\, x\, .\, f_1(L_2(x), L_2(g_1(x))) \\
L_2 & \doteq & \lambda\, x\, .\, f_2(L_3(x), L_3(g_2(x))) \\
\vdots & & \vdots \\
L_{n-1} & \doteq & \lambda\, x\, .\, f_{n-1}(L_n(x), L_n(g_{n-1}(x))) \\
L_n & \doteq & g_n
\end{array}$$

Notice that the representation of $F_{ucl}$ is linear in $n$. Suppose we perform beta-substitution on $L_1$. The sub-expression $L_1(b)$ gets transformed to $f_1(L_2(b), L_2(g_1(b)))$. Next, if we expand $L_2$, we get four applications of $L_3$, viz., $L_3(b), L_3(g_1(b)), L_3(g_2(b)), andL_3(g_2(g_1(b)))$. Notice that there were originally only two applications of $L_3$.

Continuing the elimination process, after $k-1$ elimination steps, we will get $2^{k-1}$ distinct applications of $L_k$. This can be formalized by observing that after $k-1$ steps each argument to $L_k$ is comprised of applications of functions from a distinct subset of $\mathcal{P}(\{g_1, g_2, \dots, g_{k-1}\})$. Thus, after all lambda elimination steps, $F_{norm}$ will contain $2^{n-1}$ distinct applications of $g_n$, and hence is exponential in the size of $F_{ucl}$. $\Box$

In practice, however, we have never encountered this exponential blow-up. This is because the recursive structure in most lambda expressions, including those for memory operations, tends to be linear. For example, here is the lambda expression corresponding to the result of the memory write operation:

$$\lambda\, addr\ .\ \mathit{ITE}(addr = A,\ D,\ M(addr))$$

Notice that the "recursive" use of $M$ occurs only in one of the branches of the *ITE* expression.

### 7.2.2 Elimination of Function and Predicate Applications

The second step in the transformation to a Boolean formula is to eliminate applications of function and predicate symbols of non-zero arity. These applications are replaced by symbolic constants (integer or Boolean, as the case may be), but only after encoding enough information to maintain functional consistency.

There are two different techniques of eliminating function (and predicate) applications. The first is a classic method due to Ackermann [2] that involves creating sufficient instances of the congruence axiom (as stated in Equation 7.1). The second is a recent technique introduced by Bryant et al. [29] that exploits the polarity of equations and is based on the use of *ITE* expressions. We briefly review each of these methods.

#### Ackermann's method

We illustrate Ackermann's method using an example.

Suppose that function symbol $f$ has three occurrences: $f(a_1)$, $f(a_2)$, and $f(a_3)$. First, we generate three fresh symbolic constants $xf_1$, $xf_2$, and $xf_3$ to replace all instances of these applications in $F_{norm}$.

Then, the following set of functional consistency constraints for $f$ is generated:

$$
\begin{aligned}
a_1 = a_2 &\implies xf_1 = xf_2 \\
a_1 = a_3 &\implies xf_1 = xf_3 \\
a_2 = a_3 &\implies xf_2 = xf_3
\end{aligned}
$$

In a similar fashion, functional consistency constraints are generated for each function and predicate symbol in $F_{norm}$. Denote the conjunction of all these constraints by $F_{cong}$. Then, $F_{arith}$ is the formula $F_{cong} \implies F_{norm}$.

**Bryant et al.'s method**

The function elimination method proposed by Bryant et al. exploits a property of function applications called *positive equality*. The general idea is to determine the polarity of each equation in the formula, i.e., whether it appears under an even (positive) or odd (negative) number of negations. Applications of uninterpreted functions can then be classified as either p-function applications, i.e., used only under positive equalities, or g-function applications, i.e., general function applications that appear under other equalities or under inequalities. The p-function applications can be encoded in propositional logic with fewer Boolean variables than the g-function applications, thus greatly simplifying the resulting SAT problem. We omit the details.

In order to exploit positive equality, Bryant et al. eliminate function applications using a nested series of *ITE* expressions. As an example, if function symbol $f$ has three occurrences: $f(a_1)$, $f(a_2)$, and $f(a_3)$, then we would generate three new symbolic constants $xf_1$, $xf_2$, and $xf_3$. We would then replace all instances of $f(a_1)$ by $xf_1$, all instances of $f(a_2)$ by *ITE*$(a_2 = a_1,\ xf_1,\ xf_2)$, and all instances of $f(a_3)$ by *ITE*$(a_3 = a_1,\ xf_1,$ *ITE*$(a_3 = a_2,\ xf_2,\ xf_3))$. It is easy to see that this preserves functional consistency.

Predicate applications can be removed by a similar process. In eliminating applications of some predicate $p$, we introduce symbolic Boolean constants $xp_1, xp_2, \ldots$.

Function and predicate applications in the resulting formula $F_{arith}$ are all of zero arity.

### 7.2.3 Summary

We conclude this section with observations on the worst-case blow-up in formula size in going from the starting formula $F_{clu}$ to the quantifier-free Presburger formula $F_{arith}$. The lambda elimination step can result in a worst-case exponential blow-up. In going from the lambda-free formula $F_{norm}$ to $F_{arith}$, the worst-case blow-up is only quadratic. Thus, if the result of lambda expansion is linear in the size of $F_{clu}$, as is typically the case, $F_{arith}$ is at most quadratic in the size of $F_{clu}$.

## 7.3 Verification Techniques in UCLID

UCLID is a tool for specifying and verifying systems modeled in the extended logic described in this chapter. The UCLID system has been publicly available on the Web [156] since May 2001. It has been applied to a range of systems, including out-of-order, pipelined, microprocessor designs [89, 90, 95], a complex load-store unit of an industrial microprocessor, a cache coherence protocol [61], and analyzing software for security vulnerabilities [58]. The last application is the subject of Section 7.4.

#### Specifying Infinite-State Systems in UCLID

The UCLID specification language can be used to specify an infinite-state system. The state variables can either have one of three primitive types — Boolean, enumerated, and integer — or are functions of integer arguments that evaluate to one of these primitive types. The initial (reset) state of each state variable is described by an expression in the extended logic. The transition relation is specified by assigning an expression for computing the value of a variable in state $i$, given the values of variables in states $i-1$ and $i$. Specifically, the next state of a state variable is specified as an expression in the extended logic in which references to the values of state variables in the current and next state can appear in place of symbolic constants. Details on the specification language and UCLID usage are given in Appendix A; we only mention here that the language was inspired by and is similar to that of the CMU version of the SMV model checker [42, 98].

It is also worth mentioning one notable feature about the internal encoding of enumerated types in UCLID. An enumerated type $E$ of $k$ values is encoded as an integer sequence $\{z_E, z_E+1, \ldots, z_E+k-1\}$, where a different symbolic constant $z_E$ is used for each type $E$. The type checker in the UCLID front-end enforces the restriction that variables of an enumerated type can only be compared for equality against other variables of the same enumerated type. Thus, each enumerated type generates a unique singleton variable class $\{z_E\}$. If the small-domain encoding is used, $z_E$ is encoded with a constant bit encoding. On the other hand, if the DIRECT encoding is used, each equation corresponding to an enumerated type gets reduced to either **true** or **false** after *ITE* expressions are eliminated.

#### Verification Techniques

Figure 7.2 shows how the UCLID verification system is structured. The UCLID verification engine comprises two main components:

1. A symbolic simulator that can be configured by the user for different kinds of verification tasks.

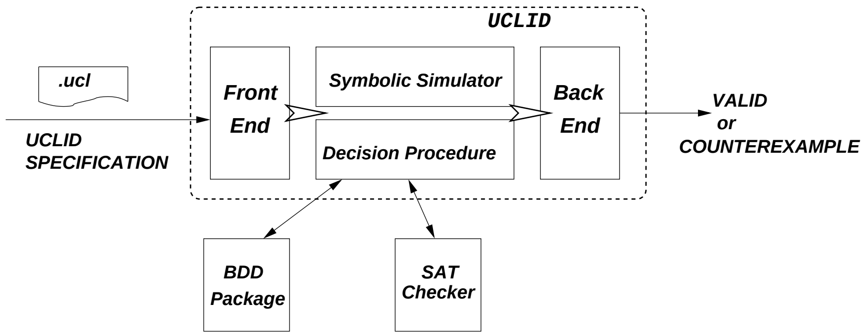



Figure 7.2: **Structure of the UCLID system**

2. A decision procedure for the extended logic described in this chapter.

In addition, there is a front-end that includes a type checker, and a back-end that translates the result of the decision procedure into an output either stating that the system satisfies the property being verified, or giving a counterexample comprising a sequence of states showing how the property is violated.

The following verification methods are supported:[1]

1. *Bounded model checking:* The system is symbolically simulated for a fixed number of steps, specified by the user, starting from a reset state. At each step, the decision procedure is invoked to check the validity of a safety property. If the property fails, UCLID generates a counterexample trace from the reset state.

2. *Inductive invariant checking:* The system is initialized in a most general state satisfying the invariant to be proved. It is symbolically simulated for one step, and the invariant is checked on the resulting state by the decision procedure.

3. *Proving commutative diagrams:* In this method, we attempt to show that a specification machine simulates an implementation machine. This includes the method of *correspondence checking* for superscalar processors, such as in the style of Burch and Dill [34]. UCLID allows the user to set the values of certain designated state variables at different steps of the symbolic simulation. For example, in verifying pipelined processors, this allows the user to specify the steps at which the pipeline must be flushed.

UCLID's decision procedure can check the satisfiability of the Boolean formula $\neg F_{bool}$ using either a BDD package or a SAT solver. We have found SAT solvers to outperform BDDs in all practical

[1]We only describe the methods supported by the base version of UCLID. Shuvendu Lahiri has built a predicate abstraction-based verifier [91] on top of UCLID, but describing that tool is outside the scope of this thesis. We only mention that the Boolean encoding methods described in Chapters 3–7 can be used with Lahiri's work as well.

applications explored thus far; however, we have also encountered artificially generated examples on which BDDs outperform SAT.

A very useful feature of UCLID is its ability to generate counterexample traces, like a model checker. A counterexample to a formula $F_{ucl}$ in UCLID's logic is a partial interpretation $I$ to the function and predicate symbols in the formula, which is generated from a satisfying assignment to $\neg F_{bool}$. If the system has been symbolically simulated for $k$ steps, then the interpretation $I$ generated above can be applied to the expressions at each step, thereby resulting in a complete counterexample trace for $k$ steps.

Unbounded model checking of infinite-state systems that can be modeled in UCLID is undecidable [31].

## 7.4 Case Study: Finding Format-String Exploits

Format-string vulnerabilities [76, 113] are a dangerous class of security bugs that allow an attacker to execute arbitrary code on the victim machine. `printf` is a variable-argument C function that treats its first argument as a *format-string*.[2] A format-string contains *conversion specifications*, which are instructions that specify the types that this call on `printf` expects for its arguments, and instructions on how to format the output. For instance, the conversion specification `"%s"` instructs `printf` to look for a pointer to a `char` value as its next argument, and print the value at that location as a string. When `arg` does not contain conversion specifiers, the statements `printf("%s", arg)` and `printf(arg)` have the same effect. However, if `printf(arg)` is used in an application, and a user can control the value passed to `arg`, then the application may be susceptible to a format-string vulnerability. A possible fix for such vulnerabilities is to do a source-to-source transformation that replaces all occurrences of `printf(arg)` with `printf("%s", arg)`, but this may not always be possible, for instance when the source code of the application is not available, or when the application generates format-strings dynamically.

Shankar *et al.* [140] have built a tool, Percent-S, to analyze source code and identify "tainted" format-strings that can be controlled by an attacker. Potentially vulnerable `printf` locations can also be identified in binary executables [76]. However, the aforementioned techniques do not produce *format-string exploits*, i.e., strings that exploit the vulnerabilities they identify.

We present a novel way to analyze and understand `printf`-family format-string vulnerabilities. The format-string can be viewed as a sequence of commands that instructs `printf` to look for different types of arguments on the application's runtime stack. We have used UCLID to analyze potentially vulnerable call sites to `printf` and determine if an exploit is possible. If an exploit is

[2] While we restrict our discussion to `printf`, the concepts discussed apply to other `printf`-family functions as well, e.g., `syslog`, `sprintf`.

possible, UCLID produces a format-string that demonstrates the exploit. Our technique does not require the source code of the application and can analyze potentially vulnerable `printf` locations from binary executables. We have also used UCLID in conjunction with Percent-S to generate format-strings that exploit the vulnerabilities identified (see Section 7.4.3). Our discussion and implementation make the following platform-specific assumptions, although the technique applies to other platforms as well:

1. We work with the x86 architecture. In particular, the runtime stack of an application grows from higher addresses to lower addresses, and the machine is assumed to be little-endian.
2. The arguments to a function are placed on the stack from right to left. A call to foo($\texttt{arg}_1$, $\texttt{arg}_2$) first places $\texttt{arg}_2$ on the stack, followed by $\texttt{arg}_1$. This is a popular C calling convention implemented by several compilers.
3. We analyze `printf` from the `glibc-2.3` library.

### 7.4.1 Background

This section reviews the working of `printf` and describes how an attacker can read from or write to an arbitrary location.

#### Understanding `printf`

Consider the code fragment shown in Figure 7.3. Procedure `foo` accepts user input, which is copied

```
(1) int foo (char *usrinp) {
(2)   char fmt[LEN];
(3)   int a, b;
(4)   strncpy(fmt, usrinp, LEN - 1);
(5)   fmt[LEN - 1] = '\0';
(6)   printf(fmt);
(7) }
```

Figure 7.3: **A procedure with a vulnerable call to `printf`**

into the local variable `fmt`, a local array of `LEN` characters. `printf` is then called with `fmt` as its argument. Because the first argument to `printf` can be controlled by the user, this program can potentially be exploited. When `printf` is called on line (6), the arguments passed to `printf` are placed on the stack, the return address and frame pointer are saved, and space is allocated for the local variables of `printf`, as shown in Figure 7.4(A). In this case, `printf` is called with a pointer to fmt, which is a local character buffer in foo. This pointer is shown as the darkly shaded region in Figure 7.4(A).

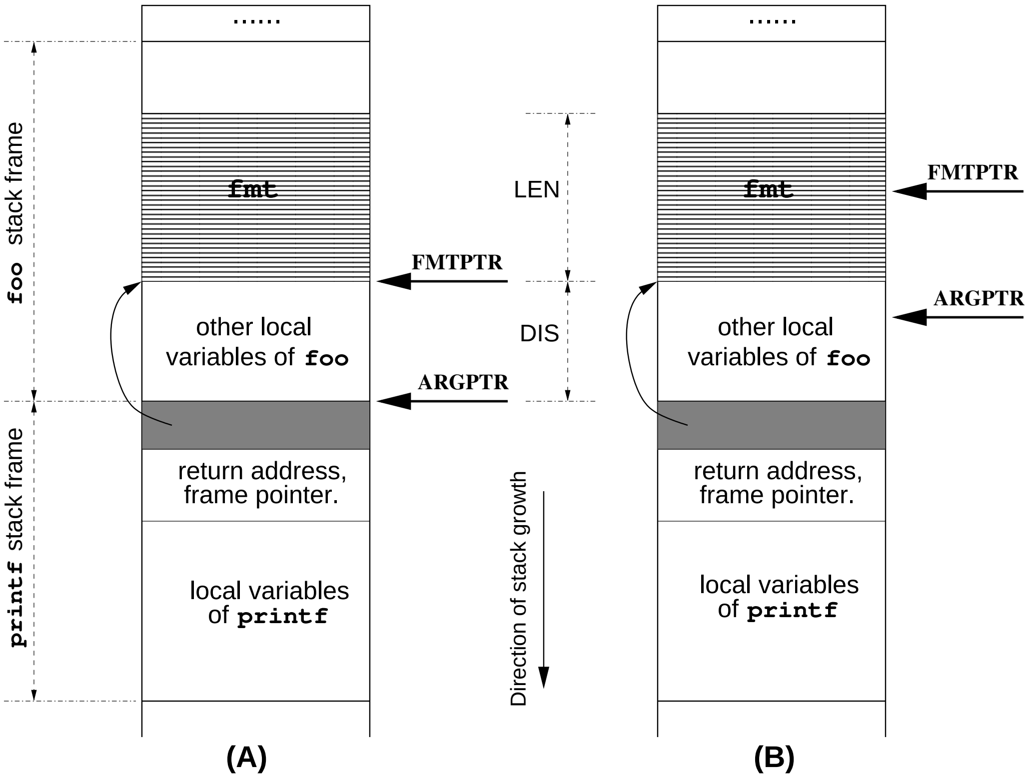



Figure 7.4: **Runtime execution stack for the program in Figure 7.3**

As mentioned earlier, printf assigns special meaning to the first argument passed to it, and treats it as a format-string. Any other arguments passed to printf appear at higher addresses than the format-string on the runtime stack. In our case, only fmt was passed as an argument, and hence there are no other arguments on the runtime stack.

The printf implementation internally maintains two pointers to the stack; we will refer to these pointers as FMTPTR and ARGPTR. The purpose of FMTPTR is to track the current formatting character being scanned from the format-string, while ARGPTR keeps track of the location on the stack from where to read the next argument. Before printf begins to read any arguments, FMTPTR is positioned at the beginning of the format-string and ARGPTR is positioned just after the pointer to the format-string fmt, as shown in Figure 7.4(A).

When printf begins to execute, it moves FMTPTR along format-string fmt. Advancing a pointer makes it move towards higher addresses in memory, hence FMTPTR moves in the direction opposite to which the stack grows. printf can be in one of two "modes". In *printing* mode, it reads bytes off the format-string and prints them. In *argument-capture* mode, it reads arguments from the stack from the location pointed to by ARGPTR. The type of the argument, and thus the number of bytes by which ARGPTR has to be advanced as it reads the argument, is determined by the contents of the location pointed to by FMTPTR. As FMTPTR and ARGPTR move toward higher addresses, they reach intermediate configurations, as shown in Figure 7.4(B). Note that ARGPTR advances only if the contents of `fmt` causes `printf` to enter argument-capture mode at least once.

To take a concrete example, suppose that `fmt` is `"Hi%d"` when `printf` is called in Figure 7.3. `printf` starts off in printing mode, and advances FMTPTR, printing `Hi` to `stdout` as a result. When FMTPTR encounters the byte `"%"`, it enters argument-capture mode. When FMTPTR is advanced, it points to the byte `"d"` – which instructs `printf` to read four bytes from the location pointed to by ARGPTR and print the resulting value to the terminal as an integer. This also results in ARGPTR being advanced by four bytes, the size of an integer. Note that no integer arguments were explicitly passed to `printf` in Figure 7.3, hence instead of reading a legitimate integer value off the stack, in this case ARGPTR reads the values of local variables in the stack frame of `foo`. As a result, it is possible to read the contents of the stack, which may possibly contain values of interest to an attacker, such as return addresses.

### Format-String Exploits

The key observation in understanding format-string exploits is that each byte in the format-string is an instruction to `printf` to move FMTPTR and ARGPTR by an appropriate amount, and to interpret the arguments passed to it. In the format-string exploits discussed herein, the goal of the attacker is to control the contents of the format-string in such a way that ARGPTR advances along the stack until it enters the format-string itself. By doing so, the attacker can control the arguments read by `printf` as well as how those arguments are interpreted.

Each call to `printf` is characterized by two parameters, namely the values DIS and LEN shown in Figure 7.4. The format-string vulnerabilities we consider occur when the format-string is a buffer on the runtime stack. LEN denotes the length of this buffer. DIS denotes the number of bytes that separate the pointer to the format-string from the format-string itself. Figure 7.4 shows a simple scenario where the stack frame containing the format-string and the stack frame of `printf` are adjacent. In general, they can be separated by stack frames of several intermediate functions, resulting in larger values of DIS. From the attacker's viewpoint, ARGPTR has to move by at least DIS bytes by the time FMTPTR moves LEN $-$ 1 bytes.

There are two main kinds of format-string exploits:

1. **Read exploits:** One of the ways an attacker can print the contents of memory at address $a_4a_3a_2a_1$, where $a_4$ is the most-significant byte, is to construct a format-string that satisfies the following property: The format-string must move FMTPTR and ARGPTR such that when

`printf` is in printing mode and FMTPTR points to the beginning of a `"%s"`, ARGPTR points to the beginning of a sequence of four bytes whose value as a pointer is $a_4a_3a_2a_1$. Then, when `printf` reads the `"%s"`, it interprets the argument at ARGPTR as a pointer and prints the contents of the memory location specified by the pointer as a string, which would let the attacker achieve his goal.

2. **Write exploits:** Another kind of format-string exploit allows an attacker to write a value of his choice at a location in memory chosen by him. To do so, he makes use of the `"%n"` feature provided by `printf`. When `printf` is in printing mode and encounters a `"%n"` in the format-string, it reads an argument off the stack, which it interprets to be a pointer to an integer. It then writes to this location the number of bytes that have been output by this call on `printf`. As the write location is of the attacker's choice, it could be the return address of `printf`, for example, making `printf` return to an attack script instead of the function it was called from.

Note that the values of the address bytes $a_1, a_2, a_3, a_4$ must be non-zero, because a zero value is interpreted as `'\0'`, and terminates the format-string. For ease of explanation, we impose the additional restriction that $a_i \neq$ `"%"`, for $i \in \{1, 2, 3, 4\}$. If $a_i =$ `"%"`, the address can contain (parts of) a conversion specifier. However, UCLID can also discover exploits where the address $a_4a_3a_2a_1$ contains `"%"`.

### 7.4.2 Formal Specification

The main insight in deriving a formal model of the problem is to view `printf` as the system being subverted and the format-string as the input to `printf` that is under the attacker's control. We will show in this section how `printf` can be modeled as an infinite-state system and how the two kinds of exploits described in Section 7.4.1 can be formalized as violations of safety properties.

#### Formal Model of `printf`

We can model `printf` as an infinite-state system expressible in UCLID with the following three components:

1. **State Variables:** The set of state variables $\mathcal{V}$ is simply the set of local variables in the implementation of `printf` that captures the current state. We identified 24 local variables (or "flags") with integer and Boolean values[3] by examining the source code and manuals of

[3] In the actual implementation of `printf`, the flags are C integer and pointer data types, i.e., finite-precision bit-vectors. In our model, flags that just take two values, 0 and 1, are defined as Boolean variables, while the rest are treated

`printf`. While our implementation considers all these flags, for ease of explanation we restrict ourselves to describing just four flags: FMTPTR, ARGPTR, DONE, and IS_LONGLONG. FMTPTR and ARGPTR are pointers whose functionality was discussed earlier. We shall treat these as integer values. DONE is an integer that counts the number of bytes printed, and IS_LONGLONG is a Boolean variable that determines whether the argument on the stack is a `long long` value or not (a `long long int` is 8 bytes in length). In addition to the local variables in `printf`, $\mathcal{V}$ also includes a variable MODE that models the program counter.

In addition to the state variables mentioned above, we needed to model the runtime stack. This was modeled using an uninterpreted function $stack$ just as illustrated for modeling memories in Section 7.1.2.

2. **Initial State:** The initial state of `printf` is determined by the initial values of the flags in $\mathcal{V}$. We assume that all addressing is relative to the initial location of ARGPTR. Thus, the assignment of initial values to the four flags discussed here are as follows: ARGPTR = $0$, FMTPTR = DIS, DONE = $0$, and IS_LONGLONG = FALSE.

3. **Transition Relation:** As described earlier, each byte in the format-string is interpreted as an instruction to `printf`. Thus, the next state of each state variable is a function of the current and next state of other variables as well as current byte at the stack location pointed to by FMTPTR. For each variable, the next state function involves several cases, far too many to be listed here. We will therefore just illustrate how one of the $256$ possible values of the current entry in the format-string buffer affects the next state values of the four variables highlighted here.

   Consider the effect of reading the character `'%'`. If `printf` is in printing mode (determined by the value of MODE), FMTPTR is incremented, and `printf` enters argument-capture mode. If `printf` is in argument-capture mode, then FMTPTR and DONE are incremented, and `printf` enters printing mode (corresponds to printing a `"%"` to `stdout`). Formally, [(MODE = printing) $\rightarrow$ (FMTPTR$'$ = FMTPTR + 1) $\wedge$ (MODE$'$ = argument-capture)] $\wedge$ [(MODE = argument-capture) $\rightarrow$ (FMTPTR$'$ = FMTPTR + 1) $\wedge$ (DONE$'$ = DONE + 1) $\wedge$ (MODE$'$ = printing)], where, following customary notation, primed variables denote next-state values of the corresponding variables.

The model of `printf` described above was manually extracted from the `glibc-2.3` source code. All arithmetic operations performed by `printf` are expressible as linear arithmetic operators.

as (unbounded) integers. While this approach achieves efficiency by raising the level of abstraction, it does not model integer overflow, and may lead to imprecision.

### Safety Property Formulation

Each kind of format-string exploit is formalized using a predicate we shall denote by **`Bad`**. This predicate is a formula on the elements of $\mathcal{V}$ in UCLID's logic; viz., it involves quantifier-free Presburger arithmetic, uninterpreted functions, and the theory of memories.

Figure 7.5 shows the values of the predicate **`Bad`** for the read-exploit and write-exploit described in Section 7.4.1.

| (A) **`Bad`** for Read Exploit | (B) **`Bad`** for Write Exploit |
|---|---|
| $[\text{FMTPTR} < \text{DIS} + (\text{LEN} - 1) - 1]$<br>$\wedge\ [\text{ARGPTR} > \text{DIS}]$<br>$\wedge\ [\text{ARGPTR} < \text{DIS} + (\text{LEN} - 1) - 4]$<br>$\wedge\ [*\text{FMTPTR} = \text{`\%'}]$<br>$\wedge\ [*(\text{FMTPTR} + 1) = \text{`s'}]$<br>$\wedge\ [*\text{ARGPTR} = a_1]$<br>$\wedge\ [*(\text{ARGPTR} + 1) = a_2]$<br>$\wedge\ [*(\text{ARGPTR} + 2) = a_3]$<br>$\wedge\ [*(\text{ARGPTR} + 3) = a_4]$<br>$\wedge\ [\text{MODE} = \text{printing}]$ | $[\text{FMTPTR} < \text{DIS} + (\text{LEN} - 1) - 1]$<br>$\wedge\ [\text{ARGPTR} > \text{DIS}]$<br>$\wedge\ [\text{ARGPTR} < \text{DIS} + (\text{LEN} - 1) - 4]$<br>$\wedge\ [*\text{FMTPTR} = \text{`\%'}]$<br>$\wedge\ [*(\text{FMTPTR} + 1) = \text{`n'}]$<br>$\wedge\ [*\text{ARGPTR} = a_1]$<br>$\wedge\ [*(\text{ARGPTR} + 1) = a_2]$<br>$\wedge\ [*(\text{ARGPTR} + 2) = a_3]$<br>$\wedge\ [*(\text{ARGPTR} + 3) = a_4]$<br>$\wedge\ [\text{DONE} = \text{WRITEVAL}]$<br>$\wedge\ [\text{MODE} = \text{printing}]$ |

Figure 7.5: **The predicate `Bad` used for read and write exploits.** We use the notation *PTR as a short-form for $stack(\text{PTR})$.

Note the following two points about the entries in Figure 7.5:

1. The little-endianness of the machine is reflected in the formulation of **`Bad`**: bytes are arranged from most-significant to least-significant as addresses decrease; for example, $a_1$ appears at a lower address than $a_4$.
2. Symbolic values of different stack locations, such as those at FMTPTR and ARGPTR, appear in **`Bad`**, and show the need to track stack contents precisely.

### Verification Method

We chose to use the bounded model checking capabilities of UCLID, checking at each step whether the predicate **`Bad`** is satisfied. If so, the counterexample generated by UCLID is directly translated to a format-string that demonstrates the exploit. At each call-site to `printf`, we only need to examine format-strings of length less than or equal to LEN$-1$ (we exclude the terminating `'\0'`). Hence, a bound of LEN$-1$ suffices to make bounded model checking *complete* at that call-site; i.e., a `printf` location deemed safe using our tool with the bound LEN$-1$ will indeed be safe with respect to class of exploits being checked.

### 7.4.3 Results

Given the UCLID model for `printf` constructed as described in Section 7.4.2 and the predicate **`Bad`** for a family of exploits, the only remaining details are the values of DIS and LEN. Note that these values are the only details that are specific to the software being analyzed. The values of DIS and LEN for each `printf` call are obtained by disassembling the binary executable of the application that calls `printf`, and examining the call graph and the sizes of stack frames of relevant functions.

In this section, we describe the results obtained by analyzing the UCLID model for a range of values of DIS and LEN, both for toy models and for real software packages.

#### Analysis for a range of values of DIS and LEN

Figure 7.6 shows some examples of read-exploits produced by the tool for various values of DIS and LEN. For instance, line (3) shows that the format-string "$a_1a_2a_3a_4$`%d%s`" can be used to read the contents of memory at $a_4a_3a_2a_1$ when DIS and LEN are 4 and 16, respectively. The exploit proceeds as follows: initially FMTPTR points to the format-string, and ARGPTR is 4 smaller than FMTPTR. `printf` starts execution in printing mode; it advances FMTPTR and prints the bytes $a_1$, $a_2$, $a_3$, and $a_4$ to `stdout`. When `printf` reads the `'%'`, it advances FMTPTR by one and enters argument-capture mode. When it reads `'d'`, it advances FMTPTR by one, reads an integer (4 bytes) from the location pointed to by ARGPTR, prints this integer to `stdout`, and returns to printing mode. As a result ARGPTR points to the beginning of the format-string, and FMTPTR is positioned at the beginning of the sequence `"%s"`. When `printf` processes the `"%s"`, the contents of memory at location $a_4a_3a_2a_1$ are printed to `stdout`.

We make a few more observations on the entries in Figure 7.6:

1. In line (2), the tool is able to infer that an exploit is not possible. Intuitively, this is because the format-string is too small to contain a sequence of commands that carry out the exploit.
2. Lines (3) and (4) present two format-strings for the same parameters. We achieved this by first observing case (3), and running the tool again, appending a suitable term to **`Bad`** to exclude case (3). This technique can be iterated to infer as many variants of this exploit as desired.

Figure 7.6 also gives examples of write-exploits, where the integer 234 is to be written to memory

| Sl.no. | DIS | LEN | Read exploit | | Write exploit | |
|---|---|---|---|---|---|---|
| | | | Exploit string discovered | Time (sec.) | Exploit string discovered | Time (sec.) |
| (1) | 0 | 7 | "$a_1a_2a_3a_4$%s" | 0.2 | No exploit possible. | 0.3 |
| (2) | 4 | 7 | No exploit possible. | 0.3 | No exploit possible. | 0.3 |
| (3) | 4 | 16 | "$a_1a_2a_3a_4$%d%s" | 0.4 | "%234Lg%n$a_1a_2a_3a_4$" | 4.8 |
| (4) | 4 | 16 | "%Lx%ld%s$a_1a_2a_3a_4$" | 1.0 | "$a_1a_2a_3a_4$%%%229X%n" | 13.1 |
| (5) | 8 | 16 | "$a_1a_2a_3a_4$%Lx%s" | 0.9 | "$a_1a_2a_3a_4$%230g%n" | 22.2 |
| (6) | 16 | 16 | "%Lg%Lg%s$a_1a_2a_3a_4$" | 1.1 | "$a_1a_2a_3a_4$%137g%93g%n" | 106.5 |
| (7) | 20 | 20 | "$a_1a_2a_3a_4$%Lg%g%s" | 5.3 | "$a_1a_2a_3a_4$%210Lg%20g%n" | 148.7 |
| (8) | 24 | 20 | "$a_1a_2a_3a_4$%Lg%Lg%s" | 2.1 | "$a_1a_2a_3a_4$%61Lg%169Lg%n" | 204.2 |
| (9) | 32 | 24 | "$a_1a_2a_3a_4$%g%Lg%Lg%s" | 13.5 | "$a_1a_2a_3a_4$%78Lg%80g%72Lg%n" | 343.5 |

Figure 7.6: **Some format-string exploits generated by UCLID.** For the write exploit, we chose to write the integer 234 to the memory location with a specific address $a_4a_3a_2a_1$.

address $a_4a_3a_2a_1$. Consider line (5) for instance; for the values 8 and 16 for DIS and LEN, respectively, the tool inferred the format-string "$a_1a_2a_3a_4$%230g%n". When `printf` starts execution, it is in printing mode, and ARGPTR is 8 bytes below FMTPTR on the stack. As FMTPTR moves along the format-string, $a_1$, $a_2$, $a_3$, and $a_4$ (4 bytes) are printed to `stdout`, thus incrementing DONE by 4. The next byte "%" increments FMTPTR by 1 byte and forces `printf` into argument-capture mode. The next 3 bytes, '2', '3' and '0' are treated as a width parameter, and `printf` stores the value 230 in an internal flag WIDTH (part of $\mathcal{V}$ for `printf`). When `printf` processes the next byte, 'g', it advances ARGPTR by 8 bytes, reads a `double` value from the stack, prints this value (appropriately formatted) to `stdout`, increments DONE by the value of WIDTH, and returns to printing mode. At this point, ARGPTR points to the beginning of the format-string, whose first four bytes contain $a_1a_2a_3a_4$, DONE is 234, and FMTPTR points to the beginning of the sequence "%n". When `printf` processes "%n", the value of DONE is written to $a_4a_3a_2a_1$, completing the exploit.

The execution times shown in Figure 7.6 were obtained on a machine with an Intel Pentium-4 processor running at 2GHz, with 1GB of RAM, running Redhat Linux-7.2. For these experiments, UCLID used the Siege SAT solver [142]. All runs completed within a few minutes. As a general trend, the time taken increases as LEN increases, although not monotonically. The reason is that for larger values of LEN, it is necessary to run the bounded model checker UCLID for more steps, leading to a larger formula for it to check; the largest formulas were Boolean combinations of several thousand linear constraints over about a hundred integer variables. Note also that the time taken for finding read exploits is much lower than that for finding write exploits. This is because finding a write exploit involves solving a more constrained problem than for the read exploit: In addition to finding a sequence of conversion specifications that moves ARGPTR into the format-string, one needs to find associated width values that add up to the desired value (234 in Figure 7.6). Furthermore, the length of this sequence can be at most LEN$-1$ (of course, this holds for read exploits as well).

### Optimizations

In our model of `printf`, *each byte* in the format-string requires one step of execution. As an optimization we can augment the model so that more than one character is processed at a time. For example, we could augment the model so that the group of three characters `"%Lg"` moves FMTPTR by 3 bytes, ARGPTR by 12 bytes, and reads a `long double` value. Similarly, aggregated groups of characters can include conservative width specifiers;[4] e.g., `"%60Lg"` increments DONE by 60 in addition to changing the other flags as described above. Augmenting the model in this way does not affect soundness because we retain all previously modeled behavior. Thus, all the format-strings that UCLID could previously generate can still be generated. It is an optimization because longer strings can potentially be found with fewer iterations of bounded model checking.

### Comparison with existing tools

To demonstrate the effectiveness of our tool, we compared it with Percent-S [140], a tool that analyzes source code using type-qualifiers [57] to identify "tainted" (i.e., user-controlled) inputs that could potentially be used as format-strings. We report on two experiments here: the first showing how we can reduce the false alarm rate, and the second showing how we can confirm a true vulnerability by generating an exploit.

Consider the program in Figure 7.3. When compiled on our machine, the value of DIS is 28 bytes. Irrespective of the value of LEN, the size of the buffer `fmt`, Percent-S reports that the `printf` statement on line (6) is exploitable. Clearly, small values of LEN preclude the possibility of attack. As a result, Percent-S produces false alarms, because it does not account for the values of the parameters DIS and LEN.

On the other hand, using our model of `printf`, we were able to infer that a read-exploit (similar to the one reported earlier) is not possible unless LEN is at least 15 bytes, and a write-exploit (to write the integer 234) is not possible unless LEN is at least 20 bytes. In each of these cases, our analysis produces a format-string that demonstrates the exploit, while Percent-S does not.

We also used the tool to analyze known format-string vulnerabilities in software packages; Figure 7.1 has the details. `php-3.0.16` is a language-processor for the widely-used web-scripting language `php`, `qpopper-2.53` is a POP3 mail server, and `wu-ftpd-2.6.0` is a popular file-transfer daemon. We explain in detail the exploit against `wu-ftpd-2.6.0`; the others are similar.

[4]The number of bytes printed is the maximum of the width specifier and that needed to precisely represent the output; so the width specifier must be conservatively large.

| No. | Software | DIS | LEN | Exploit | Exploit string discovered |
|---|---|---|---|---|---|
| (1) | `php-3.0.16` [45] | 24 | 1024 | Write `0xbfff8cc3` to `0xbfff88c3` ($a_4a_3a_2a_1$) | "$a_1a_2a_3a_4$`%36000Lg%31Lg%n`" + "`?`$b_1b_2b_3b_4$`%13000Lg%111g%n`" |
| (2) | `qpopper-2.53` [132] | 2120 | 1024 | Read contents at `0xbfff88c3` ($a_4a_3a_2a_1$) | ("`%Lg`")$^{240}$ + ("`?`")$^{52}$ + "`%Ld%Ld%d%d%s`$a_1a_2a_3a_4$" |
| (3) | `wu-ftpd-2.6.0` [154] | 9364 | 4096 | Write `0xbfffbcab` to `0xbfff88c3` ($a_4a_3a_2a_1$) | "$a_1a_2a_3a_4$`%99g`$b_1b_2b_3b_4$" + ("`%60Lg`")$^{778}$ + "`%912g%600Lg%n%852X%n`" |

Table 7.1: **Exploits generated against vulnerabilities in real-world software packages.** "`?`" represents a non-zero non-`%` ASCII character. The address $b_4b_3b_2b_1$ is $a_4a_3a_2a_1 + 2$.

Percent-S correctly identified the location of the vulnerability in `wu-ftpd-2.6.0`, but did not produce a format-string demonstrating the exploit. The value of DIS and LEN for this example were $9364$ and $4096$, respectively, which we obtained by disassembling the binary executable. For these values of DIS and LEN, we checked whether the attacker could perform the following exploit: The attacker uses the buffer that stores the format-string to additionally store malicious code, and then overwrites the return address in the stack frame of `printf` using a write exploit so as to point to the beginning of the malicious code sequence instead. We assumed that the return address to be overwritten is at the stack location `0xbfff88c3`, and that the malicious code is located at the address `0xbfffbcab`, 13288 bytes above (and hence located within the buffer that stores the format-string). These address values are easily read off the stack using another exploit, as explained in Section 7.4.1. Because the value to be written is fairly large, we used a variant of the predicate **`Bad`** that allows for writing to a single address using multiple, slightly misaligned writes of smaller values. (Details on doing such misaligned writes can be found in [113, 154].)

Because the values of DIS and LEN are quite large, we had to use the optimizations described in Section 7.4.3. We were able to infer, in about 10 minutes, a format-string that is the concatenation of the following three strings: A prefix "$a_1a_2a_3a_4$`%99g`$b_1b_2b_3b_4$", a middle part ("`%60Lg`")$^{778}$ consisting of 778 repetitions of group of characters ”%60Lg”, and a suffix "`%912g%600Lg%n%852X%n`", where $a_4a_3a_2a_1$ is `0xbfff88c3` and $b_4b_3b_2b_1 = a_4a_3a_2a_1 + 2$. It can be verified that this string writes the desired value to the desired location. One write is performed by each "`%n`": the first writes `0xbcab` to $a_4a_3a_2a_1$ and the second writes `0xbfff` to $b_4b_3b_2b_1$.

Existing format-string exploit generators attempt to construct format strings from fixed conversion specifiers. For instance, Thuemmel [154] constructs format-strings with the "`%.8x`" conversion specifier as the only building block. As a result, these techniques lack soundness: there may be exploit strings outside the space of strings explored by these tools. By doing an exhaustive search of the state space, our technique guarantees soundness within our model of `printf`. In addition, existing tools are incapable of finding variants of an exploit. As demonstrated in lines (3) and (4) of Figure 7.6, our technique can be used to discover variants of an exploit for the same values of DIS and LEN.

## 7.5 Summary

This chapter extended quantifier-free Presburger arithmetic with uninterpreted functions and restricted lambda expressions. The resulting logic, which forms the underlying logic for the UCLID verification system, is expressive and the eager approach to translating to SAT can be easily extended to it. We have demonstrated the practical applicability of UCLID by applying it to the analysis of format-string vulnerabilities and the generation of exploit strings for real software packages.

# Part II

# Model Checking Timed Systems

# Chapter 8

# Quantified Difference Logic

*Quantified difference logic* (QDL) is the logic obtained by extending difference logic with universal and existential quantifiers. QDL has applications in model checking timed systems, expressed, for example, as timed automata [3, 5], since the fundamental model checking operations are expressible in QDL.

Formally, a QDL formula $\omega$ is generated by the following grammar:

$$\omega ::= \phi \mid \neg\omega \mid \omega_1 \wedge \omega_2 \mid \omega_1 \vee \omega_2 \mid \exists x.\omega \mid \exists e.\omega \mid \forall x.\omega \mid \forall e.\omega \qquad (8.1)$$

We will denote real-valued variables by $x, x_1, x_2, \ldots$, Boolean variables by $e, e_1, e_2, \ldots$, and real-valued constants by $c, c_1, c_2, \ldots$. As before, $x_0$ denotes a special variable representing the constant 0. The symbol $\phi$ denotes an arbitrary difference logic formula over Boolean and real-valued variables. Unlike in Chapters 3–7, Boolean and reals are the only primitive data types. We will also not employ the *ITE* construct.

We will denote QDL formulas by $\omega, \omega_1, \omega_2, \ldots$. The satisfiability problem for QDL is known to be PSPACE-complete [86].

In this chapter, we show how to perform operations in QDL using Boolean methods. The general strategy is to transform the problem of eliminating quantifiers on real-valued variables to one of eliminating quantifiers on Boolean variables. Specifically, given a QDL formula $\omega$ with quantifiers over real-valued variables, we transform it to an *equivalent* QDL formula $\omega_{bool}$ that has quantifiers only over Boolean variables. These quantifiers can then be eliminated using standard Boolean techniques (e.g., [33, 99]) that are based on Binary Decision Diagrams (BDDs) or Boolean satisfiability (SAT) solvers. Compared to previous quantifier elimination approaches, ours has the twin advantages of leveraging previous work on finite-state model checking as well as avoiding the need to enumerate terms in the Disjunctive Normal Form (DNF) of the quantifier-free portion of the formula. Moreover, for a special class of QDL formulas occurring in model checking of timed automata, the transformation can be greatly optimized.

We begin in Section 8.1 by describing how quantifiers over real-variables are replaced by those over Boolean variables. The Boolean encoding method employed is very similar to the DIRECT encoding algorithm introduced in Chapter 3. Next, in Section 8.2, we describe a modified version of the DIRECT encoding algorithm for DL formulas over real-valued variables. Section 8.3 describes how DL formulas are represented and manipulated as Boolean formulas. Finally, Section 8.4 describes several optimizations that have proved useful in practice. We will defer a discussion of related work to Section 9.1, as all prior work has been done in the context of model checking timed systems.

## 8.1 Quantifier Elimination Using Boolean Methods

Let $\phi$ denote a DL formula over $n$ real variables $x_1, x_2, \dots, x_n$, and $k$ Boolean variables $e_1, e_2, \dots, e_k$. Also, let $\bowtie, \bowtie_1, \bowtie_2 \in \{>, \geq\}$.

Consider the QDL formula $\omega_a \doteq \exists x_a.\phi$, where $a \in [1..n]$.

We transform $\omega_a$ to an equivalent QDL formula $\omega_{bool}$ with quantifiers over only Boolean variables in the following three steps:

1. *Encode difference constraints:*

   Consider each difference constraint in $\phi$ of the form $x_i \bowtie x_j + c$ where either $i = a$ or $j = a$. For each such predicate, we generate a corresponding Boolean variable $e_{i,j}^{\bowtie,c}$. Difference constraints that are negations of each other are represented by Boolean literals (true or complemented variables) that are negations of each other; however, for ease of presentation, we will extend the naming convention for Boolean variables to Boolean literals, writing $e_{j,i}^{>,-c}$ for the negation of $e_{i,j}^{\geq,c}$.

   Let the added Boolean variables be $e_{i_1,a}^{\bowtie_{i_1},c_{i_1}}, e_{i_2,a}^{\bowtie_{i_2},c_{i_2}}, \dots, e_{i_m,a}^{\bowtie_{i_m},c_{i_m}}$ for the upper bounds on $x_a$, and $e_{a,j_1}^{\bowtie_{j_1},c_{j_1}}, e_{a,j_2}^{\bowtie_{j_2},c_{j_2}}, \dots, e_{a,j_{m'}}^{\bowtie_{j_{m'}},c_{j_{m'}}}$ for the lower bounds on it.

   We replace each predicate $x_a \bowtie x_j + c$ (or $x_i \bowtie x_a + c$) in $\phi$ by the corresponding Boolean variable $e_{a,j}^{\bowtie,c}$ (or $e_{i,a}^{\bowtie,c}$). Let the resulting DL formula be $\phi_{bool}^a$.

2. *Add transitivity constraints:*

   Notice that there can be assignments to the $e_{i,a}^{\bowtie,c}$ and $e_{a,j}^{\bowtie,c}$ variables that have no corresponding assignment to the real-valued variables. To disallow such assignments, we place constraints on these added Boolean variables. Each constraint is generated from two Boolean literals that encode predicates containing $x_a$. Following the terminology introduced in Chapter 3, we will refer to these constraints as *transitivity constraints for* $x_a$.

   A transitivity constraint for $x_a$ has one of the following types:

(a) $e_{i,a}^{\bowtie_1,c_1} \wedge e_{a,j}^{\bowtie_2,c_2} \implies (x_i \bowtie x_j + c_1 + c_2)$,
where if $\bowtie_1 = \bowtie_2$, then $\bowtie = \bowtie_1$, otherwise, we must duplicate this constraint for both $\bowtie = \bowtie_1$ and for $\bowtie = \bowtie_2$.

(b) $e_{i,j}^{\bowtie_1,c_1} \implies e_{i,j}^{\bowtie_2,c_2}$, where $c_1 > c_2$ and either $i = a$ or $j = a$.

(c) $e_{i,j}^{>,c} \implies e_{i,j}^{\geq,c}$, where either $i = a$ or $j = a$.

Note that a constraint of type (a) involves a difference constraint $(x_i \bowtie x_j + c_1 + c_2)$. This predicate might not be present in the original formula $\phi$.

After generating all transitivity constraints for $x_a$, we conjoin them to get the DL formula $\phi^a_{cons}$.

3. Finally, generate the QDL formula $\omega_{bool}$ given below:

$$\exists e_{i_1,a}^{\bowtie_{i_1},c_{i_1}}, e_{i_2,a}^{\bowtie_{i_2},c_{i_2}}, \ldots, e_{i_m,a}^{\bowtie_{i_m},c_{i_m}}.\exists e_{a,j_1}^{\bowtie_{j_1},c_{j_1}}, e_{a,j_2}^{\bowtie_{j_2},c_{j_2}}, \ldots, e_{a,j_{m'}}^{\bowtie_{j_{m'}},c_{j_{m'}}}.[\phi^a_{cons} \wedge \phi^a_{bool}]$$

We formalize the correctness of the preceding transformation in the following theorem.

**Theorem 8.1** *$\omega_a$ and $\omega_{bool}$ are equivalent.*

*Proof:* To show that $\omega_a$ and $\omega_{bool}$ are equivalent, we show that $\omega_a \implies \omega_{bool}$ and $\omega_{bool} \implies \omega_a$.

Denote the formula $\omega_a \implies \omega_{bool}$ by $\omega^1$ and the formula $\omega_{bool} \implies \omega_a$ by $\omega^2$. Note first that the free variables in both implications are the real-valued variables $x_1, x_2, \ldots, x_{a-1}, x_{a+1}, \ldots, x_n$ and the Boolean variables $e_1, e_2, \ldots, e_k$. For all $i$ and $j$, the values assigned to $x_i$ and $e_j$ by an assignment $\sigma$ are denoted by $\sigma[x_i]$ and $\sigma[e_j]$ respectively.

1. We first show that $\omega^1$ is valid.

   Let $\sigma$ denote an arbitrary assignment to all free variables and to the bound real variable $x_a$ in $\omega_a$ such that $\sigma[\omega_a] = \mathbf{true}$. We extend $\sigma$ with an assignment to the Boolean variables $e_{i_1,a}^{\bowtie_{i_1},c_{i_1}}, e_{i_2,a}^{\bowtie_{i_2},c_{i_2}}, \ldots, e_{i_m,a}^{\bowtie_{i_m},c_{i_m}}$ and $e_{a,j_1}^{\bowtie_{j_1},c_{j_1}}, e_{a,j_2}^{\bowtie_{j_2},c_{j_2}}, \ldots, e_{a,j_{m'}}^{\bowtie_{j_{m'}},c_{j_{m'}}}$, such that $\sigma[\omega_{bool}] = \mathbf{true}$ and hence $\sigma[\omega^1] = \mathbf{true}$.

   Define an evaluation of the newly added Boolean variables according to the following rules:

   $$\sigma[e_{a,j}^{c,\bowtie}] = \sigma[x_a \bowtie x_j + c] \qquad \forall j \neq a, \text{ for all constants } c \text{ and relations } \bowtie \tag{8.2}$$
   $$\sigma[e_{i,a}^{c,\bowtie}] = \sigma[x_i \bowtie x_a + c] \qquad \forall i \neq a, \text{ for all constants } c \text{ and relations } \bowtie \tag{8.3}$$

   Since $\sigma[\omega_a] = \mathbf{true}$, $\sigma[\phi] = \mathbf{true}$. Further, using Equations 8.2 and 8.3, we can conclude that $\sigma[\phi^a_{bool}] = \sigma[\phi]$ because $\phi^a_{bool}$ is obtained from $\phi$ by replacing predicates $(x_a \bowtie x_j + c)$ and $(x_i \bowtie x_a + c')$ (for all $i, j$ and for all constants $c, c'$) with Boolean variables $e_{a,j}^{c,\bowtie}$ and $e_{i,a}^{c',\bowtie}$. Therefore, $\sigma[\phi^a_{bool}] = \mathbf{true}$.

To show that $\sigma[\omega_{bool}] = \mathbf{true}$, we need to additionally show that $\sigma[\phi^a_{cons}] = \mathbf{true}$. We consider an arbitrary transitivity constraint of each type:

(a) $e^{\bowtie_1,c_1}_{i,a} \wedge e^{\bowtie_2,c_2}_{a,j} \implies (x_i \bowtie x_j + c_1 + c_2)$.
Suppose $\sigma[e^{\bowtie_1,c_1}_{i,a}] = \sigma[e^{\bowtie_2,c_2}_{a,j}] = \mathbf{true}$. Then, by Equations 8.2 and 8.3, we conclude that $\sigma[x_i] \bowtie_1 \sigma[x_a] + c_1$ and $\sigma[x_a] \bowtie_2 \sigma[x_j] + c_2$. If $\bowtie_1 = \bowtie_2 = \bowtie$, we can infer $\sigma[x_i] \bowtie \sigma[x_j] + c_1 + c_2$, and thus $\sigma[x_i \bowtie x_j + c_1 + c_2] = \mathbf{true}$. If $\bowtie_1 \neq \bowtie_2$, then we can infer $\sigma[x_i \bowtie_1 x_j + c_1 + c_2] = \sigma[x_i \bowtie_2 x_j + c_1 + c_2] = \mathbf{true}$.

(b) $e^{\bowtie_1,c_1}_{i,j} \implies e^{\bowtie_2,c_2}_{i,j}$, where $c_1 > c_2$ and either $i = a$ or $j = a$.
Suppose $\sigma[e^{\bowtie_1,c_1}_{i,j}] = \mathbf{true}$. Then, by Equations 8.2 and 8.3, $\sigma[x_i \bowtie_1 x_j + c_1] = \mathbf{true}$. Since $c_1 > c_2$, $\sigma[x_i \bowtie_2 x_j + c_2] = \mathbf{true}$, and hence $\sigma[e^{\bowtie_2,c_2}_{i,j}] = \mathbf{true}$.

(c) $e^{>,c}_{i,j} \implies e^{\geq,c}_{i,j}$, where either $i = a$ or $j = a$.
Exactly as for type (b) constraints, $\sigma[e^{>,c}_{i,j}] = \sigma[x_i > x_j + c] = \mathbf{true}$. Therefore, $\sigma[x_i \geq x_j + c] = \mathbf{true}$ and hence $\sigma[e^{\geq,c}_{i,j}] = \mathbf{true}$.

Thus, $\sigma$ satisfies all transitivity constraints, and hence $\sigma[\phi^a_{cons}] = \mathbf{true}$, completing the proof for the first part.

2. We now show that $\omega^2$ is valid.

Let $\sigma$ denote an arbitrary assignment to all free variables and to the bound Boolean variables in $\omega_{bool}$ such that $\sigma[\omega_{bool}] = \mathbf{true}$. We extend $\sigma$ with an evaluation of $x_a$ such that $\sigma[\omega_a] = \mathbf{true}$ and hence $\sigma[\omega^2] = \mathbf{true}$.

Since $\sigma[\omega_{bool}] = \mathbf{true}$, we know that $\sigma[\phi^a_{cons}] = \mathbf{true}$ (i.e., the transitivity constraints are satisfied by $\sigma$) and $\sigma[\phi^a_{bool}] = \mathbf{true}$.

Suppose we can find a value $\sigma[x_a]$ that satisfies the following equations:

$$\sigma[x_a \bowtie x_j + c] = \sigma[e^{c,\bowtie}_{a,j}] \qquad \forall j \neq a, \forall \text{ constants } c \tag{8.4}$$

$$\sigma[x_i \bowtie x_a + c] = \sigma[e^{c,\bowtie}_{i,a}] \qquad \forall i \neq a, \forall \text{ constants } c \tag{8.5}$$

Then, $\sigma[\phi^a_{bool}] = \sigma[\phi]$ because $\phi^a_{bool}$ is obtained from $\phi$ by replacing predicates $(x_a \bowtie x_j + c)$ and $(x_i \bowtie x_a + c')$ (for all $i, j$ and for all constants $c, c'$) with Boolean variables $e^{c,\bowtie}_{a,j}$ and $e^{c',\bowtie}_{i,a}$. Since $\sigma[\phi^a_{bool}] = \mathbf{true}$, $\sigma[\phi] = \mathbf{true}$, and hence $\sigma[\omega_a] = \mathbf{true}$.

A value $\sigma[x_a]$ that satisfies Equations 8.4 and 8.5 exists if:

$$\sigma[x_a] \geq \sigma[x_j] + c \qquad \text{if } \sigma[e^{c,\geq}_{a,j}] = \mathbf{true} \tag{8.6}$$

$$\sigma[x_a] < \sigma[x_j] + c \qquad \text{if } \sigma[e^{c,\geq}_{a,j}] = \mathbf{false} \tag{8.7}$$

$$\sigma[x_a] > \sigma[x_j] + c \qquad \text{if } \sigma[e^{c,>}_{a,j}] = \mathbf{true} \tag{8.8}$$

$$\sigma[x_a] \leq \sigma[x_j] + c \qquad \text{if } \sigma[e^{c,>}_{a,j}] = \mathbf{false} \tag{8.9}$$

In the above equations, w.l.o.g., we use literals encoding lower bounds on $x_a$ (e.g., $e_{a,j}^{c,\geq}$) in place of those encoding upper bounds (e.g., $e_{j,a}^{-c,>}$).

Let

$$U_a = \min_{j,c \ s.t. \ e_{a,j}^{c,\bowtie} = \mathbf{false}} (\sigma[x_j] + c)$$

and

$$L_a = \max_{j,c \ s.t. \ e_{a,j}^{c,\bowtie} = \mathbf{true}} (\sigma[x_j] + c)$$

$U_a$ and $L_a$ are respectively the tightest upper and lower bounds on $\sigma[x_a]$.

Define the ordering relation $\diamond$ as follows

$$\diamond = \begin{cases} \geq & \text{if the tightest bounds are non-strict, i.e., } \sigma[x_a] \leq U_a \text{ and } \sigma[x_a] \geq L_a \\ > & \text{otherwise} \end{cases} \tag{8.10}$$

Then, the inequalities 8.6 to 8.9 can be satisfied if:

$$U_a \ \diamond \ L_a \tag{8.11}$$

In other words, if the minimum upper bound on $\sigma[x_a]$ is greater (or greater than or equal to) the maximum lower bound on $\sigma[x_a]$.

To show that the above is true, it is enough to show that for any pair of upper and lower bounds on $\sigma[x_a]$, the relation $\diamond$ holds, and so it holds in particular for the minimum upper bound and the maximum lower bound. For example, for the two inequalities $\sigma[x_a] < \sigma[x_j] + c_1$ and $\sigma[x_a] \geq \sigma[x_k] + c_2$ to be true we need that $\sigma[x_j] + c_1 > \sigma[x_k] + c_2$.

Therefore, consider two arbitrary indices $j$ and $k$ different from $a$. We need to consider four cases based on evaluations of the Boolean literals $e_{a,j}^{c_1,\bowtie}$ and $e_{a,k}^{c_2,\bowtie}$. Note that cases in which both literals evaluate to $\mathtt{true}$ or both to $\mathbf{false}$ only give rise to two lower bounds or to two upper bounds. By the transitivity constraints of types (b) and (c), if the minimum upper bound (or maximum lower bound) is satisfied, then every other upper bound (or lower bound) will be satisfied.

The four cases are enumerated below:

(a) $e_{a,j}^{c_1,>} = \mathbf{false}, e_{a,k}^{c_2,\geq} = \mathtt{true}$.
This implies that

$$\sigma[x_j] \geq \sigma[x_a] - c_1 \text{ and } \sigma[x_a] \geq \sigma[x_k] + c_2$$

We need to show that

$$\sigma[x_j] + c_1 \geq \sigma[x_k] + c_2$$

Or

$$\sigma[x_j] \geq \sigma[x_k] + (c_2 - c_1)$$

The last inequality is true, since $\sigma$ satisfies the transitivity constraint $e_{j,a}^{-c_1,\geq} \wedge e_{a,k}^{c_2,\geq} \implies (x_j \geq x_k + c_2 - c_1)$.

(b) $e_{a,j}^{c_1,\geq} = \mathbf{false}, e_{a,k}^{c_2,>} = \mathbf{true}$.
This case is identical to the one above, with $\geq$ and $>$ interchanged.

(c) $e_{a,j}^{c_1,>} = \mathbf{false}, e_{a,k}^{c_2,>} = \mathbf{true}$.
This implies that

$$\sigma[x_j] \geq \sigma[x_a] - c_1 \text{ and } \sigma[x_a] > \sigma[x_k] + c_2$$

We need to show that

$$\sigma[x_j] + c_1 > \sigma[x_k] + c_2$$

Or

$$\sigma[x_j] > \sigma[x_k] + (c_2 - c_1)$$

The last inequality is true, since $\sigma$ satisfies the transitivity constraint $e_{j,a}^{-c_1,\geq} \wedge e_{a,k}^{c_2,>} \implies (x_j > x_k + c_2 - c_1)$.

(d) $e_{a,j}^{c_1,\geq} = \mathbf{false}, e_{a,k}^{c_2,\geq} = \mathbf{true}$.
This case is identical to the one above, with $\geq$ and $>$ interchanged.

Thus, we can conclude that Equation 8.11 is satisfied, completing the proof of this part.

$\square$

We illustrate the transformation with a simple example.

**Example 8.1** Let $\omega_a = \exists x_a.\phi$ where $\phi = x_a \leq x_0 \wedge x_1 \geq x_a \wedge x_2 \leq x_a$. Then, $\phi_{bool}^a = e_{0,a}^{\geq,0} \wedge e_{1,a}^{\geq,0} \wedge e_{a,2}^{\geq,0}$. $\phi_{cons}^a$ is the conjunction of the following constraints:

1. $e_{0,a}^{\geq,0} \wedge e_{a,2}^{\geq,0} \implies x_0 \geq x_2$
2. $e_{1,a}^{\geq,0} \wedge e_{a,2}^{\geq,0} \implies x_1 \geq x_2$

Then, $\omega_{bool} = \exists e_{0,a}^{\geq,0}, e_{1,a}^{\geq,0}, e_{a,2}^{\geq,0}.[\phi_{cons}^a \wedge \phi_{bool}^a]$ evaluates to $x_0 \geq x_2 \wedge x_1 \geq x_2$. $\square$

The quantifier transformation procedure described here works even when $\phi$ is replaced by a QDL formula with quantifiers only over Boolean variables. In the general case, $\phi$ can be replaced by $\exists e_1, e_2, \ldots, e_l.\phi'$ where $\phi'$ is a DL formula. The transformation extends to this more general case for the following reason: any satisfying assignment $\sigma$ for $\omega_a$ can be extended to one for $\omega_{bool}$ (and vice-versa), as in the proof of Theorem 8.1, keeping the partial assignment to $e_1, e_2, \ldots, e_l$ unchanged.

## 8.2 Satisfiability Checking of DL Formulas over $\mathbb{R}$

Suppose we want to decide the satisfiability of a DL formula $\phi$. The DIRECT encoding method introduced in Chapter 3 cannot directly be used as it assumes that the DL formula has integer variables and constants, and hence that every difference constraint can be re-written as a non-strict inequality.

We use a Boolean encoding algorithm that differs slightly from the DIRECT encoding algorithm and is based on the following fact: The DL formula $\phi$ is satisfiable iff the QDL formula $\omega_{1..n} = \exists x_1, x_2, \ldots, x_n.\phi$ is satisfiable.

We can transform $\omega_{1..n}$ to an equivalent QDL formula $\omega_{bool}$ with existential quantifiers only over Boolean variables encoding all difference constraints. This is done by first imposing an order on the variables $x_1, x_2, \ldots, x_n$, and then eliminating the quantifiers over those variables in that order, one at a time, using Theorem 8.1. The resulting formula $\omega_{bool}$ is a quantified Boolean formula with only existential quantifiers. Therefore, its satisfiability can be decided by simply discarding the quantifiers and using a Boolean satisfiability solver to decide the resulting Boolean formula.

The order in which variables are eliminated from $\omega_{1..n}$ can have an impact on the size of the resulting Boolean formula. For instance, suppose that $\phi = x_1 \geq x_2 \wedge x_2 \geq x_3$. If we choose to eliminate $x_2$ first, we will generate a new inequality $x_1 \geq x_3$ and a corresponding transitivity constraint. However, if instead we eliminated $x_1$ first, we will generate no transitivity constraints. Observe that none are required to preserve satisfiability.

A good variable elimination order is the one used in the DIRECT encoding algorithm in Chapter 3. For each quantified real-valued variable $x_i$, we count the number of upper and lower bound constraints for it and compute the product of the counts. (The counts are updated as new constraints are added.) Variables are eliminated in increasing order of their corresponding products.

Note that the procedure described above can be viewed as one way to implement the algorithm given by Strichman et al. [148].

## 8.3 Representation and Manipulation of DL Formulas

The material discussed up to this point does not rely on any specific representation of DL formulas. However, since we make use of Boolean methods for quantifier elimination and satisfiability solving, it is convenient to encode a DL formula $\phi$ as a Boolean formula $\beta$.

The encoding is performed as follows. Consider each difference constraint $x_i \bowtie x_j + c$ in $\phi$. As in Section 8.1, we introduce a Boolean variable $e_{i,j}^{\bowtie,c}$ for $x_i \bowtie x_j + c$, only this time we do it for every single difference constraint. Also as before, difference constraints that are negations of each other are represented by Boolean literals that are negations of each other. We then replace each difference constraint in $\phi$ by its corresponding Boolean literal. The resulting Boolean formula is $\beta$. Standard representations of Boolean functions, such as Binary Decision Diagrams (BDDs) [27], can be used to represent $\beta$.

Clearly, $\beta$, by itself, stores insufficient information for generating transitivity constraints. Therefore, we also store the 1-1 mapping of difference constraints to the Boolean literals that encode them. However, this mapping is used only lazily, i.e., when generating transitivity constraints during quantification and in deciding DL formulas.

**Substitution**

A common operation in model checking is to substitute a "next-state" version of a state variable (Boolean or real-valued) by a the "current-state" version or by an expression of the corresponding type.

Given the Boolean representation described above, we implement substitution of a real-valued variable $x_i$ by substituting the Boolean variables corresponding to difference constraints containing $x_i$. Specifically, for a real-valued variable $x_i$, we perform the substitution $[x_i \leftarrow x_k + d]$ (where $k = 0$ or $d = 0$), by replacing all Boolean variables of the form $e_{i,j}^{\bowtie,c}$ and $e_{j,i}^{\bowtie',c'}$, for all $j$, by variables $e_{k,j}^{\bowtie,c-d}$ and $e_{j,k}^{\bowtie',c'+d}$ respectively, creating fresh replacement variables if necessary.

Substitution of a Boolean variable by the Boolean encoding of a difference logic formula is done by Boolean function composition.

## 8.4 Optimizations

The quantifier elimination method presented in Section 8.1 can be optimized in a few ways.

First, we can use the Boolean structure of the QDL formula to be more selective in deciding when to add transitivity constraints. Second, the quantifier elimination method can be optimized for a special class of QDL formulas that arise commonly in model checking timed systems. We describe these two optimizations in Sections 8.4.1 and 8.4.2 respectively.

There is one other optimization, described in Section 8.4.3, that is specific to a BDD representation of DL formulas. This optimization eliminates paths in the BDD representation that violate transitivity constraints.

### 8.4.1 Determining if Bounds are Conjoined

Suppose $\phi$ is a DL formula with Boolean encoding $\beta$, and we wish to eliminate the quantifier in $\exists x_a.\phi$. As described in Section 8.1, a transitivity constraint for $x_a$ involves two Boolean literals that encode difference constraints involving $x_a$. For a syntactic representation of $\beta$, as the number of constraints grows, so does the size of $[\beta^a_{cons} \wedge \beta^a_{bool}]$, the Boolean encoding of $[\phi^a_{cons} \wedge \phi^a_{bool}]$. Further, new difference constraints can be added when a transitivity constraint is generated from an upper bound and a lower bound on $x_a$. For a BDD-based implementation, this corresponds to the addition of a new BDD variable. We would therefore like to avoid adding transitivity constraints wherever possible.

In fact, we only need to add a constraint involving an upper bound literal and a lower bound literal if they are conjoined in a minimized DNF representation of $\beta$.[1] From a geometric viewpoint, this means that we check that the predicates corresponding to the two literals are bounds for the same convex region. This check can be posed as a Boolean satisfiability problem, which is easily solved using a BDD representation of $\beta$. Let the literals be $e_1$ and $e_2$. Then, we use cofactoring and Boolean operations to compute the following Boolean formula:

$$e_1 \wedge e_2 \wedge [\beta|_{e_1=\mathbf{true}} \wedge \neg(\beta|_{e_1=\mathbf{false}})] \wedge [\beta|_{e_2=\mathbf{true}} \wedge \neg(\beta|_{e_2=\mathbf{false}})] \tag{8.12}$$

Consider the subformula $e_i \wedge [\beta|_{e_i=\mathbf{true}} \wedge \neg(\beta|_{e_i=\mathbf{false}})]$ for $i = 1, 2$. This formula represents the set of input combinations $\overline{e}$ in which $e_i$ must be set to $\mathtt{true}$ in order for $\beta(\overline{e})$ to evaluate to $\mathtt{true}$. Thus, the conjunction of the subformulas for $i = 1$ and $i = 2$ is satisfiable only if there exists a non-empty set of input combinations $\overline{e}$ in which both $e_1$ and $e_2$ must be set to $\mathtt{true}$ for $\beta(\overline{e})$ to evaluate to $\mathtt{true}$. Viewed alternately, Formula 8.12 expresses the Boolean function corresponding to the disjunction of all terms in the minimized DNF representation of $\beta$ that contain both $e_1$ and $e_2$ in true form. Therefore, if Formula 8.12 is satisfiable, it means that $e_1$ and $e_2$ are conjoined, and we must add a transitivity constraint involving them both.

Note however, that since $\beta$ does not, by itself, represent the original DL formula $\phi$, finding that $e_1$ and $e_2$ are conjoined in $\beta$ does not imply that they are bounds in the same convex region of $\phi$. However, the converse is true, so our method is sound.

### 8.4.2 Quantifier Elimination by Eliminating Upper Bounds on $x_0$

A special class of formulas that appear in the model checking of timed systems is expressed as the formula $\omega_\epsilon$ below:

$$\omega_\epsilon = \exists \epsilon . \epsilon \geq x_0 \wedge \phi + \epsilon \tag{8.13}$$

[1] A conservative, syntactic variant of this idea has been proposed earlier by Strichman [147].

In the above equation, $\phi$ is an arbitrary DL formula, and $\phi+\epsilon$ denotes the formula obtained by adding $\epsilon$ to all real variables occurring in $\phi$, computed as $\phi[x_i + \epsilon/x_i, 1 \leq i \leq n]$, where $x_1, x_2, \ldots, x_n$ are the real variables in $\phi_i$ excluding the zero variable $x_0$. Note that even though $\phi + \epsilon$ is not in QDL as described above, it can be rewritten to be in QDL; this rewriting procedure is described in Section 9.3 and we omit it here as it is not relevant to the discussion.

From a geometric viewpoint, $\phi$ is a region in $\mathbb{R}^n$ and $\omega_\epsilon$ is the shadow of $\phi$ for a light source at $\infty^n$. Examples of $\phi$ and the corresponding $\omega_\epsilon$ are shown in Figures 8.1(a) and 8.1(c) respectively.

We can transform $\omega_\epsilon$ to an equivalent DL formula $\phi_{ub}$ by eliminating upper bounds on $x_0$, i.e., Boolean variables of the form $e_{i,0}^{\bowtie,c}$. The transformation is performed iteratively in the following steps:

1. Let $\phi_0 = \phi$. Let $e_{i_1,0}^{\bowtie_1,c_1}, e_{i_2,0}^{\bowtie_2,c_2}, \ldots, e_{i_m,0}^{\bowtie_m,c_m}$ be Boolean literals encoding all upper bounds on $x_0$ that occur in $\phi$.

   Note that an upper bound literal $e_{i_j,0}^{\bowtie_j,c_j}$ occurs in $\phi$, if it appears in some term in the minimized DNF representation of $\phi$. This can be checked by evaluating the Boolean function $[\beta|_{e_{i_j,0}^{\bowtie_j,c_j}=\mathbf{true}} \wedge \neg(\beta|_{e_{i_j,0}^{\bowtie_j,c_j}=\mathbf{false}})]$, where $\beta$ is the Boolean encoding of $\phi$, and checking that it is not $\mathbf{false}$.

2. For $j = 1, 2, \ldots, m$, we construct $\phi_j$ as follows:

   (a) Replace all occurrences of $x_{i_j} \bowtie_j x_0 + c_j$ in $\phi_{j-1}$ with $e_{i_j,0}^{\bowtie_j,c_j}$ to get $\phi_{bool}^{0,j-1}$.

   (b) Construct $\phi_{cons}^{0,j-1}$, the conjunction of all transitivity constraints[2] for $x_0$ involving $e_{i_j,0}^{\bowtie_j,c_j}$ and real-valued variables in $\phi_{bool}^{0,j-1}$.

   (c) Construct the formula $\phi_j$, a disjunction of two terms:

   $$\phi_j = \{(\phi_{bool}^{0,j-1} \wedge \phi_{cons}^{0,j-1})|_{e_{i_j,0}^{\bowtie_j,c_j}=\mathbf{true}}\} \vee \{[\neg(x_{i_j} \bowtie_j x_0 + c_j)] \wedge [\phi_{bool}^{0,j-1}|_{e_{i_j,0}^{\bowtie_j,c_j}=\mathbf{false}}]\}$$

   The first disjunct is the region obtained by dropping the bound $x_{i_j} \bowtie_j x_0 + c_j$ from convex sub-regions of $\phi_{j-1}$ where it is a lower bound on $x_{i_j}$, while enforcing existing and transitively implied bounds. The second disjunct corresponds to sub-regions where $\neg(x_{i_j} \bowtie_j x_0 + c_j)$ is an upper bound; these regions are left unchanged.

The output of the above transformation, $\phi_{ub}$, is given by $\phi_{ub} = \phi_m$. The correctness of this procedure is formalized in the following theorem.

**Theorem 8.2** *$\omega_\epsilon$ and $\phi_{ub}$ are equivalent.*

*Proof:* We make use of the following lemmas.

[2]We can use the optimization technique of Section 8.4.1 in this step.

**Lemma 8.1** *For all* $j = 1, \ldots, m$, $\exists\epsilon.\epsilon \geq x_0 \wedge \phi_{j-1} + \epsilon$ *is equivalent to* $\exists\epsilon.\epsilon \geq x_0 \wedge \phi_j + \epsilon$.

*Proof:*(Lemma 8.1)
We give the proof for an arbitrary $j$ satisfying $1 \leq j \leq m$. Let $\omega_{j-1}$ and $\omega_j$ respectively denote $\exists\epsilon_{j-1}.\epsilon_{j-1} \geq x_0 \wedge \phi_{j-1} + \epsilon_{j-1}$ and $\exists\epsilon_j.\epsilon_j \geq x_0 \wedge \phi_j + \epsilon_j$. Notice that we have renamed the bound variable $\epsilon$.

1. First, we show that $\omega_{j-1} \implies \omega_j$. Let $\sigma$ be an assignment to the free and bound variables in $\omega_{j-1}$ such that $\sigma[\omega_{j-1}] = \mathbf{true}$. This means that $\sigma[\phi_{j-1} + \epsilon_{j-1}] = \mathbf{true}$. Extend $\sigma$ so that $\sigma[\epsilon_j] = \sigma[\epsilon_{j-1}]$. Thus, $\sigma[\epsilon_{j-1} \geq x_0] = \sigma[\epsilon_j \geq x_0] = \mathbf{true}$.

   We consider two cases.

   (a) *Case 1:* $\sigma[(x_{i_j} \bowtie_j x_0 + c_j) + \epsilon_{j-1}] = \mathbf{true}$.

   Note that by construction,

   $$\phi_{bool}^{0,j-1} = \phi_{j-1}[e_{i_j,0}^{\bowtie_j,c_j} / (x_{i_j} \bowtie_j x_0 + c_j)]$$

   From the two equalities above, and since $\sigma[\epsilon_j] = \sigma[\epsilon_{j-1}]$, we get

   $$\sigma[\phi_{j-1} + \epsilon_{j-1}] = \sigma[\phi_{bool}^{0,j-1}|_{e_{i_j,0}^{\bowtie_j,c_j} = \mathbf{true}} + \epsilon_j]$$

   In addition, the transitivity constraints are satisfied, i.e.,

   $$\sigma[\phi_{cons}^{0,j-1}|_{e_{i_j,0}^{\bowtie_j,c_j} = \mathbf{true}} + \epsilon_j] = \mathbf{true}$$

   because $\phi_{cons}^{0,j-1}|_{e_{i_j,0}^{\bowtie_j,c_j} = \mathbf{true}} + \epsilon_j$ only involves real-valued variables. Therefore,

   $$\sigma[\phi_{j-1} + \epsilon_{j-1}] = \sigma[(\phi_{bool}^{0,j-1} \wedge \phi_{cons}^{0,j-1})|_{e_{i_j,0}^{\bowtie_j,c_j} = \mathbf{true}} + \epsilon_j]$$

   Thus, we conclude that

   $$\sigma[\phi_{j-1} + \epsilon_{j-1}] = \sigma[\phi_j + \epsilon_j] = \mathbf{true}$$

   which in turn implies that

   $$\sigma[\epsilon_{j-1} \geq x_0 \wedge \phi_{j-1} + \epsilon_{j-1}] = \sigma[\epsilon_j \geq x_0 \wedge \phi_j + \epsilon_j] = \mathbf{true}$$

   and so

   $$\sigma[\omega_{j-1}] = \sigma[\omega_j] = \mathbf{true}$$

   This concludes the first case.

(b) *Case 2:* $\sigma[(x_{i_j} \bowtie_j x_0 + c_j) + \epsilon_{j-1}] = \mathbf{false}$.

Since
$$\phi_{bool}^{0,j-1} = \phi_{j-1}[e_{i_j,0}^{\bowtie_j,c_j} / (x_{i_j} \bowtie_j x_0 + c_j)]$$
and, in addition, $\sigma[\epsilon_j] = \sigma[\epsilon_{j-1}]$, we have
$$\sigma[\phi_{j-1} + \epsilon_{j-1}] = \sigma[\phi_{bool}^{0,j-1}|_{e_{i_j,0}^{\bowtie_j,c_j}=\mathbf{false}} + \epsilon_j]$$
Now, since $\sigma[\phi_{j-1} + \epsilon_{j-1}] = \mathbf{true}$, we get
$$\sigma[\phi_{bool}^{0,j-1}|_{e_{i_j,0}^{\bowtie_j,c_j}=\mathbf{false}} + \epsilon_j] = \mathbf{true}$$
and
$$\sigma[[\neg(x_{i_j} \bowtie_j x_0 + c_j) \wedge \phi_{bool}^{0,j-1}|_{e_{i_j,0}^{\bowtie_j,c_j}=\mathbf{false}}] + \epsilon_j] = \mathbf{true}$$
and so, we conclude that
$$\sigma[\phi_j + \epsilon_j] = \sigma[\epsilon_j \geq x_0 \wedge \phi_j + \epsilon_j] = \sigma[\omega_j] = \mathbf{true}$$
which concludes case 2.

Thus, $\omega_{j-1} \implies \omega_j$.

2. We next show that $\omega_j \implies \omega_{j-1}$.

Let $\sigma$ be an assignment to the free and bound variables in $\omega_j$ such that $\sigma[\omega_j] = \mathbf{true}$. This means that $\sigma[\phi_j + \epsilon_j] = \mathbf{true}$. We wish to extend $\sigma$ by an assignment to $\epsilon_{j-1}$ so that $\sigma[\phi_{j-1} + \epsilon_{j-1}] = \mathbf{true}$ and $\sigma[\epsilon_{j-1} \geq x_0] = \mathbf{true}$.

We consider two cases.

(a) *Case 1:* $\sigma[(\phi_{bool}^{0,j-1} \wedge \phi_{cons}^{0,j-1})|_{e_{i_j,0}^{\bowtie_j,c_j}=\mathbf{true}} + \epsilon_j] = \mathbf{true}$.

Therefore,
$$\sigma[\phi_{bool}^{0,j-1}|_{e_{i_j,0}^{\bowtie_j,c_j}=\mathbf{true}} + \epsilon_j] = \mathbf{true} \tag{8.14}$$
and
$$\sigma[\phi_{cons}^{0,j-1}|_{e_{i_j,0}^{\bowtie_j,c_j}=\mathbf{true}} + \epsilon_j] = \mathbf{true}$$
If $\sigma[(x_{i_j} \bowtie_j x_0 + c_j) + \epsilon_j] = \mathbf{true}$, then using the equality
$$\phi_{bool}^{0,j-1} = \phi_{j-1}[e_{i_j,0}^{\bowtie_j,c_j} / (x_{i_j} \bowtie_j x_0 + c_j)] \tag{8.15}$$
we can set $\sigma[\epsilon_{j-1}] = \sigma[\epsilon_j]$, which yields $\sigma[(x_{i_j} \bowtie_j x_0 + c_j) + \epsilon_{j-1}] = \mathbf{true}$, and so using Equations 8.14 and 8.15, we get
$$\sigma[\phi_{j-1} + \epsilon_{j-1}] = \sigma[\phi_j + \epsilon_j] = \mathbf{true} \tag{8.16}$$

However, if $\sigma[(x_{i_j} \bowtie_j x_0 + c_j) + \epsilon_j] = \mathbf{false}$, then we must find an alternate assignment to $\epsilon_{j-1}$, such that $\sigma[(x_{i_j} \bowtie_j x_0 + c_j) + \epsilon_{j-1}] = \mathbf{true}$. Then, we can conclude, as above, that Equation 8.16 holds.

Consider, w.r.t. the assignment $\sigma$, all lower bounds on $x_0$ that occur in $\phi_{j-1} + \epsilon_j$ (and hence in $\phi_{bool}^{0,j-1} + \epsilon_j$); more precisely, a lower bound on $x_0$ is a predicate $(x_0 \bowtie_k x_{i_k} + c_k) + \epsilon_j$ such that $\sigma[(x_0 \bowtie_k x_{i_k} + c_k) + \epsilon_j] = \mathbf{true}$.

If no such lower bound on $x_0$ exists, then we can set $\epsilon_{j-1}$ to any value that results in $\sigma[(x_{i_j} \bowtie_j x_0 + c_j) + \epsilon_{j-1}] = \mathbf{true}$, because there is no lower bound to be violated by increasing the value of a real-valued variable.

So suppose at least one lower bound on $x_0$ exists in $\phi_{j-1}$. Define the value $v_s$ as

$$v_s = \min_{k\ s.t.\ \sigma[(x_0 \bowtie_k x_{i_k} + c_k) + \epsilon_j] = \mathbf{true}} (-c_k - \sigma[x_{i_k} + \epsilon_j]) \tag{8.17}$$

Note that $v_s \geq 0$ since $\sigma[(x_0 \bowtie_k x_{i_k} + c_k) + \epsilon_j] = \mathbf{true}$ for all $k$ in Equation 8.17.

Let $l$ be the $k$ for which the minimum on the right-hand side of Equation 8.17 is attained. If there are many such $k$, say $k_1, k_2, \ldots, k_d$, set $l$ according to the following rules:

i. If there exists $k_i$ for which $\bowtie_{k_i} = >$, set $l$ to any one such $k_i$.
ii. Otherwise select $l$ to be any one of $k_1, k_2, \ldots, k_d$.

Thus,

$$v_s = -c_l - \sigma[x_{i_l} + \epsilon_j] \tag{8.18}$$

Next, we define a positive real number $\chi$ as follows:

$$\chi = \begin{cases} \chi_0 & \text{if } \bowtie_l = >\text{, and where } \chi_0 \in (0, \sigma[x_{i_j} - x_{i_l} - c_j - c_l]) \\ 0 & \text{otherwise} \end{cases} \tag{8.19}$$

Note that $\sigma[x_{i_j} - x_{i_l} - c_j - c_l]$ is non-negative and is strictly positive when $\bowtie_l = >$. This is because there exists a transitivity constraint in $\phi_{cons}^{0,j-1}$ of the form

$$(e_{i_j,0}^{\bowtie_j, c_j} \wedge x_0 \bowtie_l x_{i_l} + c_l) \implies (x_{i_j} \bowtie_j x_{i_l} + c_j + c_l)$$

which occurs in $\phi_{cons}^{0,j-1}|_{e_{i_j,0}^{\bowtie_j, c_j} = \mathbf{true}}$ as

$$(x_0 \bowtie_l x_{i_l} + c_l) \implies (x_{i_j} \bowtie_j x_{i_l} + c_j + c_l)$$

If $\bowtie_j \neq \bowtie_l$, the following constraint also holds:

$$(x_0 \bowtie_l x_{i_l} + c_l) \implies (x_{i_j} \bowtie_l x_{i_l} + c_j + c_l)$$

Since $\sigma[(x_0 \bowtie_l x_{i_l} + c_l) + \epsilon_j] = \mathbf{true}$, the following equalities hold:

$$\sigma[(x_{i_j} \bowtie_j x_{i_l} + c_j + c_l) + \epsilon_j] = \sigma[x_{i_j} \bowtie_j x_{i_l} + c_j + c_l] = \mathbf{true} \tag{8.20}$$

$$\sigma[(x_{i_j} \bowtie_l x_{i_l} + c_j + c_l) + \epsilon_j] = \sigma[x_{i_j} \bowtie_l x_{i_l} + c_j + c_l] = \mathbf{true} \tag{8.21}$$

Thus, $\sigma[x_{i_j} - x_{i_l} - c_j - c_l]$ is non-negative and is strictly positive when $\bowtie_l = >$.

We now show that $v_s - \chi \geq 0$. If $\chi = 0$, clearly $v_s - \chi \geq 0$. So, assume that $\bowtie_l = >$, and thus $\chi \in (0, \sigma[x_{i_j} - x_{i_l} - c_j - c_l])$. Then we can conclude the following:

$$\begin{aligned}
v_s - \chi &= -c_l - \sigma[x_{i_l}] - \sigma[\epsilon_j] - \chi \\
&> -c_l - \sigma[x_{i_l}] - \sigma[\epsilon_j] - \sigma[x_{i_j} - x_{i_l} - c_j - c_l] \\
&= -c_l - \sigma[x_{i_l}] - \sigma[\epsilon_j] - \sigma[x_{i_j}] + \sigma[x_{i_l}] + c_j + c_l \\
&= c_j - \sigma[x_{i_j}] - \sigma[\epsilon_j] \\
&\geq 0 \quad (\text{since } \sigma[(x_{i_j} \bowtie_j x_0 + c_j) + \epsilon_j] = \mathbf{false})
\end{aligned}$$

Intuitively, $v_s - \chi$ is a non-negative real number we can add to all real-valued variables without violating lower bounds on $x_0$ in $\phi_{j-1} + \epsilon_j$.

Now, define $\sigma[\epsilon_{j-1}]$ as follows:

$$\sigma[\epsilon_{j-1}] = \sigma[\epsilon_j] + v_s - \chi \tag{8.22}$$

Since $v_s - \chi \geq 0$, $\sigma[\epsilon_{j-1}] \geq \sigma[\epsilon_j]$.

Given the above assignment to $\epsilon_{j-1}$, we first show that $\sigma[(x_{i_j} \bowtie_j x_0 + c_j) + \epsilon_{j-1}] = \mathbf{true}$. We have the following sequence of equalities:

$$\begin{aligned}
&\sigma[(x_{i_j} \bowtie_j x_0 + c_j) + \epsilon_{j-1}] \\
= \; &\sigma[x_{i_j}] + \sigma[\epsilon_{j-1}] \bowtie_j c_j \\
= \; &\sigma[x_{i_j}] \bowtie_j c_j - \sigma[\epsilon_{j-1}] \\
= \; &\sigma[x_{i_j}] \bowtie_j c_j - v_s + \chi - \sigma[\epsilon_j] \\
= \; &\sigma[x_{i_j}] \bowtie_j \chi + c_j - \min_k(-\sigma[x_{i_k} + \epsilon_j] - c_k) - \sigma[\epsilon_j] \\
= \; &\sigma[x_{i_j}] \bowtie_j \chi + c_j + (\sigma[x_{i_l} + \epsilon_j] + c_l) - \sigma[\epsilon_j] \\
= \; &\sigma[x_{i_j}] \bowtie_j \chi + \sigma[x_{i_l}] + c_j + c_l \\
= \; &\mathbf{true} \quad (\text{since } \chi \in (0, \sigma[x_j - x_l - c_j - c_l]) \text{ and from Eqn. 8.20})
\end{aligned}$$

We next show that the assignment to $\epsilon_{j-1}$ in Equation 8.22 preserves the truth assignment to other bounds on $x_0$; i.e., bounds in $\phi_{j-1} + \epsilon_j$ other than $(x_{i_j} \bowtie_j x_0 + c_j) + \epsilon_j$. Formally, we show that for all bounds $x_0 \bowtie_k x_{i_k} + c_k$ where $k \neq j$:

$$\sigma[(x_0 \bowtie_k x_{i_k} + c_k) + \epsilon_{j-1}] = \sigma[(x_0 \bowtie_k x_{i_k} + c_k) + \epsilon_j]$$

Note that the value of difference constraints of the form $x_{i_{k_1}} \bowtie x_{i_{k_2}} + c_{k_1 k_2}$ is unaffected by the assignment to $\epsilon_j$ or $\epsilon_{j-1}$.

If $\sigma[(x_0 \bowtie_k x_{i_k} + c_k) + \epsilon_j] = \mathbf{false}$, then $\sigma[(x_0 \bowtie_k x_{i_k} + c_k) + \epsilon_{j-1}] = \mathbf{false}$, since $\sigma[\epsilon_{j-1}] \geq \sigma[\epsilon_j]$.

On the other hand, if $\sigma[(x_0 \bowtie_k x_{i_k} + c_k) + \epsilon_j] = \mathtt{true}$, then

$$\begin{aligned}
& \sigma[(x_0 \bowtie_k x_{i_k} + c_k) + \epsilon_{j-1}] \\
= \; & 0 \bowtie_k \sigma[x_{i_k}] + c_k + \sigma[\epsilon_{j-1}] \\
= \; & 0 \bowtie_k \sigma[x_{i_k}] + c_k + \sigma[\epsilon_j] + v_s - \chi \\
= \; & 0 \bowtie_k (c_k + \sigma[x_{i_k}]) + \sigma[\epsilon_j] + (-c_l - \sigma[x_{i_l} + \epsilon_j]) - \chi \\
= \; & (-c_k - \sigma[x_{i_k}]) \bowtie_k (-c_l - \sigma[x_{i_l}]) - \chi \\
= \; & \mathtt{true} \quad \text{(since } \chi \geq 0 \text{ and from Equations 8.17 and 8.18)}
\end{aligned}$$

To sum up, we have shown that $\sigma[(x_{i_j} \bowtie_j x_0 + c_j) + \epsilon_{j-1}] = \mathtt{true}$, even though $\sigma[(x_{i_j} \bowtie_j x_0 + c_j) + \epsilon_j] = \mathbf{false}$. Thus, we can conclude that

$$\sigma[\phi_{j-1} + \epsilon_{j-1}] = \sigma[\phi_j + \epsilon_j] = \mathtt{true}$$

This completes the proof for the first case.

(b) *Case 2:* $\sigma[[\neg(x_{i_j} \bowtie_j x_0 + c_j) \wedge \phi_{bool}^{0,j-1}|_{e_{i_j,0}^{\bowtie_j,c_j}=\mathbf{false}}] + \epsilon_j] = \mathtt{true}$.

Thus

$$\sigma[\phi_{bool}^{0,j-1}|_{e_{i_j,0}^{\bowtie_j,c_j}=\mathbf{false}} + \epsilon_j] = \mathtt{true}$$

and

$$\sigma[(x_{i_j} \bowtie_j x_0 + c_j) + \epsilon_j] = \mathbf{false}$$

Letting $\sigma[\epsilon_{j-1}] = \sigma[\epsilon_j]$ and from Equation 8.15, we get

$$\sigma[\phi_{j-1} + \epsilon_{j-1}] = \mathtt{true}$$

as required.

Thus, $\omega_j \implies \omega_{j-1}$.

From parts 1 and 2 above, we conclude that $\omega_{j-1}$ and $\omega_j$ are equivalent.

□

**Lemma 8.2** *Suppose the DL formula $\phi$ does not contain any difference constraints that are upper bounds on $x_0$; i.e., any satisfying assignment to $\phi$ sets all upper bounds on $x_0$ to* $\mathbf{false}$*, and all lower bound predicates to* $\mathtt{true}$*. Then, $\exists \epsilon . \epsilon \geq x_0 \wedge \phi + \epsilon$ is equivalent to $\phi$.*

*Proof:*(Lemma 8.2)
We first show that $\phi \implies (\exists\epsilon.\epsilon \geq x_0 \wedge \phi + \epsilon)$.

Let $\sigma$ be an assignment to the variables in $\phi$ such that $\sigma[\phi] = \mathbf{true}$. We extend $\sigma$ with an evaluation of $\epsilon$ so that $\sigma[\epsilon] = 0 = \sigma[x_0]$. Then, $\sigma[\epsilon \geq x_0 \wedge \phi + \epsilon] = \mathbf{true}$, since $\sigma[\phi + \epsilon] = \sigma[\phi]$. Therefore, $\sigma[\exists\epsilon.\epsilon \geq x_0 \wedge \phi + \epsilon] = \mathbf{true}$. Thus, $\phi \implies (\exists\epsilon.\epsilon \geq x_0 \wedge \phi + \epsilon)$.

Next, we show that $(\exists\epsilon.\epsilon \geq x_0 \wedge \phi + \epsilon) \implies \phi$. Let $\sigma$ be an assignment such that $\sigma[\exists\epsilon.\epsilon \geq x_0 \wedge \phi + \epsilon] = \mathbf{true}$. Thus, $\sigma[\epsilon \geq x_0] = \mathbf{true}$ and $\sigma[\phi + \epsilon] = \mathbf{true}$. Since $\phi$ does not contain any difference constraints that are upper bounds on $x_0$, for any lower bound $x_0 \bowtie_k x_k + c_k$ on $x_0$, $\sigma[(x_0 \bowtie_k x_k + c_k) + \epsilon] = \mathbf{true}$ and for an upper bound $x_l \bowtie_l x_0 + c_l$ on $x_0$, $\sigma[(x_l \bowtie_l x_0 + c_l) + \epsilon] = \mathbf{false}$.

Then, since $\sigma[\epsilon] \geq 0$,

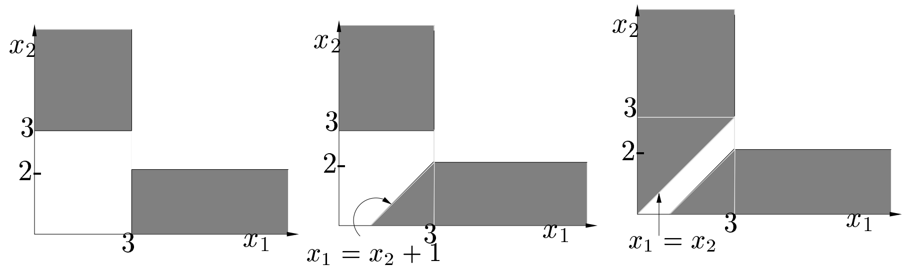
$$\sigma[(x_0 \bowtie_k x_k + c_k) + \epsilon] = \mathbf{true} = \sigma[x_0 \bowtie_k (x_k + \epsilon) + c_k] = \sigma[x_0 \bowtie_k x_k + c_k]$$

Similarly, for an upper bound predicate on $x_0$, $\sigma[x_l \bowtie_l x_0 + c_l] = \mathbf{false}$.

It then follows that $\sigma[\phi] = \mathbf{true}$.

□

From Lemma 8.1, we infer that $\omega_\epsilon = \exists\epsilon.\epsilon \geq x_0 \wedge \phi_0 + \epsilon$ is equivalent to $\exists\epsilon.\epsilon \geq x_0 \wedge \phi_m + \epsilon$. Additionally, since $\phi_m$ does not contain any upper bounds on $x_0$, using Lemma 8.2, we conclude that $\omega_\epsilon$ is equivalent to $\phi_m = \phi_{ub}$. This completes the proof of Theorem 8.2. □




(a) $\phi_0 = \phi$ (b) $\phi_1$ (c) $\phi_2 = \omega_\epsilon$

Figure 8.1: **Eliminating upper bounds on** $x_0$

**Example 8.2** Let the subformula $\phi$ of $\omega_\epsilon$ be

$$\phi = (x_1 \geq x_0 + 3 \wedge x_2 \leq x_0 + 2) \vee (x_1 < x_0 + 3 \wedge x_2 \geq x_0 + 3)$$

$\phi$ is depicted geometrically as the shaded region in Figure 8.1(a). It comprises two sub-regions, one for each disjunct. The lower bounds on these regions, $x_1 \geq x_0 + 3$ and $x_2 \geq x_0 + 3$, are upper bounds on $x_0$. We encode these by $e^{\geq,3}_{1,0}$ and $e^{\geq,3}_{2,0}$.

Figure 8.1(b) shows $\phi_1$, the result of eliminating $e_{1,0}^{\geq,3}$. Formally, we calculate

$$\begin{aligned}\phi_{bool}^{0,0} &= (e_{1,0}^{\geq,3} \wedge x_2 \leq x_0 + 2) \vee (\neg e_{1,0}^{\geq,3} \wedge x_2 \geq x_0 + 3)\\ \phi_{cons}^{0,0} &= (e_{1,0}^{\geq,3} \wedge x_2 \leq x_0 + 2) \implies (x_1 \geq x_2 + 1)\end{aligned}$$

Then, applying step 2(c) of the transformation, we get

$$\phi_1 = (x_2 \leq x_0 + 2 \wedge x_1 \geq x_2 + 1) \vee (x_1 < x_0 + 3 \wedge x_2 \geq x_0 + 3)$$

Similarly, in the next iteration, we introduce and eliminate $e_{2,0}^{\geq,3}$ to get $\phi_2$, shown in Figure 8.1(c), which is equivalent to $\omega_\epsilon$. $\square$

### 8.4.3 Eliminating Infeasible Paths in BDDs

Suppose $\beta$ is the Boolean encoding of DL formula $\phi$. Let $\phi_{cons}$ denote the conjunction of transitivity constraints for all real-valued variables in $\phi$, and let $\beta_{cons}$ denote its Boolean encoding. Finally, denote the BDD representations of $\beta$ and $\beta_{cons}$ by $\mathsf{Bdd}(\beta)$ and $\mathsf{Bdd}(\beta_{cons})$ respectively.

We would like to eliminate paths in $\mathsf{Bdd}(\beta)$ that violate transitivity constraints, i.e., those corresponding to assignments to variables in $\beta$ for which $\beta_{cons} = \textbf{false}$. We can do this by using the BDD `Restrict` operator, replacing $\mathsf{Bdd}(\beta)$ by `Restrict(`$\mathsf{Bdd}(\beta), \mathsf{Bdd}(\beta_{cons})$`)`. Informally, `Restrict(`$\mathsf{Bdd}(\beta)$`, `$\mathsf{Bdd}(\beta_{cons})$`)` traverses $\mathsf{Bdd}(\beta)$, eliminating a path on which $\beta_{cons}$ is **false** as long as it doesn't involve adding new nodes to the resulting BDD. Details about the `Restrict` operator may be found in the paper by Coudert and Madre [44].

Since eliminating infeasible paths in a large BDD can be quite time consuming, we do not apply this optimization very often. For example, in model checking timed automata, this optimization is applied only to the BDD for the set of reachable states, and only once on each fixpoint iteration.

## 8.5 Summary

This chapter showed how to eliminate quantifiers over real-valued variables in a quantified difference logic (QDL) formula by transforming the problem to one of eliminating quantifiers over Boolean variables from a quantified Boolean formula. Satisfiability solving of DL formulas over Boolean and real-valued variables was discussed, as also were techniques of representing and manipulating DL formulas. Several optimizations can be used to improve on the quantifier elimination method in practice.

In the next chapter, we will see how the Boolean methods for QDL discussed in this chapter can be applied to the problem of model checking timed automata.

# Chapter 9

# Model Checking and Timed Circuits

A *timed system* is a generalization of a finite-state system with real-valued *clock* or *timer* variables. A particularly expressive formalism for timed systems is the *timed automaton* [3, 5].

A timed automaton is a generalization of a finite automaton with a set of real-valued clock variables. The state space of a timed automaton thus has a finite component (over Boolean state variables) and an infinite component (over clock variables). Several model checking techniques for timed automata have been proposed over the past 15 years. These can be classified, on the one hand, as being either *symbolic* or *fully symbolic*, and on the other, as being *bounded* or *unbounded*. Symbolic techniques use a symbolic representation for the infinite component of the state space, and explicit representations for the finite component. In contrast, fully symbolic methods employ a single symbolic representation for both finite and infinite components of the state space. Bounded model checking techniques work by unfolding the transition relation $d$ times, finding counterexamples of length up to $d$, if they exist. As in the untimed case, these methods suffer from the limitation that, unless a bound on the length of counterexamples is known, they cannot verify the property of interest. Unbounded methods, on the other hand, can produce a guarantee of correctness.

The theoretical foundation for unbounded, fully symbolic model checking of timed automata was laid by Henzinger et al. [71]. The characteristic function of a set of states is a formula in difference logic (DL). The most important model checking operations involve deciding DL formulas and eliminating quantifiers on real variables from quantified difference logic (QDL) formulas.

This chapter describes the first approach to unbounded, fully symbolic model checking of timed automata that is based on a Boolean encoding of DL formulas and that preserves the interpretation of clocks over the reals. Unlike some other fully symbolic techniques, our method can be used to model check any property in the timed $\mu$ calculus or Timed Computation Tree Logic (TCTL) [4]. The method is based on the results of Chapter 8, and especially on the technique for transforming the problem of eliminating quantifiers on real variables to one of eliminating quantifiers on Boolean variables.

We begin this chapter with a discussion of related work. Section 9.2 gives background information on timed automata and the timed $\mu$ calculus. We describe our fully symbolic model checking algorithm in Section 9.3, including a description of our implementation and results on a toy example. Section 9.4 describes our experience applying this model checking algorithm to the verification of timed circuits.

## 9.1 Related Work

We discuss the related work that is most relevant to our approach to fully symbolic model checking of timed automata. A more detailed survey of techniques for model checking timed systems can be found in the recent paper by Wang [162].

The work that is most closely related to ours is the approach based on representing DL formulas using Difference Decision Diagrams (DDDs) [102]. A DDD is a BDD-like data structure, where the node labels are generalized to be difference constraints rather than just Boolean variables, with the ordering of constraints induced by an ordering of clock variables. This constraint ordering permits the use of local reduction operations, such as eliminating inconsistent combinations of two constraints that involve the same pair of clock variables. Deciding a DL formula represented as a DDD is done by eliminating all inconsistent paths in the DDD. This is done by enumerating all paths in the DDD and checking the satisfiability of the conjunction of constraints on each path using a constraint solver based on the Bellman-Ford shortest path algorithm. Note that each path can be viewed as a disjunct in the Disjunctive Normal Form (DNF) representation of the DDD, and in the worst case there can be exponentially many calls to the constraint solver. Quantifier elimination is performed by the Fourier-Motzkin technique [49], which also requires enumerating all possible paths. In contrast, our Boolean encoding method is general in that any representation of Boolean functions may be used. Our decision procedure and quantifier elimination scheme use a direct translation to SAT and Boolean quantification, respectively, avoiding the need to explicitly enumerate each DNF term. In theory, the use of DDDs permits unbounded, fully symbolic model checking of TCTL; however, the DDD-based model checker [102] can only check reachability properties (these can express safety and bounded-liveness properties [1]).

UPPAAL2K and KRONOS are unbounded, symbolic model checkers that explicitly enumerate the discrete component of the state space. KRONOS uses Difference Bound Matrices (DBMs) as the symbolic representation [168] of the infinite component. UPPAAL2K uses, in addition, Clock Difference Diagrams (CDDs) to symbolically represent unions of convex clock regions [15]. In a CDD, a node is labeled by the difference of a pair of clock variables, and each outgoing edge from a node is labeled with an interval bounding that difference. Note that while KRONOS can check arbitrary

TCTL formulas, UPPAAL2K is limited to checking reachability properties and very restricted liveness properties such as $\mathbf{AF}p$.

RED is an unbounded, fully symbolic model checker based on a data structure called the Clock Restriction Diagram (CRD) [161]. The CRD is similar to a CDD, labeling each node with the difference between two clock variables. However, each outgoing edge from a node is labeled with an upper bound, instead of an interval. RED represents difference formulas by a combined BDD-CRD structure, and can model check TCTL formulas.

A fully symbolic version of KRONOS using BDDs has been developed by interpreting clock variables over integers [24]; however, this approach is restricted to checking reachability for the subclass of closed timed automata[1], and the encoding blows up with the size of the integer constants. Rabbit [18] is a tool based on this approach that additionally exploits compositional methods to find good BDD variable orderings. In comparison, our technique applies to all timed automata and its efficiency is far less sensitive to the size of constants. Also, the variable ordering methods used in Rabbit could be used in a BDD-based implementation of our technique.

Many fully symbolic, but bounded model checking methods based on SAT have been developed (e.g., [9, 114]). McMillan [100] has recently combined bounded model checking methods with an interpolating theorem prover to perform unbounded model checking of a sub-class of infinite-state systems that includes timed automata.

## 9.2 Background

We begin with a brief presentation of background material, based on papers by Alur [3] and Henzinger et al. [71]. We refer the reader to these papers for details.

### 9.2.1 Timed Automata

A *timed automaton* $\mathcal{T}$ is a tuple $\langle \mathcal{L}, \mathcal{L}_0, \Sigma, \mathcal{X}, \mathcal{I}, \mathcal{E} \rangle$, where $\mathcal{L}$ is a finite set of locations, $\mathcal{L}_0 \subseteq \mathcal{L}$ is a finite set of initial locations, $\Sigma$ is a finite set of labels used for product construction, $\mathcal{X}$ is a finite set of non-negative real-valued clock variables, $\mathcal{I}$ is a function mapping a location to a DL formula (called a *location invariant*), and $\mathcal{E}$ is the transition relation, a subset of $\mathcal{L} \times \Psi \times \mathcal{R} \times \Sigma \times \mathcal{L}$, where $\Psi$ is a set of DL formulas that form enabling *guard* conditions for each transition, and $\mathcal{R}$ is a set of *clock reset assignments*. A location invariant is the condition under which the system can stay in that location. A clock reset assignment is of the form $x_i := x_0 + c$ or $x_i := x_j$, where $x_i, x_j \in \mathcal{X}$ and $c$ is an integer constant,[2] and indicates that the clock variable on the left-hand side of the assignment

[1] Clock constraints in a closed timed automaton do not contain strict inequalities.

[2] The assignment $x_i := c$ is represented as $x_i := x_0 + c$. Wherever we use $x_i$ to denote a clock variable, $i > 0$.

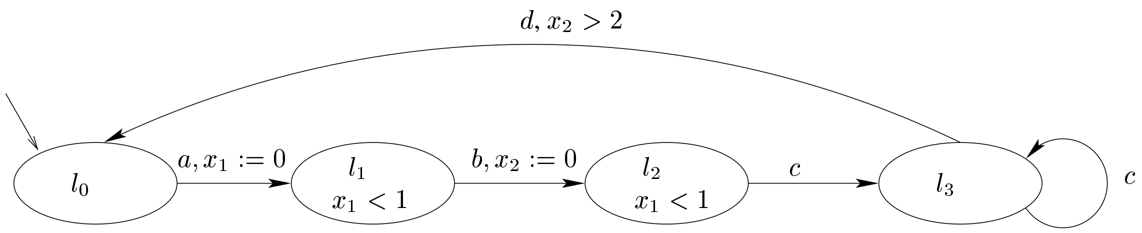



Figure 9.1: **Example of a timed automaton.** Reproduced from [3].

is reset to the value of the expression on the right-hand side. We will denote guards by $\psi, \psi_1, \ldots$.

**Example 9.1** An example of a timed automaton is given in Figure 9.1.

For this example, $\mathcal{L} = \{l_0, l_1, l_2, l_3\}$, $\mathcal{L}_0 = \{l_0\}$, $\Sigma = \{a, b, c, d\}$, $\mathcal{X} = \{x_1, x_2\}$, $\mathcal{I}(l_0) = \mathcal{I}(l_3) =$ $\mathtt{true}$, $\mathcal{I}(l_1) = \mathcal{I}(l_2) = x_1 < 1$. The latter location invariant ensures that the transition labeled $c$ from $l_2$ to $l_3$ occurs within 1 time unit of the occurrence of $a$. Similarly, the guard $x_2 > 2$ on the transition from $l_3$ to $l_0$ ensures that the time between that transition and the one labeled with $b$ is at least 2 units. $\Box$

Two timed automata are composed by synchronizing over common labels. We refer the reader to Alur's paper [3] for a formal definition of product construction. Note that in contrast to the definition of timed automata given by Alur [3], we allow location invariants and guards to be arbitrary DL formulas, rather than simply conjunctions over difference constraints involving clock variables.

The invariant $\mathcal{I}_{\mathcal{T}}$ for the timed automaton $\mathcal{T}$ is defined as $\mathcal{I}_{\mathcal{T}} = \bigwedge_{l \in \mathcal{L}} [enc(l) \implies \mathcal{I}(l)]$, where $enc(l)$ denotes the Boolean encoding of location $l$. We will also denote a transition $t \in \mathcal{E}$ as $\psi \implies \mathcal{A}$, where $\psi$ is a guard condition over both Boolean state variables (used to encode locations) and clock variables of the system, and $\mathcal{A}$ is a set of assignments to clock and Boolean state variables.

### Timed Guarded Commands

Henzinger et al. [71] show how timed automata can be expressed as *timed guarded command programs*. A *guarded command* is of the form $\psi \implies \mathcal{A}$, where $\psi$ is a guard condition over both Boolean state variables (used to encode locations) and clock variables of the system, and $\mathcal{A}$ is a set of assignments to clock and Boolean state variables. In general, we have one guarded command corresponding to each transition between two locations. A *timed guarded command program* corresponding to a timed automaton is a pair $(\mathcal{P}, \mathcal{I}_{\mathcal{P}})$ where $\mathcal{P}$ is a set of guarded commands, and $\mathcal{I}_{\mathcal{P}}$ is the program invariant defined as $\mathcal{I}_{\mathcal{P}} = \mathcal{I}_{\mathcal{T}}$.

We will use the *timed guarded command program* representation of a *timed automaton* where suitable.

### 9.2.2 Timed $\mu$ Calculus and TCTL

We express properties of timed automata in a generalization of the $\mu$ calculus called the *timed* $\mu$ ($\mathbf{T}\mu$) calculus. A formula $\varphi$ of the $\mathbf{T}\mu$ calculus is generated by the following grammar:

$$\varphi ::= X \mid \phi \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \triangleright \varphi_2 \mid z.\varphi \mid \mu X.\varphi \mid \nu X.\varphi \tag{9.1}$$

$z$ is a *specification clock variable* (i.e., $z \notin \mathcal{X}$) and $X$ is a *formula variable* used in fixpoint computation. The formula $\varphi_1 \triangleright \varphi_2$ means that the formula $\varphi_1$ is true at the present state, and remains true (as time elapses) until some transition is taken, at which time formula $\varphi_2$ becomes true; thus "$\triangleright$" is essentially a next-state operator. The formula $z.\varphi$ is true in a state where $\varphi$ is true after setting specification clock variable $z$ to zero. The expression $\mu X.\varphi$ stands for the least fixpoint of $\varphi$, where $X$ is a formula variable bound inside $\varphi$; $\nu$ denotes the greatest fixpoint operator.

Henzinger et al. [71] show that the $\mathbf{T}\mu$ calculus can express the dense-real-time version of Computation Tree Logic (CTL), Timed CTL (TCTL) [4]. TCTL generalizes CTL by allowing atomic propositions to be any DL formula, and in addition contains formulas of the form $z.\varphi$ where $z$ is a specification clock variable and $\varphi$ is a TCTL formula in which $z$ appears free; the latter class enables one to write time-bounded properties. We omit the details for brevity.

Several model checkers are specialized to check reachability properties. Using the notation of the $\mathbf{T}\mu$ calculus, a reachability property is a formula of the form

$$\phi_{init} \implies \neg\mu X.[\phi_{err} \vee (\mathbf{true} \triangleright X)]$$

where $\phi_{init}$ is the initial set of states, and $\phi_{err}$ characterizes the bad states; the formula evaluates to $\mathbf{true}$ if no error state is reachable from any initial state.

## 9.3 Fully Symbolic Model Checking

Our model checking algorithm can be viewed as an implementation of one given by Henzinger et al. [71], where we perform operations in QDL using Boolean methods. This algorithm checks that a timed automaton $\mathcal{T}$ satisfies a specification given as a $\mathbf{T}\mu$ formula $\varphi$. The algorithm always terminates, and generates a DL formula $|\varphi|$, such that, if $\mathcal{T}$ is *non-zeno* (i.e., time can diverge from any state), then $|\varphi|$ is equivalent to $\mathcal{I}_{\mathcal{T}}$.

The algorithm is fully symbolic since it avoids the need to enumerate locations by representing sets of values of both Boolean state variables and clock variables as DL formulas. It performs backward exploration of the state space and uses the following three special operators over DL formulas:

1. **Time Elapse:** $\phi_1 \rightsquigarrow \phi_2$ denotes the set of all states that can reach the state set $\phi_2$ by allowing

time to elapse, while staying in state set $\phi_1$ at all times in between. Formally,

$$\phi_1 \rightsquigarrow \phi_2 \doteq \exists\delta\{\delta \geq x_0 \wedge \phi_2 + \delta \wedge \forall\epsilon[x_0 \leq \epsilon \leq \delta \implies \phi_1 + \epsilon]\} \tag{9.2}$$

where $\phi + \delta$ denotes the formula obtained by adding $\delta$ to all clock variables occurring in $\phi$, computed as $\phi[x_i + \delta / x_i, 1 \leq i \leq n]$, where $x_1, x_2, \ldots, x_n$ are the clock variables in $\phi_i$ (i.e., not including the zero variable $x_0$).

2. **Assignment:** $\phi[\mathcal{A}]$, where $\mathcal{A}$ is a set of assignments, denotes the formula obtained by simultaneously substituting in $\phi$ the right hand side of each assignment in $\mathcal{A}$ for the left hand side. Formally, if $\mathcal{A}$ is the list $e_1 := \phi_1, \ldots, e_k := \phi_k, x_1 := x_{j_1} + c_1, \ldots, x_n := x_{j_n} + c_n$, where each $e_i$ is a Boolean variable, each $x_j$ is a clock variable, and for each $x_{j_l}$, $j_l = 0$ or $c_l = 0$, then
$$\phi[\mathcal{A}] = \phi[\phi_1/e_1, \ldots, \phi_k/e_k, x_{j_1} + c_1/x_1, \ldots, x_{j_n} + c_n/x_n]$$
Assignments are thus performed via substitutions of Boolean and real-valued variables by expressions of the corresponding type. We use the techniques described in Section 8.3 to perform these substitutions.

3. **Weakest Pre-condition:** $pre_{\mathcal{T}}\phi$ denotes the weakest precondition of $\phi$ with respect to the timed automaton $\mathcal{T}$. Formally,
$$pre_{\mathcal{T}}\phi = \mathcal{I}_{\mathcal{T}} \wedge (\phi \vee \bigvee_{t \in \mathcal{E}} pre_t(\mathcal{I}_{\mathcal{T}} \wedge \phi))$$
where for a transition $t = \psi \implies \mathcal{A}$
$$pre_t(\phi) = \psi \wedge \phi[\mathcal{A}]$$
Note that $pre_{\mathcal{T}}$ is defined using assignments and Boolean operations.

The model checking algorithm is defined inductively on the structure of $\mathbf{T}\mu$ formulas, as shown below:

- $|\phi| := \mathcal{I}_{\mathcal{T}} \wedge \phi$
- $|\neg\varphi| := \mathcal{I}_{\mathcal{T}} \wedge \neg|\varphi|$
- $|\varphi_1 \vee \varphi_2| := |\varphi_1| \vee |\varphi_2|$
- $|\varphi_1 \triangleright \varphi_2| := |(|\varphi_1| \vee |\varphi_2|) \rightsquigarrow pre_{\mathcal{T}}(|\varphi_2|)|$
- $|z.\varphi| := |\varphi|[z := 0]$
- $|\mu X.\varphi|$ is the result of the following iteration:

$\phi_{new} := \mathbf{false};$
  $\mathbf{repeat}$
    $\phi_{old} := \phi_{new};$
    $\phi_{new} := |\varphi[X := \phi_{old}]|;$
  $\mathbf{until}\ (\phi_{new} \implies \phi_{old});$
  $\mathbf{return}\ \phi_{old};$

As can be seen from the algorithm description above, apart from Boolean operators, the main components of the algorithm are: quantifier elimination in the time elapse operation, substitution of state variables in an assignment, and the decision procedure used to check containment in fixpoint computation. For a fully symbolic model checker that represents state sets as DL formulas, these model checking operators can be defined as operations in QDL. We elaborate below.

**Time Elapse**

Consider the formula on the right hand side of Equation 9.2, the definition of the time elapse operator. This formula is not in QDL, since it includes expressions that are the sum of two real variables (e.g., $x + \delta$). However, it can be transformed to a QDL formula, by using, instead of $\delta$ and $\epsilon$, variables $\overline{\delta}$ and $\overline{\epsilon}$ that represent their negations:

$$\exists\overline{\delta}\{\overline{\delta} \leq x_0 \wedge \phi_2 + (-\overline{\delta}) \wedge \forall\overline{\epsilon}[\overline{\delta} \leq \overline{\epsilon} \leq x_0 \implies \phi_1 + (-\overline{\epsilon})]\} \tag{9.3}$$

Formula 9.3 is expressible in QDL, since the substitution $\phi[x_i + (-\overline{\delta})/x_i, 1 \leq i \leq n]$ can be computed as $\phi[\overline{\delta}/x_0]$.[3] This yields,

$$\exists\overline{\delta}\{\overline{\delta} \leq x_0 \wedge \phi_2[\overline{\delta}/x_0] \wedge \forall\overline{\epsilon}(\overline{\delta} \leq \overline{\epsilon} \leq x_0 \implies \phi_1[\overline{\epsilon}/x_0])\} \tag{9.4}$$

Finally, we can rewrite Formula 9.4 purely in terms of existential quantifiers:

$$\exists\overline{\delta}\{\overline{\delta} \leq x_0 \wedge \phi_2[\overline{\delta}/x_0] \wedge \neg\exists\overline{\epsilon}(\overline{\epsilon} \leq x_0 \wedge \overline{\delta} \leq \overline{\epsilon} \wedge \neg\phi_1[\overline{\epsilon}/x_0])\} \tag{9.5}$$

A procedure for performing the time elapse operation therefore requires one for eliminating (existential) quantifiers over real variables from a DL formula. For this purpose, we use the quantifier transformation technique described in Section 8.1.

In addition, we can exploit the special structure of Formula 9.5 so as to avoid introducing $\overline{\epsilon}$ altogether. Thus, we can avoid adding new quantified Boolean variables encoding predicates involving $\overline{\epsilon}$.

[3]Note that substituting $x_0$ by $\overline{\delta}$ or $\overline{\epsilon}$ can be viewed as shifting the zero reference point to a more negative value, thus increasing the value of any clock variable relative to zero (e.g., [9, 102]).

Consider the inner existentially quantified DL formula in Formula 9.5, reproduced here:

$$\exists\overline{\epsilon}(\overline{\epsilon} \leq x_0 \wedge \overline{\delta} \leq \overline{\epsilon} \wedge \neg\phi_1[\overline{\epsilon}/x_0])$$

Grouping the inequality $\overline{\delta} \leq \overline{\epsilon}$ with the formula $\neg\phi_1[\overline{\epsilon}/x_0]$, we get:

$$\exists\overline{\epsilon}\{\overline{\epsilon} \leq x_0 \wedge (\overline{\delta} \leq x_0 \wedge \neg\phi_1)[\overline{\epsilon}/x_0]\} \tag{9.6}$$

Finally, treating $\overline{\delta}$ as a clock variable, we can revert back to $\epsilon$ from $\overline{\epsilon}$, transforming Formula 9.6 to the following form:

$$\exists\epsilon[\epsilon \geq x_0 \wedge (\overline{\delta} \leq x_0 \wedge \neg\phi_1) + \epsilon] \tag{9.7}$$

Formula 9.7 is a special case of the formula $\omega_\epsilon$ given in Equation 8.13. Therefore, we can employ the optimization described in Section 8.4.2.

### Checking Containment

Containment of one set of states, $\phi_{new}$, in another, $\phi_{old}$, is checked by deciding the validity of the DL formula $\phi \doteq \phi_{new} \implies \phi_{old}$ (or equivalently, the satisfiability of $\neg\phi$). The satisfiability of $\neg\phi$ is decided using the technique of Section 8.2.

### Reachability Analysis

A simple but very useful special case of model checking is to compute the set of reachable states of the timed automaton. This can be used for checking safety properties.

Let $\phi_0$ denote a DL formula characterizing the initial set of states of a timed automaton $\mathcal{T}$. The following three-step algorithm computes a DL formula $\phi_{\text{reach}}$ representing the set of reachable states of $\mathcal{T}$.

1. $\phi_{new} := \phi_0$.
2. **Do**
   (a) $\phi_{old} := \phi_{new}$
   (b) $\phi' := \textbf{post}_{\textbf{time}}(\phi_{old})$ {Let time elapse}
   (c) $\phi'' := \textbf{post}_{\mathcal{P}}(\phi')$ {Fire a transition}
   (d) $\phi_{new} := \phi_{old} \vee \phi''$ {Union of sets}

   **While** $(\phi_{old} \neq \phi_{new})$ {Check termination}

3. $\phi_{\text{reach}} := \phi_{new}$.

The symbolic "next-state" operators $\mathbf{post}_{\mathbf{time}}$ and $\mathbf{post}_{\mathcal{P}}$ are defined as follows:

$$\mathbf{post}_{\mathbf{time}}(\phi) \doteq \exists\delta\{\delta \geq 0 \wedge \phi - \delta \wedge \forall\epsilon[0 \leq \epsilon \leq \delta \implies \mathcal{I}_{\mathcal{T}} - \epsilon]\} \tag{9.8}$$

where $\phi - \delta$ denotes the formula obtained by subtracting $\delta$ from all clock variables occurring in $\phi$, computed as $\phi[x_i - \delta/x_i, 1 \leq i \leq n]$, where $x_1, x_2, \ldots, x_n$ are the clock variables in $\phi_i$ (and similarly for $\mathcal{I}_{\mathcal{T}} - \epsilon$).

Intuitively, $\delta$ is the time elapsed since the last transition fired. The inner quantified formula in Equation 9.8 ensures that while allowing time to elapse, the values of clock variables must always respect the invariant $\mathcal{I}_{\mathcal{T}}$. The formula obtained after eliminating quantifiers from $\mathbf{post}_{\mathbf{time}}(\phi)$ represents all states reachable from $\phi$ by allowing some duration of time to elapse within the constraints imposed by $\mathcal{I}_{\mathcal{T}}$.

The operation $\mathbf{post}_{\mathcal{P}}$, when applied to a set of states $\phi$, returns the set of states reached from $\phi$ by making some transition. Formally,

$$\mathbf{post}_{\mathcal{P}}(\phi) \doteq \bigvee_{(\psi \implies \mathcal{A}) \in \mathcal{E}} (\phi \wedge \psi)[\mathcal{A}] \tag{9.9}$$

### 9.3.1 Implementation and Results

We implemented a model checker called TMV that uses BDDs to represent Boolean functions and incorporates all the optimizations described in Section 8.4. The model checker is written in the O'Caml language and uses the CUDD package [47] for BDD manipulation.

We have performed experiments comparing the performance of our model checker for both reachability and non-reachability $\mathbf{T}\mu$ properties. For reachability properties, we compare against the other unbounded, fully symbolic model checkers, viz., a DDD-based checker (DDD) [102] and RED version 4.1 [161], which have been shown to outperform UPPAAL2K and KRONOS for reachability analysis. For non-reachability properties, such as checking that a system is non-zeno, we compare against KRONOS and RED, the only other unbounded model checkers that check such properties.

As an illustrative example, we use Fischer's protocol for mutual exclusion. Tools such as DDD and RED that we compare against have been shown to perform well on this example for reachability properties. The automaton for the $i$th process in this protocol is shown in Figure 9.2. We ran two experiments with this example. The first experiment compared our model checker against DDD and RED, checking that the system preserves mutual exclusion (a reachability property). In the second experiment, we compared against KRONOS and RED for checking that the product automaton is non-zeno (a non-reachability property). All experiments were run on a notebook computer with a 1

GHz Pentium-III processor and 128 MB RAM, running Linux. We ran DDD, KRONOS, and RED with their default options. For our implementation, we turned off dynamic variable reordering in CUDD. To come up with a static variable ordering, we classified the BDD variables in our Boolean encoding as follows. The first class, $C_{\texttt{id}}$, consists of variables encoding the shared integer `id`. For each $i$, class $C(i)$ contains the BDD variables encoding locations and clock constraints for process $i$. Finally, class $C(i, j)$ encodes predicates relating clock variables from processes $i$ and $j$. We used a static variable ordering that groups together variables in the same class, places class $C_{\texttt{id}}$ at the top, orders $C(i)$ before $C(j)$ if $i < j$, and places $C(i, j)$ right after $C(j)$ for $j > i$. New BDD variables added during model checking are inserted into the order at positions that depend upon the class they fall into. The same static variable order was used for the corresponding Boolean variables and difference constraints in DDD.

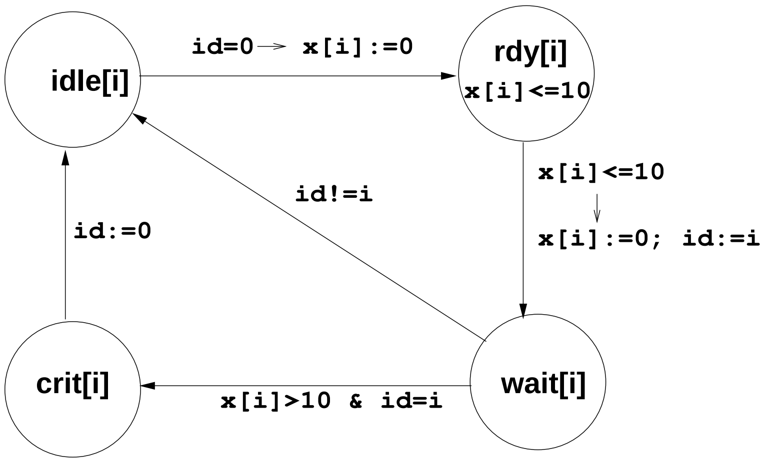



Figure 9.2: **Fischer's mutual exclusion protocol.** The timed automaton for the $i$th process is shown. Edges are labeled with guards and assignments, omitting either where unnecessary.

Table 9.1 shows the results of the comparison against DDD and RED for checking mutual exclusion for increasing numbers of processes. For DDD and TMV, the table lists both the run-times and the peak number of nodes in the decision diagram for the reachable state set. We find that DDD outperforms TMV due to the blow-up of BDDs. In spite of the optimizations of Section 8.4, the peak node count in the case of DDD is less than that for TMV for the larger benchmarks. In particular, in addition to eliminating infeasible paths as TMV does, the local reduction operations performed by DDD during node creation can eliminate unnecessary DDD nodes without adding any time overhead. For example, DDD can reduce a function of the form $e_1 \wedge e_2 \wedge e_3$ under the transitivity constraint $[e_1 \wedge e_2] \implies e_3$ to simply the conjunction $e_1 \wedge e_2$. The BDD `Restrict` operator cannot always achieve this as it is sensitive to the BDD variable ordering. Furthermore, TMV contains many other BDDs, such as those for the transitivity constraints, to which we do not apply the `Restrict` optimization due to its runtime overhead. Finally, in comparison to RED, we see that while TMV is faster on the smaller benchmarks, RED's superior memory performance enables it to complete for 7 processes while TMV runs out of memory.

| Number of Processes | RED Time (sec.) | DDD Time (sec.) | DDD Reach Set (peak nodes) | TMV Time (sec.) | TMV Reach Set (peak nodes) |
|---|---|---|---|---|---|
| 3 | 0.21 | 0.06 | 130 | 0.11 | 101 |
| 4 | 1.13 | 0.14 | 352 | 0.38 | 316 |
| 5 | 4.53 | 0.33 | 854 | 1.85 | 1127 |
| 6 | 15.11 | 0.90 | 2375 | 17.41 | 4685 |
| 7 | 46.31 | 2.65 | 6346 | * | * |

Table 9.1: **Checking mutual exclusion for Fischer's protocol.** A "*" indicates that the model checker ran out of memory.

Table 9.2 shows the comparison with KRONOS and RED for checking non-zenoness. The time for KRONOS is the sum of the times for product construction and backward model checking. We notice that while KRONOS does better for smaller numbers of processes, the product automaton it constructs grows very quickly, becoming too large to construct at 6 processes. The run times for TMV, on the other hand, grow much more gradually, demonstrating the advantages of a fully symbolic approach. For this property, the BDDs remain small even for larger numbers of processes. Thus, TMV outperforms RED, especially as the number of processes increases. These results indicate that when the representation (BDDs) remains small, Boolean methods for quantifier elimination and deciding DL can outperform non-Boolean methods by a significant factor.

| Number of Processes | KRONOS Time (sec.) | RED Time (sec.) | TMV Time (sec.) | TMV Reach Set (peak nodes) |
|---|---|---|---|---|
| 3 | 0.03 | 0.28 | 0.24 | 28 |
| 4 | 0.23 | 1.30 | 0.44 | 39 |
| 5 | 1.98 | 5.05 | 0.80 | 54 |
| 6 | * | 17.80 | 2.15 | 69 |
| 7 | * | 57.95 | 6.61 | 88 |

Table 9.2: **Checking non-zenoness for Fischer's protocol.** A "*" indicates that KRONOS exited with an "out of memory" error.

**Discussion**

The results in this section, although limited, indicate that our model checker based on a general purpose BDD package can outperform methods based on specialized representations of DL formulas. The drawback of our BDD-based implementation is its poor memory performance on some examples. However, there is scope for improving our implementation, especially in finding more efficient ways of eliminating unnecessary BDD nodes as is possible with DDDs. Furthermore, note that the memory problems we face arise from our use of BDDs, while the techniques proposed in this thesis can make use of *any* representation of Boolean functions. In particular, a SAT-based implementation of our method might better handle the growth in the number of Boolean variables.

While Fischer's protocol is an interesting toy example, the real test of our model checker is how it performs on practical problems. In the next section, we describe an application of our model checker to the verification of timed circuits.

## 9.4 Verification of Timed Circuits

Timing assumptions are commonly used in the design of both asynchronous and synchronous circuits in order to improve performance. Examples include the GasP circuits [150], the Global STP circuit in the Intel Pentium 4 processor [72], and the RAPPID instruction decoder [143]. However, the use of timing assumptions comes at an added verification cost: The circuit behavior must be verified under these constraints, and furthermore, the constraints must themselves be verified pre- and post-layout.

A promising recent approach to this verification problem is to use a design methodology based on *relative timing* [145]. In the relative timing (RT) paradigm, timing assumptions are made *explicit*, by adding constraints on the relative ordering of signal transitions to an otherwise untimed design. In contrast, other methods use *implicit* timing assumptions, where the timing assumptions are either implicit in a design style (such as Burst-Mode techniques, e.g. [115]) or imposed at the gate-level in the circuit model (such as metric timed circuit design [105]). Using the RT paradigm, verification proceeds in two steps:

1. *Checking correctness under timing constraints:* RT constraints are identified and the correct operation of the circuit is verified under those constraints. Typically, one either checks that the implemented circuit $\mathcal{I}$ only exhibits behaviors of a specification $\mathcal{S}$, or that it satisfies a specific property $\varphi$ formulated in a suitable temporal logic.

2. *Verifying that the circuit obeys timing constraints:* The identified RT constraints are themselves verified using standard simulation or static timing analysis techniques. The constraints

can be verified pre-layout to ensure that they have sufficient margin based on expected design parameters. The constraints also must be validated post-layout with extracted data to ensure that place and route, sizing, and buffer insertion have not skewed the delays beyond acceptable values.

The RT approach of explicitly stating timing constraints has the advantage that it applies to many asynchronous design styles [145]. It supports a design philosophy of adding timing constraints incrementally and of giving the designer flexibility in using timing constraints. Also, unlike gate-level metric timing, it does not rely on conservatively set min-max bounds on gate delays.

However, current RT-based verification techniques (e.g., [85, 121]) fall short in three respects. First, not all timing constraints can be expressed as the relative ordering of signal transitions. Secondly, current verification tools are yet to scale up to relatively large circuits and achieve the success obtained by symbolic methods for untimed systems (e.g., [33]). Finally, previous work on relative timing-based verification [85, 121] does not satisfactorily address the problem of verifying that the circuit obeys the constraints.

In this section, we address these shortcomings by making the following novel contributions:

- *A generalized notion of relative timing:* We introduce the concept of a *generalized relative timing* (GRT) constraint, one that specifies a relative ordering not just between events, but between the time intervals between pairs of events. This generalization adds the capability to model some metric timing information which is formally modeled using real-valued clock variables. The resulting circuit model is a timed automaton. However, since metric timing constraints are typically far fewer than non-metric GRT constraints, we employ relatively few clock variables.

- *Application of fully symbolic verification methods:* We use the new fully symbolic model checking algorithm introduced earlier in this chapter. Along with the modeling methodology described above, this enables us to verify circuits that are significantly larger than those verifiable with other methods. As an example we have efficiently analyzed the Global STP circuit [72], finding an error in the published circuit, and then successfully verifying a fixed version.

This section is organized as follows. We introduce the idea of generalized relative timing in Section 9.4.2. In Section 9.4.3, we describe how timed circuits are formalized as timed automata. Case studies are presented in Section 9.4.4.

### 9.4.1 Previous Work

Several techniques have been proposed in the past 15 years to model timing constraints in circuit design. A common approach is to specify upper and lower bounds on the delay between when a transition is enabled and when it fires. Formalisms such as timed transition systems [70], timed Petri nets [128] and timed event and event/level structures [16, 101, 105] are used for this purpose, and the constraints are referred to as *gate-level metric timing* constraints. This is an intuitive model, but since the timing information is provided at the gate-level, verification tools based on this model are restricted to relatively small circuits. Even with the use of partial order reduction methods (e.g., [16, 101]), the size of the untimed state space still presents a performance bottleneck. The min-max delay bounds can impose unnecessary timing constraints on unrelated parts of the circuit. Furthermore, designers must be relatively conservative on how they set the bounds, since these can depend on post-layout information.

Another formalism for modeling timed systems is that of timed automata [5], which is more expressive than timed transition systems [6], in that it can model "more global" timing constraints. Maler and Pnueli [94] model asynchronous circuits using timed automata, but their model is also at the gate-level, requiring one clock variable per gate. Thus, it suffers from the same scaling problems as the afore-mentioned metric timing methods. Our work also uses timed automata as the modeling formalism, but in an entirely different way: We model timing constraints at a higher level of abstraction, and introduce clock variables only where necessary.

The observation that enables us to selectively use clock variables is that most timing constraints are on pairs of events that have a common start event, i.e., a "point-of-divergence." A similar observation was made by Negulescu and Peeters [107, 108], who present the notion of a *chain constraint*, which specifies that one sequence of transitions must occur before another with both sequences sharing a common prefix. A "point-of-divergence" constraint is more restrictive than a chain constraint in a logical sense (it specifies a relative ordering for *all* intermediate sequences of transitions between the start and end events), but for the same reason, it is more compact to specify. Moreover, we can model more general kinds of constraints, as we describe in Section 9.4.2.

There has been prior work on RT-based verification, with a focus on automatically generating constraints. Peña *et al.* [121] present an approach based on the notion of *lazy transition systems*. Their approach automatically and iteratively generates RT constraints to rule out spurious counterexamples; however, the process of adding RT constraints relies on knowing min-max bounds on gate delays. Kim *et al.* [85] present a verification methodology based on a different technique of automatically generating RT constraints, but do not address the problem of verifying that the circuit obeys the constraints. While we do not automatically generate timing constraints, our work targets a more general class of timing constraints, and provides ways of verifying that the constraints hold for the circuit.

Clarisó and Cortadella [40] present a gate-level modeling approach that represents gate delays by symbols, rather than by constant bounds. Thus, this model is more expressive than metric timing. However, the verification problem is even harder than for timed transition systems, and the approach is restricted to very small circuits.

In the context of asynchronous circuits, there has been much work on algorithms for model checking timed systems; see, for example, the work by Myers, Yoneda, *et al.*(e.g., [16, 101, 105, 167]). The main difference with our work is that these methods are symbolic in the real-valued part, but explicit-state in the Boolean part; hence, in spite of incorporating partial-order reduction, large circuits are often outside their capacity.

There has also been work on methods that use compositional reasoning or abstraction to achieve better scalability (e.g., [170]). Our focus, in this thesis, is on demonstrating scalability without using compositional reasoning or abstraction; however, nothing precludes using the techniques presented herein along with such methods.

### 9.4.2 Modeling Timed Circuits

A timed circuit is a triple $(\mathcal{V}, \mathcal{R}, \mathcal{T})$, where $\mathcal{V}$ is a set $\{v_1, v_2, \ldots, v_n\}$ of circuit *signals*, $\mathcal{R}$ is a set $\{r_1, r_2, \ldots, r_m\}$ of *rules*, and $\mathcal{T}$ is a set $\{\tau_1, \tau_2, \ldots, \tau_p\}$ of *timing constraints*. The set of initial values of signals in $\mathcal{V}$ is specified as a Boolean formula $I_{\mathcal{V}}$.

The circuit signals, which are the *state variables* of the system, are comprised of inputs, outputs, and intermediate signals. A *transition* (also referred to as *event*) is a change in logic level of a signal. Transition $v_i\!\uparrow$ corresponds to the transition of $v_i$ from 0 to 1, and $v_i\!\downarrow$ to the transition from 1 to 0. We will use the symbol $\theta_i$ to refer to either transition for signal $v_i$.

The untimed circuit behavior is defined by the set of rules $\mathcal{R}$, which comprises $m = 2n$ rules, one for each signal transition.[4] The 2 rules for the $i$th signal $v_i$ are written as

$$\mathcal{E}_{v_i\uparrow} \mapsto v_i\!\uparrow \quad \text{and} \quad \mathcal{E}_{v_i\downarrow} \mapsto v_i\!\downarrow$$

where $\mathcal{E}_{\theta_i}$ is a Boolean formula over $\mathcal{V}$ indicating the enabling condition for transition $\theta_i$ to fire.

Although we have only introduced two events per signal (corresponding to up and down transitions), it would be straightforward to add finitely-many instances of each event. That is, for a given event $\theta_i$, we can keep track of not only each instance of $\theta_i$, but also every second, third, ..., $k^{\text{th}}$ instances of $\theta_i$ for a constant $k$, with the use of additional state bits to keep track of a "count." However, we have rarely needed to track more than one instance of each event.

We will assume an *inertial* gate model (but without bounds on gate delays). Thus, it is allowed for a transition that was enabled to become disabled without having fired, as long as the circuit satisfies

[4] Notice that this is similar to the language of production rules [96].

its specification. In the absence of an explicit timing constraint involving transition $\theta_i$, the time taken for $\theta_i$ to fire after being enabled can be any value in $[0, \infty)$; i.e., rules, by themselves, are purely untimed.

### Generalized Relative Timing

The novel aspect of how we model circuits is in the formulation of *generalized relative timing* constraints, which combine relative timing with a capability to incorporate some metric timing information.

Let $\Delta(\theta_i, \theta_j)$ denote the time interval between an occurrence of $\theta_j$ and *the* occurrence of $\theta_i$ immediately preceding it.

The following definition formalizes the notion of generalized relative timing (GRT):

**Definition 9.1** *Let $\theta_i, \theta_i', \theta_j, \theta_k$ be four transitions such that $\theta_j \neq \theta_k$. Then, a generalized relative timing constraint on $\theta_i, \theta_i', \theta_j, \theta_k$ is of the form:*
For all occurrences of transitions $\theta_j$ and $\theta_k$,

$$\Delta(\theta_i, \theta_j) < \Delta(\theta_i', \theta_k) + d$$

*where $d$ is a rational constant.*

It is sometimes useful to use a non-strict inequality ($\leq$) instead of the strict inequality used above, or to drop one of the $\Delta(\cdot, \cdot)$ terms in the inequality so as to impose an upper or lower bound on the time interval between events.

**Point-of-divergence constraint.** An extremely common sub-class of GRT constraints are those such that $\theta_i = \theta_i'$, $d = 0$, and the *same* occurrence of $\theta_i$ immediately precedes all occurrences of both $\theta_j$ and $\theta_k$. In this case, the timing constraint specifies that measuring time from the point $\theta_i$ occurs, $\theta_j$ must always occur before $\theta_k$. We will refer to this special case as a *point-of-divergence* (POD) constraint. (The name comes from the divergence in two paths starting from transition $\theta_i$.) We write a POD constraint as $\theta_i \rightarrow \theta_j \prec \theta_k$.

Typically, $\theta_j$ and $\theta_k$ causally depend on $\theta_i$. However, note that this need not be the case! By the definition of $\Delta(\theta_i, \theta_j)$, the point-of-divergence in the constraint is simply the occurrence of $\theta_i$ that is *closest in time* to $\theta_j$ and $\theta_k$, which need not have caused either of them.

Note also that the concept of a POD constraints is essentially the same as that of the original RT constraint, since, in order to implement a relative ordering between events, one would have to trace them back to a point-of-divergence; hence the name *generalized* relative timing.

**Metric timing constraints.** The presence of $d$ in the definition allows us to express a limited form of metric timing constraints. In particular, we can express constraints of the form $d_1 \leq \Delta(\theta_i, \theta_j) \leq d_2$.

Note, however, that we cannot directly specify the min-max timing assumptions used in timed transition systems [70] and related formalisms, since that would require constraining the delay between when a transition is *enabled* and when it fires.[5]

**Compound timing constraints.** In some cases, such as the Global STP circuit that is our primary case study, we have observed the need for *compound timing constraints* formed as an XOR of two (simple) timing constraints. Such a constraint is written as $\tau_i \text{ XOR } \tau_j$. We have needed such compound constraints to reason about relative ordering between instances of events from different cycles of circuit operation. Further discussion of such constraints is deferred to the case study in Section 9.4.4.

In all our case studies to date, we have found the class of generalized relative timing constraints to be sufficient. In fact, most constraints tend to be simple (i.e., not compound) POD constraints. Metric timing constraints are used only when there is explicit use of delay values in the design.

We present two examples to illustrate our methodology for modeling timing constraints.

**Example 9.2** Consider the implementation of a C-element using three AND gates and an OR gate, as shown in Figure 9.3.

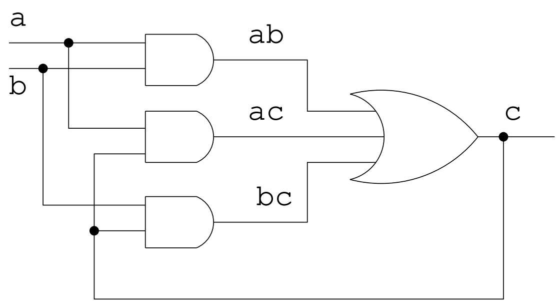



Figure 9.3: **Implementation of a C-element**

`a` and `b` denote the input signals, and `c` is the output. It is easy to see that in order to work correctly, it is sufficient for the circuit in Figure 9.3 to respect the following two fundamental mode constraints, formulated here as POD constraints: $\mathtt{c}\uparrow \rightarrow \mathtt{ac}\uparrow \prec \mathtt{b}\downarrow$ and $\mathtt{c}\uparrow \rightarrow \mathtt{bc}\uparrow \prec \mathtt{a}\downarrow$. $\square$

While POD constraints suffice for the preceding example, in general, we might need a more expressive timing constraint. The following example demonstrates the need for increased expressiveness.

**Example 9.3** Figure 9.4 depicts a simple buffer stage element generated from the CASH compiler that compiles ANSI-C programs into asynchronous circuits [159]. For correct operation, this circuit relies on two timing assumptions: data transfers between stages use a bundled data protocol, and a stage incorporates a matched delay element.

[5]However, note that the formalism that we use, viz. timed automata, is general enough to express such constraints [6].

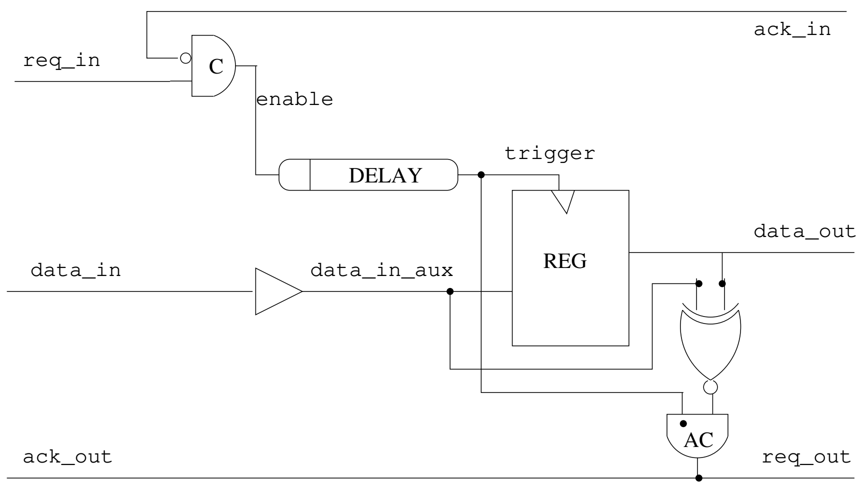



Figure 9.4: **Buffer stage from CASH compiler**

The matched delay can be formalized with the following two timing assumptions $\tau_1^{\text{CASH}}$ and $\tau_2^{\text{CASH}}$:

$$\Delta(\texttt{data\_in}\uparrow, \texttt{data\_in\_aux}\uparrow) < \Delta(\texttt{enable}\uparrow, \texttt{trigger}\uparrow) \qquad (\tau_1^{\text{CASH}})$$

$$\Delta(\texttt{data\_in}\downarrow, \texttt{data\_in\_aux}\downarrow) < \Delta(\texttt{enable}\uparrow, \texttt{trigger}\uparrow) \qquad (\tau_2^{\text{CASH}})$$

To ensure that the stage respects the bundled data protocol, we additionally need to impose two POD constraints: $\texttt{enable}\uparrow \to \texttt{data\_out}\uparrow \prec \texttt{req\_out}\uparrow$, and $\texttt{enable}\uparrow \to \texttt{data\_out}\downarrow \prec \texttt{req\_out}\uparrow$. $\square$

Note that the matched delay assumptions $\tau_1^{\text{CASH}}$ and $\tau_2^{\text{CASH}}$ in Example 9.3 can be reformulated as POD constraints by tracing back to the `enable` signal of the previous stage. However, this breaks modularity, since the timing constraints involving signals of a module reference internal signals of another module. In general, we have found that while it is often possible to reformulate metric timing constraints as POD constraints, it is at the cost of modularity.

### Verifying Timing Constraints

The verification methods presented in this chapter prove that the timed circuit design is correct given the set of timing constraints $\mathcal{T}$. However, it does not prove that the constraints actually hold given the true delays in the design. Timing constraints can be constructed that do not hold in a design, as will be shown later in Section 9.4.4. Therefore, these must be proved separately, in addition to verifying the logical functionality of the circuit. We briefly describe this process to show a consistent design flow exists for our verification method.

Given a POD constraint $\theta_i \to \theta_j \prec \theta_k$ we must prove that any sequence of events from $\theta_i$ to $\theta_j$ always occurs before the events from $\theta_i$ to $\theta_k$. This is accomplished by tracing and timing the maximum and minimum delay paths from the POD to the end points, and comparing the results. We compute the maximum delay of the left path ($\theta_i \rightsquigarrow \theta_j$) and the minimum delay for the right path ($\theta_i \rightsquigarrow \theta_k$). This ensures that no combination of delays will cause $\theta_k$ to occur before $\theta_j$. The same conditions exist for the general form of constraints $\Delta(\theta_i, \theta_j) \leq \Delta(\theta'_i, \theta_k) + d$ where the tracing may occur to different starting points, and a constant delay is added when the path delays are compared.

We illustrate static timing validation using the circuit in Figure 1. There are two POD constraints, the first of which is $\mathtt{c}\uparrow \rightarrow \mathtt{ac}\uparrow \prec \mathtt{b}\downarrow$. Validating this constraint requires evaluation of the max-delay path from $\mathtt{c}\uparrow$ to $\mathtt{ac}\uparrow$. This is simply the maximum rise delay through the gate corresponding to `ac` since signal `a` is already asserted. Similarly, the minimum delay path from $\mathtt{c}\uparrow$ to $\mathtt{b}\downarrow$, which depends on how the gate is connected to its environment, is calculated and compared with the maximum rising delay of the gate `ac` to validate this constraint. The second constraint $\mathtt{c}\uparrow \rightarrow \mathtt{bc}\uparrow \prec \mathtt{a}\downarrow$ is similarly validated.

The capability of automatically tracing and timing maximum and minimum delay paths, and comparing the results is supported in most commercial timing tools such as PrimeTime [152]. Therefore, it is possible to automatically validate all the constraints in $\mathcal{T}$. However, some complications arise in automatically tracing signals through sequential elements (such as the C-element of Figure 9.3), since static tools may not correctly cut feedbacks that exist solely to retain state. Fully automatic translation and validation of GRT constraints using static timing tools is left to future work.

The timing constraints used in this chapter were identified manually, many with the assistance of a relative-timing enhanced verification engine [144]. Automatic generation of GRT constraints is left to future work.

### 9.4.3 From Circuits to Timed Automata

We describe how we formally model timed circuits as timed automata. The *timed guarded commands* representation of timed automata is used, as it is more intuitive in the current context.

The translation of a timed circuit $(\mathcal{V}, \mathcal{R}, \mathcal{T})$ to a timed automaton $\mathcal{T}$ is performed in three steps.

**Initialization.** The set of Boolean state variables of $\mathcal{T}$ is initialized to be the set of signals $\mathcal{V}$, while the set of clock variables $\mathcal{X}$ is initialized to $\emptyset$.

Each rule of the timed circuit gets translated to a corresponding guarded command of the timed automaton; thus, there is exactly one guarded command for each transition $\theta$. For transition $\theta$ with corresponding rule $\mathcal{E}_\theta \mapsto \theta$, we initialize its guarded command to be $\mathcal{E}_\theta \implies \theta$.

The invariant $\mathcal{I}_\mathcal{T}$ is initialized to be $\mathbf{true}$, and $\phi_0$ is set to be $I_\mathcal{V}$ (the set of initial signal values).

**Adding auxiliary variables.** For each timing constraint, we add an additional Boolean or clock variable to store timing information.

Let $\tau_i$ be the $i^{\text{th}}$ timing constraint.

If $\tau_i$ is a POD constraint, we only introduce a fresh Boolean state variable $b_i$ into $\mathcal{V}$.

Suppose $\tau_i$ is not a POD constraint, and is of the form $\Delta(\theta_i, \theta_j) \leq \Delta(\theta'_i, \theta_k) + d$. Then we not only introduce a fresh Boolean state variable $b_i$ into $\mathcal{V}$, but also add two clock variables $x_{\theta_i,\theta_j}$ and $x_{\theta'_i,\theta_k}$ to $\mathcal{X}$.

**Encoding timing constraints.** We encode timing constraints in sequence, running through the set $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_p\}$. As we encounter timing constraints containing a transition $\theta$, we update the guarded command corresponding to it.

Suppose we are encoding timing constraint $\tau_t$, which mentions transition $\theta$. Let the current form of the guarded command $\gamma$ for $\theta$ be $\psi \implies \mathcal{A}$.

How we modify $\gamma$ depends on whether the timing constraint is a POD constraint or not, and on the role of $\theta$ in the constraint, as elaborated below:

- *POD constraint:* Suppose the constraint is of the form $\theta_i \rightarrow \theta_j \prec \theta_k$. There are three cases, with $\gamma$ being modified differently in each case:

  Case $\theta = \theta_i$: $\gamma := \psi \implies \mathcal{A}'$,
  where $\mathcal{A}' = \mathcal{A} \cup \{b_t\!\uparrow\}$.

  Case $\theta = \theta_j$: $\gamma := \psi \implies \mathcal{A}'$,
  where $\mathcal{A}' = \mathcal{A} \cup \{b_t\!\downarrow\}$.

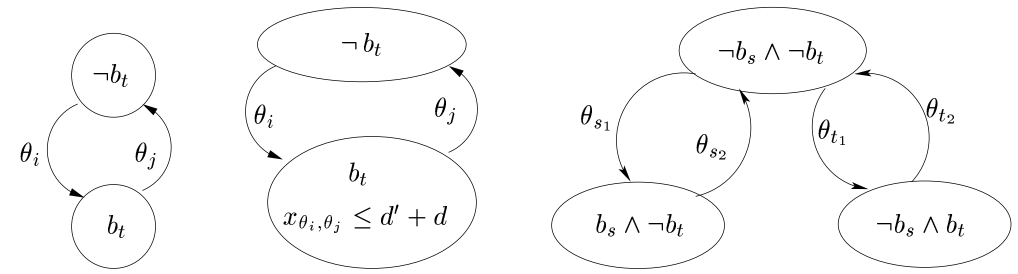
  Case $\theta = \theta_k$: $\gamma := \psi' \implies \mathcal{A}$,
  where $\psi' = \psi \wedge \neg b_t$.

  The intuition is that we take the product of the timed automaton (constructed so far) with a two-state monitor automaton as shown in Figure 9.5(a) to enforce the ordering specified by the POD constraint. The variable $b_t$ encodes the states of this automaton. Transition $\theta_k$ can only occur in the state labeled $\neg b_t$; i.e., the state in which $b_t$ is **false**.

- *Non-POD constraint:* Suppose the constraint is of the form $\Delta(\theta_i, \theta_j) \leq \Delta(\theta'_i, \theta_k) + d$. To encode this constraint, we introduce a non-negative constant $d'$ such that $\Delta(\theta_i, \theta_j) \leq d' + d$ and $d' \leq \Delta(\theta'_i, \theta_k)$. The value of $d'$ is usually known at design time since a non-POD constraint arises only in design styles that make use of some form of metric timing, such the matched delay assumption used in the circuit in Figure 9.4.

  We have four cases to consider:

  Case $\theta = \theta_i$: $\gamma := \psi \implies \mathcal{A}'$,
  where $\mathcal{A}' = \mathcal{A} \cup \{b_t\!\uparrow, x_{\theta_i,\theta_j} := 0\}$.




(a) POD (b) Non-POD (c) XOR of POD

Figure 9.5: **Monitor automata for timing**

Case $\theta = \theta_i'$: $\gamma := \psi \implies \mathcal{A}'$,
where $\mathcal{A}' = \mathcal{A} \cup \{x_{\theta_i',\theta_k} := 0\}$.

Case $\theta = \theta_j$: $\gamma := \psi \implies \mathcal{A}'$,
where $\mathcal{A}' = \mathcal{A} \cup \{b_t\!\downarrow\}$.

Case $\theta = \theta_k$: $\gamma := \psi' \implies \mathcal{A}$,
where $\psi' = \psi \wedge x_{\theta_i',\theta_k} \geq d'$.

In addition, we update the invariant $\mathcal{I}_{\mathcal{T}}$ of the timed automaton by conjoining the current invariant with the DL formula $b_t \implies x_{\theta_i,\theta_j} \leq d + d'$.

The intuition behind this translation is as follows. First, notice that the Boolean variable $b_t$ encodes, as before, the state of a monitor automaton, depicted in Figure 9.5(b). However, in this case, when $b_t$ is $\mathtt{true}$, $x_{\theta_i,\theta_j}$ cannot progress beyond $d + d'$, as enforced by the invariant $\mathcal{I}_{\mathcal{T}}$. Since the clock variable $x_{\theta_i,\theta_j}$ is reset when $\theta_i$ fires, this forces $\theta_j$ to occur within $d + d'$ time units of $\theta_i$. Secondly, clock variable $x_{\theta_i',\theta_k}$ is reset when $\theta_i'$ fires, and the augmented guard for $\theta_k$ ensures that $\theta_k$ can only fire $d'$ time units after $\theta_i'$. The above two mechanisms, in conjunction, ensure that the timing constraint $\tau_t$ is enforced.

The extension of the translation to handle compound timing constraints is straightforward; a XOR of two constraints can be encoded by making a non-deterministic choice to either monitor one constraint or the other. The monitor automaton for the compound constraint $\tau_s$ XOR $\tau_t$, where $\tau_s \doteq \theta_{s_1} \to \theta_{s_2} \prec \theta_{s_3}$ and $\tau_t \doteq \theta_{t_1} \to \theta_{t_2} \prec \theta_{t_3}$ is shown in Figure 9.5(c). We omit the details.

### Example

Consider the circuit in Figure 9.4. The rule corresponding to the transition $\mathtt{trigger}\!\uparrow$ is

$$\neg\mathtt{trigger} \wedge \mathtt{enable} \mapsto \mathtt{trigger}\!\uparrow$$

Timing constraints $\tau_1^{\text{CASH}}$ and $\tau_2^{\text{CASH}}$ both mention the transition `trigger`$\uparrow$.

Following the translation scheme described in this section, we introduce 3 clock variables $x_{\texttt{enable}\uparrow,\texttt{trigger}\uparrow}$, $x_{\texttt{data\_in}\uparrow,\texttt{data\_in\_aux}\uparrow}$, and $x_{\texttt{data\_in}\downarrow,\texttt{data\_in\_aux}\downarrow}$. The final guarded command for `trigger`$\uparrow$ is

$$\neg\texttt{trigger} \wedge \texttt{enable} \wedge (x_{\texttt{enable}\uparrow,\texttt{trigger}\uparrow} \geq d') \implies \texttt{trigger}\uparrow$$

where $d'$ is the delay corresponding to the delay element in the figure.

The invariant $\mathcal{I}_{\mathcal{T}}$ is the Boolean formula

$$(b_1 \implies x_{\texttt{data\_in}\uparrow,\texttt{data\_in\_aux}\uparrow} < d') \wedge (b_2 \implies x_{\texttt{data\_in}\downarrow,\texttt{data\_in\_aux}\downarrow} < d')$$

$b_1$ and $b_2$ are set by `data_in`$\uparrow$ and `data_in`$\downarrow$ respectively, and are reset by `data_in_aux`$\uparrow$ and `data_in_aux`$\downarrow$ respectively. Thus, our encoding simply formalizes the constraint that the delay through the buffer is less than that of the delay element.

### 9.4.4 Case Studies

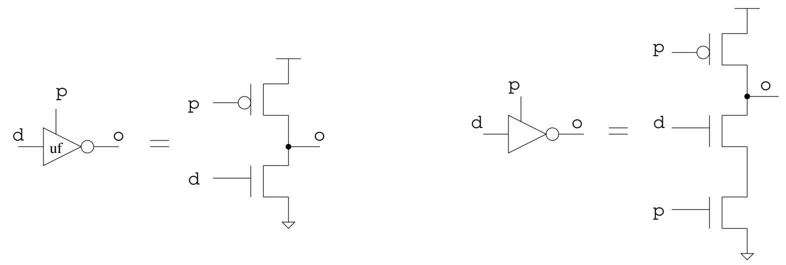
We have applied our model checker, TMV, to several case studies. The main industrial case study is a published version of the Global STP circuit, a self-timed circuit used in the integer unit in the Intel Pentium 4 processor [72]. Other case studies include the GasP FIFO control circuit [150], STAPL circuits [116], and the STARI circuit [64].

Experiments reported on here were run on a Linux workstation with a 2 GHz Pentium 4 processor and 1 GB of memory.

#### Global STP Circuit

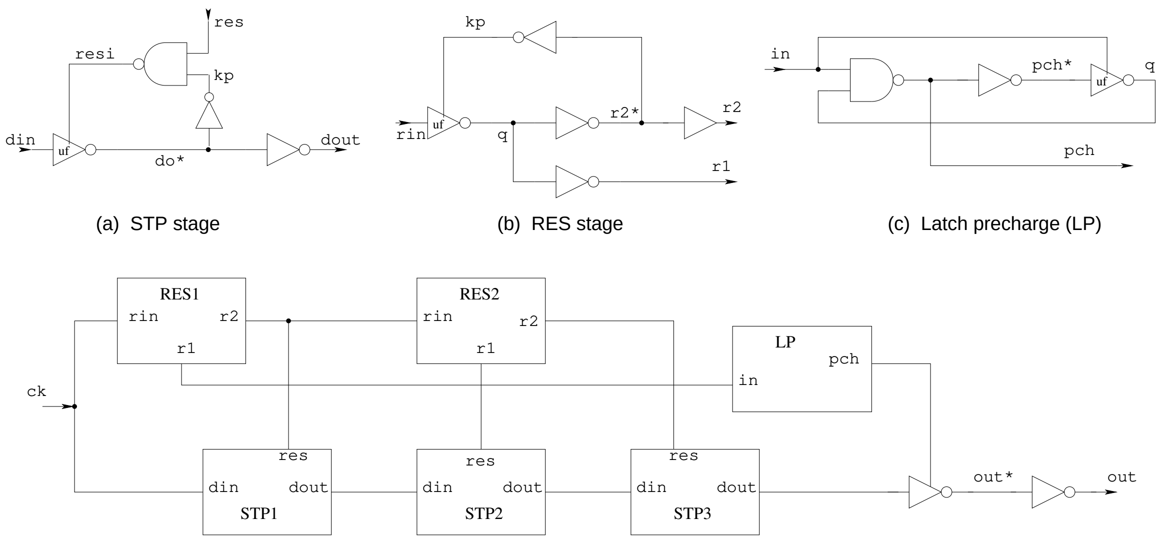
The Globally Reset Domino with Self-Terminating Precharge (Global STP) circuit [72] is a self-resetting domino circuit used in the integer unit of the Pentium 4 processor. The circuit uses both footed and unfooted domino inverters, shown in Figure 9.6. Figure 9.7 is a hierarchical, gate-level depiction of the Global STP circuit. The circuit we discuss here, shown in Figure 9.7, is the simplest form of the published circuit [72], with N-logic blocks replaced by wires, and static blocks replaced by inverters; our verification methods apply to the more general circuits as well.

The top-level circuit is shown in Figure 9.7(d), with the input `ck` being a 4-GHz clock and the output being a delayed version of the same clock. In the beginning of the clock cycle, the last footed domino gate is being reset, while the first three STP stages go through an evaluation. After the precharge of the last domino gate has been turned off, the evaluate signal propagates to the output, where it is held until the next cycle. Interestingly, note that the three STP stages are reset in the same cycle in which they evaluate.




(a) Unfooted (b) Footed

Figure 9.6: **Unfooted and footed domino inverters**




(a) STP stage (b) RES stage (c) Latch precharge (LP)

(d) Overall top−level circuit

Figure 9.7: **Global STP circuit**

This circuit relies on a number of timing constraints to ensure correct operation. We were able to formulate all these timing constraints either as POD constraints or as a XOR of two POD constraints. We discuss some of the more interesting timing constraints here.

Consider the $i^{\text{th}}$ STP stage, for all $i \in \{1, 2, 3\}$ (refer to Figure 9.7(a)). Short circuit current in the domino inverter must be avoided by ensuring that the pullup and pulldown transistors are not both conducting. This is avoided with the following POD constraint that does not allow the pullup to assert until after the pulldown has been turned off. This constraint states that for stage STP1, the delay of a clock phase must be shorter than the delay through the RES1 block:

$$\texttt{ck}\uparrow \;\rightarrow\; \texttt{STP}i\texttt{.din}\downarrow \;\prec\; \texttt{STP}i\texttt{.resi}\downarrow \qquad (\tau^{\text{GSTP}}_{1,i})$$

The pulse width of the outputs in the RES stage of Figure 9.7(b) are determined by the delay through the output buffers and the self-resetting loop. The following constrains the minimum pulse width on RES2.r2:

$$\texttt{RES2.r2}{*}\uparrow \;\rightarrow\; \texttt{RES2.r2}\uparrow \;\prec\; \texttt{RES2.r2}{*}\downarrow \qquad (\tau_2^{\mathrm{GSTP}})$$

Next, consider the footed domino inverter in Figure 9.7(d). The reset phase must terminate before the data is removed to guarantee the domino gate correctly latches data. Tracing the paths from the clock, we can express this in terms of the following ordering between two sequences of transitions:

$$\texttt{ck}\uparrow\texttt{RES1.r1}\uparrow\texttt{LP.pch}\downarrow\texttt{LP.pch}{*}\uparrow\texttt{LP.q}\downarrow\texttt{LP.pch}\uparrow$$
$$\prec$$
$$\texttt{ck}\uparrow\texttt{STP1.dout}\uparrow\texttt{STP2.dout}\uparrow\texttt{STP3.do}{*}\downarrow\texttt{STP3.kp}\uparrow\texttt{STP3.resi}\downarrow\texttt{STP3.do}{*}\uparrow\texttt{STP3.dout}\downarrow$$

This ordering is enforced with the following constraint:

$$\texttt{ck}\uparrow \;\rightarrow\; \texttt{LP.pch}\uparrow \;\prec\; \texttt{STP3.dout}\downarrow \qquad (\tau_3^{\mathrm{GSTP}})$$

To prevent incorrect overlap of the reset of the domino gate in each STP stage we need a constraint stating that $\texttt{STP}i\texttt{.res}\downarrow$ triggered by the previous rising edge of ck must occur before $\texttt{STP}i\texttt{.kp}\uparrow$ triggered by the current rising edge of ck. This is a multi-cycle constraint, which when written in terms of a sequence of transitions, is $\texttt{ck}\uparrow\texttt{STP}i\texttt{.res}\uparrow\texttt{STP}i\texttt{.res}\downarrow \prec \texttt{ck}\uparrow\texttt{ck}\downarrow\texttt{ck}\uparrow\texttt{STP}i\texttt{.din}\uparrow\texttt{STP}i\texttt{.do}{*}\downarrow\texttt{STP}i\texttt{.kp}\uparrow$. We can rephrase this multi-cycle constraint as a compound timing constraint $\tau_{4,i}^{\mathrm{GSTP}}$ XOR $\tau_{5,i}^{\mathrm{GSTP}}$, where $\tau_{4,i}^{\mathrm{GSTP}}$ and $\tau_{5,i}^{\mathrm{GSTP}}$ are two POD constraints given below:

$$\texttt{ck}\uparrow \;\rightarrow\; \texttt{STP}i\texttt{.res}\downarrow \;\prec\; \texttt{STP}i\texttt{.kp}\uparrow \qquad (\tau_{4,i}^{\mathrm{GSTP}})$$
$$\texttt{ck}\uparrow \;\rightarrow\; \texttt{STP}i\texttt{.res}\downarrow \;\prec\; \texttt{ck}\uparrow \qquad (\tau_{5,i}^{\mathrm{GSTP}})$$

To see why this is so, let us perform a case analysis. The first case is when the second instance of transition $\texttt{ck}\uparrow$ occurs before $\texttt{STP}i\texttt{.res}\downarrow$. In this case, the same instance of $\texttt{ck}\uparrow$ precedes both $\texttt{STP}i\texttt{.kp}\uparrow$ and $\texttt{STP}i\texttt{.res}\downarrow$, and hence we can simply write it as the POD constraint $\tau_{4,i}^{\mathrm{GSTP}}$. However, if the second instance of $\texttt{ck}\uparrow$ does not precede $\texttt{STP}i\texttt{.res}\downarrow$, it simply means that $\texttt{STP}i\texttt{.res}\downarrow$ occurs before $\texttt{ck}\uparrow$ fires again; i.e., $\tau_{5,i}^{\mathrm{GSTP}}$ holds, and so does the multi-cycle constraint.

Finally, consider the domino inverter in the LP stage, depicted in Figure 9.7(c). To avoid a short-circuit in this inverter, the following constraint is *necessary*:

$$\texttt{ck}\uparrow \;\rightarrow\; \texttt{LP.pch}{*}\downarrow \;\prec\; \texttt{RES1.r1}\downarrow \qquad (\tau_6^{\mathrm{GSTP}})$$

In all, we needed 33 timing constraints, as shown in Table 5.6 (we count a compound timing constraint as a single constraint). We model checked the circuit to verify the absence of short-circuits in all the domino inverters. The model checker's run-time was within a few minutes (see Table 5.6) and memory consumption was less than $150$ MB.

Next, we turned to verifying all the timing constraints, successfully verifying all but one: $\tau_6^{\mathrm{GSTP}}$. Consider this constraint. It takes only $5$ gate delays going from $\mathtt{ck}\uparrow$ to $\mathtt{RES1.r1}\downarrow$, while it takes $7$ going from $\mathtt{ck}\uparrow$ to $\mathtt{LP.pch}{*}\downarrow$. This means that the circuit, as described in the paper [72], has a short-circuit error. The main impact of this error appears to be increased power consumption and a greater propensity for device failure in the unfooted domino inverter.

To eliminate this error, we replaced the unfooted domino inverter in the LP stage by a footed domino inverter. With this replacement, constraint $\tau_6^{\mathrm{GSTP}}$ becomes unnecessary. Correctness of the modified circuit was verified *without* using this constraint in about $4$ minutes.

### Other Circuits

Among the other circuits we verified, we briefly report here on the modeling of two: the GasP control circuit [150] and the STAPL left-right buffer circuit [116].

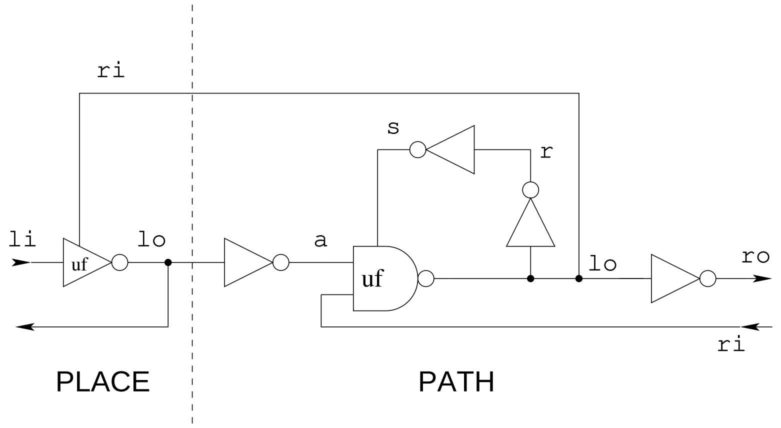



Figure 9.8: **GasP stage**

A single stage of the GasP control circuit is depicted at the gate-level in Figure 9.8 with normally distributed pullup and pulldown collapsed into the unfooted domino inverter. To ensure correct operation of this circuit, we needed to specify $4$ POD constraints for each stage. A sample constraint is

$$\mathtt{PATH.lo}\downarrow \;\rightarrow\; \mathtt{PATH.ri}\downarrow \;\prec\; \mathtt{PATH.s}\downarrow \qquad (\tau_1^{\mathrm{GASP}})$$

We connected $10$ GasP stages together in a ring with exactly one full stage, and model checked it for absence of short circuits and to verify that exactly one stage was full at any given point of time. Both verification runs completed within a minute, as shown in Table 9.3.

The STAPL left-right buffer, shown in Figure 9.9, is different from the other two circuits in that it uses metric timing constraints. Figure 9.9 shows a single FIFO stage that passes a single bit encoded using dual rail encoding (with signals `l0` and `l1`) to the output (as signals `r0` and `r1`). For correct operation, the circuit employs two pulse generators (shown in Figure 9.9 as square boxes) with pulse-lengths less than constants $\sigma_{\mathbf{true}}$ and $\sigma_{\mathbf{false}}$ respectively. Corresponding to the pulse generators, there are two paths in the circuit that are respectively required to take longer than constants $\xi_{\mathbf{true}}$ and $\xi_{\mathbf{false}}$. An additional constraint is imposed that $\xi_{\mathbf{true}} \geq \sigma_{\mathbf{true}}$ and $\xi_{\mathbf{false}} \geq \sigma_{\mathbf{false}}$. These timing constraints naturally lend themselves to being modeled as metric constraints with clock variables, with 2 constraints (4 clock variables) per buffer stage. In addition to these constraints, each stage also requires 6 POD constraints. Each stage has 10 Boolean signals (not counting its inputs; note that the pulse generators have one internal Boolean signal each). We model checked a ring of 3 STAPL buffers (for same properties as the GasP circuit); both verifications completed successfully within a few minutes.

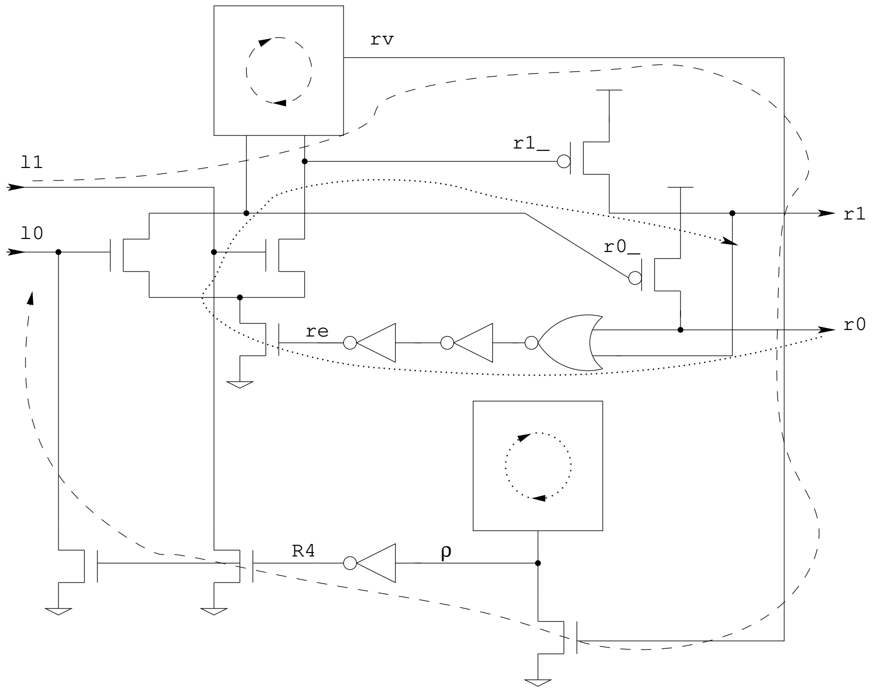



Figure 9.9: **STAPL left-right buffer.** Reproduced from [116]. The pulse generator with pulse width less than $\sigma_{\mathbf{true}}$ is shown with a dashed circle while that with width less than $\sigma_{\mathbf{false}}$ is shown using a dotted circle. The corresponding paths are dashed and dotted respectively.

**Comparison with Other Tools**

Table 9.3 summarizes our experimental results on the 3 circuits discussed so far.

| Circuit | $\lvert \mathcal{V} \rvert$ | $\lvert \mathcal{T} \rvert$ | | | TMV Run-Time (seconds) |
|---|---|---|---|---|---|
| | | POD | XOR | Metric | |
| Global STP | 28 | 27 | 6 | 0 | 66.32 |
| GasP-10 | 60 | 40 | 0 | 0 | 26.10 |
| STAPL-3 | 30 | 18 | 0 | 6 | 278.05 |

Table 9.3: **Summary of experimental results with TMV.** $|\mathcal{V}|$ is the number of signals, and $|\mathcal{T}|$ is the number of timing constraints with associated break-up into categories.

We compared the performance of TMV to ATACS [155], which is based on metric timing. ATACS uses model checking algorithms that are explicit-state in the Boolean component and prune the search space using partial-order reduction methods.[6] In modeling the Global STP (the corrected version) and STAPL circuits, we assigned min-max delay ranges to all gates so that timing is analogous to counting transitions, but for the GasP circuit we had to assign ranges more carefully so that all POD constraints were satisfied. For all three circuits, ATACS did not finish within an hour, running out of memory for the STAPL and Global STP circuits.

For the circuits discussed so far, most timing constraints are simple POD constraints, and very few constraints are metric. Hence, we only needed to introduce few clock variables, if any. This enabled TMV to scale well on these circuits.

As mentioned in Section 9.4.2, metric constraints can usually be reformulated as POD constraints, but at the cost of modularity. Using the STARI circuit [64], we studied the relative performance of TMV for two different ways of modeling constraints. (The reader is referred to Greenstreet's thesis [64] for a description of the circuit.) All timing constraints for this circuit can be modeled as POD constraints, where the POD is the clock that is distributed to both sender and receiver modules. This breaks modularity, since timing constraints for each buffer stage between the sender and the receiver require tracing back to the global clock. One can also formulate these constraints as metric timing constraints specifying that, for each buffer stage, an output data bit and `ack` must follow an input within a clock phase. In our circuit model, we abstracted the data-path to only one bit, and modeled only one of the two bits making up the dual rail encoding. Thus, each stage contributes only two Boolean state variables. The resulting timed automaton has $4$ clock variables (one per metric constraint) for every two stages; thus, there is exactly one clock variable for every Boolean signal in a stage.

We computed the set of reachable states for STARI circuits (initialized to be half-full) for increasing numbers of buffer stages and in three different ways: (1) using ATACS, (2) using TMV with purely POD constraints, and (3) using TMV with modularly specified metric constraints. The results are

[6]The results reported for ATACS are for the partial-order reduction option that yielded best results.

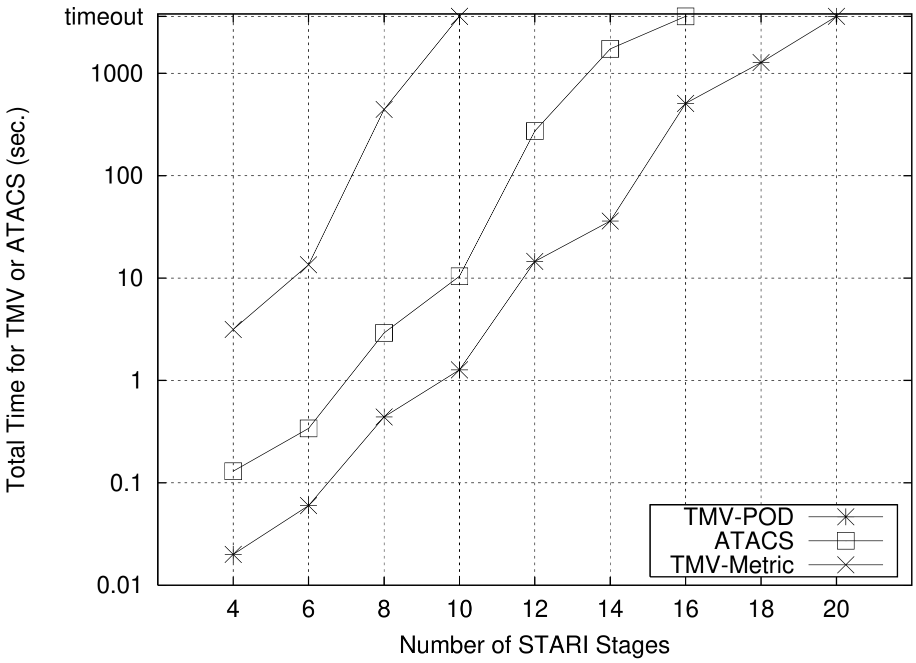



Figure 9.10: **Results for STARI circuit.** Note that the Y-axis is on a log scale. A timeout of 3600 seconds was imposed on all runs.

displayed in Figure 9.10. Using TMV with purely POD constraints is the most scalable approach, followed by ATACS. When used on a model with metric constraints, TMV scales very poorly. The reason for this appears to be that each clock zone has few corresponding Boolean states, since the ratio of clock variables to Boolean signals, per stage, is fairly high (compared to the STAPL buffer, for instance). This reduces the benefits of using fully symbolic Boolean methods of quantifier elimination. On the model based purely on POD constraints, TMV runs an order of magnitude faster than ATACS.

## 9.5 Summary

In this chapter, we presented a new approach for fully symbolic model checking of timed automata based on the Boolean methods for quantified difference logic proposed in Chapter 8. We have applied this model checker to the verification of timed circuits, including industrial examples. Results demonstrate the utility of our approach.

# Chapter 10

# Conclusion

This concluding chapter discusses the main theoretical results and design decisions in this thesis, summarizes the thesis contributions, and suggests directions for future work.

## 10.1 Summary of Contributions

Adaptive Boolean encoding methods provide a new way of building efficient, automated decision procedures for first-order logics involving arithmetic. This thesis has made a first step towards extending Boolean encoding methods to a rich subset of logic that is useful for a wide range of applications, and has demonstrated the efficiency of the approach.

A central design decision in our approach is to use eager Boolean encoding techniques. This enables us to leverage future advances in Boolean methods far more easily than the lazy encoding methods. In our experience, this clean separation of encoding and SAT can lead to orders of magnitude speedup on some problems. It also allows us to generate counterexamples fairly easily.

The eager encoding techniques are based on new theoretical results giving solution bounds for arbitrary quantifier-free Presburger arithmetic, as well as for specialized fragments such as the logic of G2SAT constraints. These results improve over previous solution bounds, in the typical case, by an exponential factor. The exponential improvement directly translates into an exponential reduction in the search space of the SAT solver.

Boolean encoding methods can be made adaptive by incorporating machine learning for automated algorithm selection. Our experience shows that the use of machine learning can not only relieve the user of the burden of setting the right combination of command-line options, but can also yield orders of magnitude speedup compared to previous approaches.

The UCLID verification system incorporated all of the above ideas, and has been applied to a wide range of applications in hardware and software verification. In this thesis, we demonstrated its application to finding format-string exploits, a class of security vulnerabilities in software that requires precise modeling of data.

Boolean encoding methods can also be used for quantifier elimination in quantified logics that admit such elimination. One such useful logic, explored in this thesis, is quantified difference logic. We have shown how quantifier elimination based on Boolean methods can be applied in the fully symbolic model checking of timed systems. In conjunction with a new approach to modeling timing assumptions in circuits, our fully symbolic model checker, TMV, has scaled to industrial-size circuits.

## 10.2 Open Problems

While this dissertation has answered many questions, it has also posed several new problems. We discuss some of the open problems here.

### Theoretical Problems

There are many open theoretical problems that deserve further exploration.

In deciding quantifier-free Presburger (QFP) arithmetic, we made use of the bound $(n+2)\cdot\Delta$ given by Borosh, Treybig, and Flahive (see Theorem 5.1). In their 1992 paper [23], Borosh and Treybig conjecture that this bound can be improved to just $\Delta$. As far as we know today, this conjecture is still open.

In Chapter 5, we showed how the presence of a large number of difference constraints can be exploited in computing a compact solution bound. Chapter 4 shows that the solution bound for G2SAT formulas is very similar to that for difference formulas. It is therefore a natural question as to whether the results of Chapter 5 can be generalized to apply to formulas comprising mainly G2SAT constraints.

The results of Chapter 5 apply to arbitrary-precision integer arithmetic. It would be interesting to see if similar results could be obtained for finite-precision (modular) integer arithmetic. In the latter case, we already have a (trivial) finite bound on solution size, but would like to find a tighter bound, or perhaps a way of performing a sparse encoding over the trivial finite bound.

### SAT and Machine Learning

In Chapter 6, we observed the impact of the structure of SAT instances generated by the DIRECT encoding on the relative performance with the SD encoding. More work needs to be done to formally understand the structure of SAT instances generated by both encoding algorithms. In particular, for the SD encoding, we have used only one choice of arithmetic circuits throughout this thesis (e.g., using ripple carry adders for addition). There needs to be a more comprehensive evaluation of different choices of arithmetic circuits in the SD encoding with respect to the ease with which the resulting SAT instances are solved.

Our experience with SAT solvers has been very positive, indicating the presence of hidden structure in the instances we generate (using all the different encoding algorithms). Formalizing this structure can help in designing more efficient SAT solvers, besides providing valuable theoretical insight into our application domains. The work of Hoos [77] and Williams et al. [163] are good starting points for tackling this problem.

Our work on using machine learning for automated algorithm selection, although demonstrated just for the SD and DIRECT encoding algorithms for difference logic, has wider applicability. Specifically, it could be used for different logical theories at multiple levels in the UCLID decision procedure. For example, it could be used for selecting between Ackermann's technique [2] for eliminating function applications and that given by Bryant et al. [29].

#### Applications

Automating the generation of formal models from descriptions of real hardware and software systems, in languages such as C and Verilog, is critical to make the techniques proposed in this thesis easier to use.

While there has been work on automatically generating finite-state models from source code using techniques such as predicate abstraction (e.g., [11, 36]), similar methods for generating infinite-state models have yet to be demonstrated. The recent work by Andraus and Sakallah [7] on generating UCLID models from Verilog is a first step in this direction.

Similarly, an important next step in our work on verifying timed circuits is to automate the generation of timing constraints. There has been past work on automatically generating relative timing constraints by attempting to rule out spurious counterexamples [85, 121], but they do not scale well, and require the use of min-max delay bounds. An approach worth exploring is to infer affine relations over time intervals between events based on applying machine learning to simulation traces; to our knowledge this has never been attempted before.

## 10.3 Looking Ahead

Boolean methods have the potential to greatly increase the scalability of decision procedures for first-order logics involving arithmetic, thereby enabling a whole new range of applications.

There are two directions in which a future research plan could be mapped out. The first involves extending the framework for decision procedures based on eager Boolean encodings that has formed the basis for this dissertation. The second concerns new applications that would benefit from a use of decision procedures.

Many applications generate not one, but a whole stream of formulas (queries) to a theorem prover. Often, these formulas differ from each other only slightly. For instance, in bounded model checking, the formula generated after symbolically simulating a system for $k$ steps, is likely to share several common sub-expressions with that generated after $k - 1$ steps. It is therefore advantageous to make the Boolean encoding algorithms incremental, with the ability to re-use work from previous translations. For the UCLID logic considered in this thesis, an incremental encoding algorithm will work in tandem with an incremental SAT solving algorithm. Algorithms and implementations for incremental SAT solving already exist [169]. In particular, one would like to prove theoretical results about the extent of additional work needed in the incremental translation, given a measure of how successive input formulas differ.

The decision procedures proposed in this thesis do not directly generate proofs. Proof-generating decision procedures are useful for at least two reasons. First, it provides the user with a certificate of the implementation's correctness on the input formula, making the system more trustworthy. Second, it can be used in verification tools that use proofs for refining abstractions, such as the Blast software model checker [69].

This dissertation has only explored purely eager Boolean encoding methods, and has demonstrated their advantages over lazy techniques for specific logics. However, eager methods suffer two limitations: the encoding phase can be a performance bottleneck, and it is harder to extend these methods for new theories. I believe that these limitations can be mitigated by an integration with lazy encoding techniques, for two reasons. First, when very little first-order reasoning is required for a given problem (e.g., if propositional reasoning suffices to decide unsatisfiability), lazy encoding methods are extremely effective. Second, lazy methods can easily build upon any method for deciding a combination of theories, such as Nelson and Oppen's method [109]. Some ideas on integrating eager and lazy algorithms are incorporated in the recent paper by Ganzinger et al. [60].

The second broad direction for future work is on new applications. New applications exist in hardware and software verification, as well as in other areas. Software security seems to be a particularly rich space for future applications. Finding security vulnerabilities in software often requires reasoning about data in addition to control, and theorem proving is particularly effective at analyzing data-dependent properties of systems. Some specific near-term applications are malware detection [39] and verifying secure information flow [130]. The work on timed circuits described in this thesis can be extended to other systems operating under timing assumptions, such as distributed systems and real-time embedded systems. Finally, automated reasoning in expressive logics has a wide range of potential applications, from established fields like constraint programming and operations research to emerging areas like systems biology.

Decision procedures will continue to play an important role in reasoning about the reliability and security of computer systems. Boolean methods can be exploited to build decision procedures that scale to industrial-scale problems. The theoretical concepts and practical techniques presented in this thesis form a foundation for future work on leveraging Boolean reasoning methods for richer logics.

# Appendix A

# UCLID

.

This appendix describes the syntax and semantics of the specification language for UCLID version 2.0, along with some of the verification methods that are supported in UCLID.

## A.1 The UCLID Specification Language

We present the semantics informally in the discussion accompanying the description of each syntactic construct.

The syntax of UCLID is very similar to the input language of the CMU version of the SMV model checker [98]. However, there are several differences, and we will point these out where necessary.

A specification in UCLID is divided into two logical sections. The first part describes the model of the system to be verified. The format of this part is very similar to that of SMV. The second part, also called the *control* section or module, specifies how the symbolic simulation is to be configured for the verification task at hand. One can view this latter portion as comprising of commands that one might ordinarily type at the cursor of an interactive tool.

### A.1.1 Format

The overall format of a UCLID specification is as follows:

```
MODEL <modelname>

<typedefs>
```

```
<Global Constants>

<modules>

<Control module>
```

`<modelname>` denotes the name of the specification being checked. `<typedefs>` is an optional section containing type definitions of user-defined enumerated types. Constants with global scope are defined in the following `<Global Constants>` section. This is followed by the specification of one or more modules. Each module has five subsections: an `INPUT` section that has declarations of inputs to the module, a `VAR` section for declaring state variables and macro variables, a `CONST` section for declaring constants, a `DEFINE` section for defining macros, and a `ASSIGN` section for defining the initial state and state transition relation of the module. The last section is the `<Control module>` section. This includes three mandatory subsections: the `EXTVAR` section for declaring external variables, the `STOREVAR` section for declaring storage variables, and the `EXEC` section for listing the commands to be used in the simulation. The optional subsections are `VAR`, `CONST` and `DEFINE`, which serve the same function as for ordinary modules. Note that the term "`MODEL`" is somewhat of a misnomer, since a UCLID specification contains both the model as well as commands to run the simulation.

### A.1.2 Language Overview

Before describing the language in detail, we present a simple example of a UCLID specification. Consider the example of a traffic light that changes based on the value of an internal timer, as shown in figure A.1.

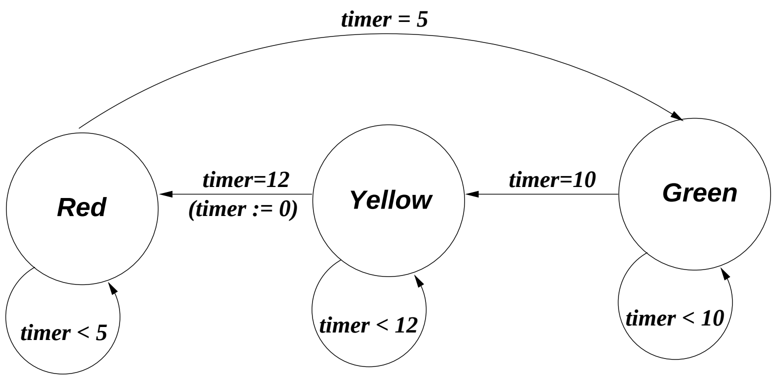



Figure A.1: A timed traffic light

The UCLID specification of this system is given below:

```
MODEL timedSignal

typedef signal : enum{red, yellow, green};


MODULE trafficLight

INPUT

VAR
(* state variables *)
light : signal;
timer : TERM;
(* macro variables *)
FIVE : TERM;
TEN : TERM;
TWELVE : TERM;

CONST

DEFINE
FIVE := succ^5(ZERO);
TEN := succ^5(FIVE);
TWELVE := succ^2(TEN);

ASSIGN
init[light] := red;
next[light] := case
  (light = red) & (timer < FIVE) : red;
  (light = red) & (timer = FIVE) : green;
  (light = green) & (timer < TEN) : green;
  (light = green) & (timer = TEN) : yellow;
  (light = yellow) & (timer < TWELVE) : yellow;
  (light = yellow) & (timer = TWELVE) : red;
  default : light;
esac;

init[timer] := ZERO;
next[timer] := case
  (light = yellow) & (timer = TWELVE) : ZERO;
  default : succ(timer);
esac;

(*----- CONTROL MODULE -----*)
CONTROL
```

```
EXTVAR

STOREVAR
initRedCondition : TRUTH;

VAR
redCondition : TRUTH;

CONST

DEFINE
redCondition := (trafficLight.light = red) =>
                (trafficLight.timer <= trafficLight.FIVE);

EXEC
initRedCondition := redCondition;
print(trafficLight.light);
decide(initRedCondition);
simulate(1);
decide(redCondition);
simulate(1);
decide(redCondition);
simulate(1);
decide(redCondition);
```

A traffic signal is modeled as a enumerated type with three values. Constants in UCLID can be Boolean, integer, of enumerated type, or uninterpreted symbols (we refer to such uninterpreted symbols as symbolic constants). In particular, the keyword `ZERO` refers to the integer constant 0. Within the module `trafficLight`, the `VAR` and `CONST` segments consist of variable and constant declarations respectively. Variables and constants declared here have names local to the module; however, these identifiers may be referenced anywhere outside the module by prefixing the identifier with the name of the module followed by a ".". The `DEFINE` segment has the same role as in CMU SMV – it is used to define "macros" for commonly occurring shared sub-expressions. The `ASSIGN` segment consists of assignments of initial values to state variables and specifications of the next state functions. The `case` expression is used for conditional assignments, just as in SMV.

The main syntactic additions (to the SMV style) illustrated in this example include the successor function symbol(`succ`), and the `CONTROL` module. The condition `redCondition`, defined in the control module, checks that the timer in the `red` state is always less than 5. We can easily see that this condition is always true for the specified model. The storage variable `initRedCondition` is used to store the initial value of this condition. In the above example, bounded model checking has been used to check the validity of `redCondition` for 3 steps. A `print` command is used to print the initial value of the state variable `trafficLight.light`. Note that the storage variable is unnecessary in this example; the formula stored in `initRedCondition` may be decided by inserting a `decide(redCondition)` statement before the first `simulate` command.

### A.1.3 Keywords and Lexical Conventions

The lexical analyzer of UCLID is case-sensitive. The following alphabetic strings are reserved keywords(some are reserved for future use).

```
MODEL CONTROL EXTVAR STOREVAR EXEC typedef enum initialize simulate
decide print printexpr FORALL MODULE INPUT VAR of CONST DEFINE ASSIGN
SPEC TERM TRUTH FUNC PRED ZERO succ pred case esac default init next Lambda
EXISTS verify model define if then else for endfor while do switch
array vector process function module procedure include boolean
integer signal input output OUTPUT local in end assert prove
```

Names of identifiers (state variables, macro variables, constants of all types) may be any sequence of symbols in {`A-Z,a-z,0-9,_`} beginning with an alphabetic character. Space, newline and tab are white spaces and are ignored. UCLID has ML-style comments, where the comment is enclosed begins with "(*" and ends with "*)". Nesting of comments is allowed.

While describing syntax in the discussion that follows, we will enclose within quotes all strings recognized as tokens by the parser. Identifiers will be denoted by the strings "id", "id0", "id1", etc.

### A.1.4 Data Types and Type Declarations

There are six classes of data types in UCLID, as listed below:

1. `TRUTH`, the Boolean data type;
2. `TERM`, the integer data type (uninterpreted function symbols of arity 0);
3. `FUNC`, the data type for uninterpreted function symbols of arity greater than 0. Functions of this type take arguments of type `TERM` and return a value of type `TERM`;
4. `PRED`, the data type for uninterpreted predicate symbols of arity greater than 0. Predicates of this type take arguments of type `TERM` and return a value of type `TRUTH`;
5. *Enumerated Types,* which are C-style enumerated types;
6. *Functions returning enumerated types*.

Enumerated types are the only user-defined types in UCLID. They must be declared at the very beginning of the UCLID specification using a `typedef` declaration, as given below:

```
type_decl ::= "typedef" id0 ":"
                  "enum" "{" id1 "," ... idn "}" ";"
```

An example is illustrated below:

```
        typedef signal : enum{red, yellow, green};
```

The scope of `typedef` declarations is global. A `typedef` declaration is mandatory for each enumerated type. After the `typedef` declaration, the enumerated type is to be referred by the type defined in that declaration.

Variable and constant declarations are made in `INPUT`, `VAR`, or `CONST` sections.[1] Types have the syntax

```
type ::= "TERM" | "TRUTH" | "FUNC" "[" integer "]"
        | "PRED" "[" integer "]"  | id
        | "FUNC" "[" integer "]" "of" id
```

Consider the following examples. Identifiers of type `TERM` and `TRUTH` are declared in a straightforward manner as shown below:

```
        foo : TRUTH;
        bar : TERM;
```

For functions and predicates, in additional to declaring the type, the user must also declare the arity. For functions returning an enumerated type, the enumerated type is also specified. In the examples below, `f` is a function of 10 arguments, `p` is a predicate of 4 arguments, and `manySignalLights` is a function of one argument that returns the type `signal`.

```
        f : FUNC[10];
        p : PRED[4];
        manySignalLights : FUNC[1] of signal;
```

Identifiers of type `FUNC` or `PRED` are useful in modeling arrays, lookup tables or memories, queues and similar data structures, using lambda expressions, as described in Chapter 7.

[1]External and storage variables are also declared in a similar fashion, but we will deal with these separately in section A.1.12.

Note that a function and a predicate of arity 0 may also be defined; however, in general, these do not always behave the same as if they were defined as TERM and TRUTH respectively. Functions or predicates of arity 0 are best defined as having TERM, TRUTH, or an enumerated type, as necessary.

### A.1.5 Constants

UCLID constants are of two kinds: *primitive* constants, and *symbolic* constants. *Primitive* constants are either of type TRUTH or of enumerated type. The primitive constants of type TRUTH are 1 and 0. There is only one primitive constant of type TERM: ZERO, standing for the integer constant 0. Primitive constants of an enumerated type E are the values in the set specified in the type declaration for E. Primitive constants do not have to be declared in a CONST declaration.

All constants other than primitive constants are *symbolic*. In UCLID version 2.0, there can be no symbolic constants of enumerated type, or of a FUNC type that returns enumerated type. All symbolic constants must be declared in a CONST declaration, either globally or within a module.

The syntax of a CONST declaration is as follows:

```
const_decl ::= "CONST"
                    id1 ":" type1 ";"
                    id2 ":" type2 ";"
                         ...
```

Examples of CONST declarations are given below:

```
CONST
       b0 : TRUTH;  (* symbolic Boolean constant *)
       T0 : TERM;   (* symbolic constant of type TERM *)
       f0 : FUNC[3]; (* symbolic constant of type FUNC and arity 3 *)
```

### A.1.6 Input Variables

Inputs to a module must be declared in the INPUT section of the module. Variables declared in this manner are called *input variables*.

Input variables are typically used to provide inputs to a module from the CONTROL module, where the value of the input signal in a given step may be controlled by the user. An input variable for module $M$ might also be a variable in another module $M'$ that $M$ references; the declaration is needed only if $M$ precedes $M'$ in the file.

The syntax of an INPUT declaration is as follows:

```
input_decl ::= "INPUT"
                    id1 ":" type1 ";"
                    id2 ":" type2 ";"
                         ...
```

UCLID version 2.0 does not support instantiation of modules within another module. We plan to implement this in the next version, and that will make different use of the INPUT section (substituting actual arguments for formal arguments).

### A.1.7 State Variables

A state of a UCLID model is an assignment of values to state variables.

The state variables of each module are declared in the VAR section of that module. A state variable may be of any of the six kinds of types discussed in section A.1.4. The syntax of a state variable declaration is as follows

```
var_decl ::= "VAR"
                    id1 ":" type1 ";"
                    id2 ":" type2 ";"
                         ...
```

In addition, to state variables, auxiliary and macro variables may be used to improve readability of the specification, and in verification. These variables must also be declared in a VAR declaration. They are typically defined in the DEFINE section.

### A.1.8 Macro Definitions

The DEFINE section of a module is used to define macros, especially for shared subexpressions, so as to improve readability. The syntax of a DEFINE declaration is as follows

```
defines ::= "DEFINE"
                    id1 ":=" expr1 ";"
                    id2 ":=" expr2 ";"
                         ...
```

Whenever any identifier that appears on the left hand side (LHS) of a DEFINE statement appears in an expression subsequent to its definition, it is replaced by the expression on the right hand side (RHS) of its DEFINE statement. It is an error to use a DEFINE identifier before its definition; circular definitions will also result in an error.

The RHS of a `DEFINE` statement is an expression whose syntax is defined in section A.1.10.

### A.1.9 State Assignments and the Transition Relation

The initial state assignment and the transition relation for state variables within a module are defined in the `ASSIGN` section.

The syntax of an `ASSIGN` declaration is as follows

```
assigns ::= "ASSIGN"
                  lval1 ":=" expr1 ";"
                  lval2 ":=" expr2 ";"
                       ...

   lval ::= "init" "[" id "]" | "next" "[" id "]"
```

Notice that UCLID syntax differs from SMV syntax in that we use square brackets instead of parentheses with the `init` and `next` strings.

An l-value, denoted above by `lval`, denotes either the initial state value of a state variable `v` (written `init[v]`, or the next state value of `v` (written `next[v]`). The expression on the RHS of an `init` assignment is evaluated prior to the simulation's run-time, and assigned to be the initial value of the state variable referenced on the LHS. For a `next` assignment, the expression is evaluated as the simulation is run, and will be the next state value of the state variable referenced on the LHS.

Expressions on the RHS of a next state assignment of a variable may reference the next state values of other state variables. It is therefore possible to have a combinational dependency amongst state variables arising from next state assignments. The UCLID interpreter extracts these dependencies automatically and evaluates the state variables in a suitable order. Circular dependencies are reported as errors; the interpreter in UCLID version 2.0 does not reproduce the dependencies in case of an error. The RHS of an initial state assignment may include other state variables, but no combinational dependencies are resolved, and if one arises, it is reported as a compile-time error. If the initial or next state of a state variable is assigned more than once, the last assignment is the only one that applies.

### A.1.10 Expressions

Expressions in UCLID are generated according to the following syntax:

```
expr ::= simple-expr
```

```
       | case-expr    /* Case expression */
       | nondet-expr  /* Nondeterministic expression */

simple-expr ::= truth-expr   /* Truth expression */
              | term-expr    /* Term expression */
              | enum-expr    /* Enum type expression */
              | func-expr    /* Function expression */
              | enum-fexpr   /* Enum type Function expression */
              | pred-expr    /* Predicate expression */
```

Note that parentheses can always be put around expressions, except for case-expressions and nondeterministic expressions, which don't need any. Parentheses also cannot be placed around `FORALL` expressions, which are introduced in Section A.1.12.

#### Truth expressions

Truth expressions or Boolean expressions, have type `TRUTH`. Their syntax is as follows:

```
truth-expr ::= 1 | 0  /* primitive Boolean constants */
             | id     /* symbolic Boolean constant or variable */
             | "next" "[" id "]"
                      /* Next state value of state variable */
             | "~" truth-expr1   /* Not */
             | truth-expr1 "&" truth-expr2 /* And */
             | truth-expr1 "|" truth-expr2 /* Or */
             | truth-expr1 "=>" truth-expr2 /* Implication */
             | truth-expr1 "<=>" truth-expr2 /* Equivalence */
             | term-expr1 "=" term-expr2   /* Equality */
             | term-expr1 "!=" term-expr2  /* Not-equality */
             | enum-expr1 "=" enum-expr2   /* Equality */
             | enum-expr1 "!=" enum-expr2  /* Not-equality */
             | term-expr1 "<" term-expr2   /* Less than */
             | term-expr1 ">" term-expr2   /* Greater than */
             | term-expr1 "<=" term-expr2   /* Less than or Equal */
             | term-expr1 ">=" term-expr2   /* Greater than or Equal */
             | pred-expr "(" term-expr1 "," term-expr2 ... ","
                           term-exprn ")" /* Predicate application */
```

The precedence of logical and relational operators is given below, from highest to lowest precedence.

```
=, !=, <, >, <=, >=
~
&
|
=> <=>
```

All operators of equal precedence associate to the left, except for =>, which associates to the right.

**Term expressions**

Term expressions have type `TERM`; they may be viewed as integers, although there are no primitive integer constants defined. Their syntax is as follows:

```
term-expr ::= id   /* symbolic constant or variable */
             | "next" "[" id "]"
                     /* Next state value of state variable */
             | "ZERO"   /* the integer constant 0 */
             | "succ" "(" term-expr ")"  /* term-expr + 1 */
             | "pred" "(" term-expr ")"  /* term-expr - 1 */
             | "succ^" k "(" term-expr ")"
                     /* term-expr + k, for constant
                        positive integer k */
             | "pred^" k "(" term-expr ")"
                     /* term-expr - k, for constant
                        positive integer k */
             | term-expr1 "+" term-expr2  /* integer addition */
             | term-expr1 "-" term-expr2  /* integer subtraction */
             | k "*" term-expr  /* multiplication by a positive integer
                                 constant k */
             | func-expr "(" term-expr1 "," term-expr2 ... ","
                           term-exprn ")" /* Function application */
```

**Enumerated type expression**

Enumerated type expressions evaluate to a user-defined enumerated type; their syntax is very similar to that of term expressions.

```
enum-expr ::= id   /* primitive constant or variable */
```

```
             | "next" "[" id "]"
                     /* Next state value of state variable */
             | enum-fexpr "(" term-expr1 "," term-expr2 ... ","
                        term-exprn ")" /* Enum Function application */
```

#### Function expressions

Function expressions evaluate to functions that take arguments of type TERM and return values of type TERM. A powerful feature of UCLID is to be able to define functions whose body changes over steps. This allows functions to model memories, queues, lists and other useful data structures.

```
func-expr ::= id   /* symbolic constant or variable */
            | "next" "[" id "]"
                     /* Next state value of state variable */
            | "Lambda" "." "(" id1 "," id2 ... "," idn
                          ")" term-expr
```

The list of arguments to the Lambda operator must have at least one element. Also, the arguments to a Lambda must be declared as symbolic constants. Both of these hold good for the Lambda operator in sections A.1.10 and A.1.10.

#### Function expressions returning enum type

Function expressions that take arguments of type TERM and return values of a user-defined enumerated type are also very useful.

```
enum-fexpr ::= id   /* symbolic constant or variable */
             | "next" "[" id "]"
                     /* Next state value of state variable */
             | "Lambda" "." "(" id1 "," id2 ... "," idn
                          ")" enum-expr
```

#### Predicate expressions

Predicate expressions evaluate to functions that take arguments of type TERM and return values of type TRUTH. Using the ability of UCLID to express lambda expressions, we can build, for example, predicate expressions that represent boolean state tables of arrays of processes.

```
pred-expr ::= id   /* symbolic constant or variable */
            | "next" "[" id "]"
                     /* Next state value of state variable */
            | "Lambda" "." "(" id1 "," id2 ... "," idn
                          ")" truth-expr
```

**Nondeterminism**

The UCLID syntax allows for expressions that evaluate to sets of values. Internally, fresh symbolic Boolean constants are generated to encode sets of values as an "if-then-else" expression conditioned on the values of these constants. These fresh Boolean constants have names of the form `_pN` where `N` is a natural number, and sometimes get assigned values in a counterexample.

The syntax of nondeterministic expressions is as follows:

```
nondet-expr ::= "{" simple-expr1 "," simple-expr2 ... ","
                    simple-exprn "}"
```

**Case expressions**

Conditional assignments are made using case expressions. The syntax of a case expression is as follows.

```
case-expr ::= simple-case-expr
            | lambda-case-expr

simple-case-expr ::= "case"
                        truth-expr1 ":" gen-expr1 ";"
                        truth-expr2 ":" gen-expr2 ";"
                          ...
                        default ":" gen-exprn ";"
                     "esac"

lambda-case-expr ::= "Lambda" "." "(" id1 "," id2 ... "," idm ")"
                     "case"
                        truth-expr1 ":" gen-expr1 ";"
                        truth-expr2 ":" gen-expr2 ";"
                          ...
                        default ":" gen-exprn ";"
```

```
                    "esac"

gen-expr ::= truth-expr
           | term-expr
           | enum-expr
           | "{" truth-expr1 "," ... "," truth-exprn "}"
           | "{" term-expr1 "," ... "," term-exprn "}"
           | "{" enum-expr1 "," ... "," enum-exprn "}"
```

Note that we use the C-style `default` for the last item in the case as opposed to the SMV-style `1`. Nesting of case expressions is not allowed.

### A.1.11 Modules

A module is used to collect together related state variables and associated constants, macro definitions and state assignments. UCLID version 2.0 has limited module support, and provides essentially two features. First, we allow local naming, where variables with same names can be declared in different modules. Second, we also allow the use of input signals from other modules, including the Control module. This latter feature allows the user to configure a simulation as needed. Note that UCLID version 2.0 does not allow one to instantiate modules within other modules.

The syntax of a module definition (other than the Control module) is as follows:

```
module ::= "MODULE" id
              "INPUT"
                  ... /* input variable declarations */
              "VAR"
                  ... /* state variable and macro declarations */
              "CONST"
                  ... /* symbolic constant declarations */
              "DEFINE"
                  ... /* macro definitions */
              "ASSIGN"
                  ... /* state variable assignments */
```

### A.1.12 The Control Module

The Control module allows the user to configure the symbolic simulation for the verification task at hand. In section A.2, we describe some of the verification techniques that UCLID can be used for, and how the Control module can be used for those techniques.

The syntax of the Control module is as follows:

```
control ::= "CONTROL"
            "EXTVAR"
               ...   /* external variable declarations */
            "STOREVAR"
               ...   /* storage variable declarations */
            "VAR"
               ...   /* macro variable declarations */
            "CONST"
               ...   /* symbolic constant declarations */
            "DEFINE"
               ...   /* macro definitions */
            "EXEC"
               ...   /* simulator commands */
```

The `VAR`, `CONST` and `DEFINE` segments of the Control module serve exactly the same purpose as for any other module, and have the same syntax. The `VAR`, `CONST` and `DEFINE` sections are optional. The `VAR` segment will not contain any declarations for state variables as there are no state variables in the Control module.

### External Variables

In symbolic simulation, the user might sometimes wish to control the value a state variable takes at a specific step. For example, in correspondence checking using the method pioneered by Burch and Dill, one side of the commutative diagram is a simulation that first performs flushing, projection, and then executes a step of the specification machine, while the other side of the diagram first executes a step of the implementation machine, and then performs flushing. In this case, the flush signal needs to take on specific values at specific steps, and these steps are different depending upon which side of the commutative diagram we are trying to simulate.

The *external variable* is a feature of UCLID that addresses this problem. An external variable is a user-controlled input to the system that can be assigned specific values at specific steps. An external variable declaration includes, in addition to the type declaration, an assignment of the default value that the variable takes, as shown here:

```
extvar_decl ::= "EXTVAR"
                   id1 ":" type1 ":=" expr1 ";"
```

```
                    id2 ":" type2 ":=" expr2 ";"
                        ...
```

External variables are also declared as inputs to modules before they are declared in the Control module (however, when they are declared as inputs, no default value is assigned). It is an error to declare an external variable that is not an input to any module.

The value of an external variable at step $i$ is used in the simulation at step $i + 1$. For example, for external variable `flush`, the assignment

```
flush[3] := 0;
```

means that the value of flush used in the fourth step of simulation is $0$.

External variables find use in verification tasks where the values of variables at certain steps must be user-specified, such as in correspondence checking. For example, they are used in the flushing operation for verifying pipelined processor designs by the Burch-Dill method [34].

### Storage Variables

During symbolic simulation, one might wish to store intermediate values of variables and expressions for later reference. Storage variables serve precisely this purpose.

The syntax of a storage variable declaration is as follows:

```
storevar_decl ::= "STOREVAR"
                     id1 ":" type1 ";"
                     id2 ":" type2 ";"
                          ...
```

### Commands and Assignments

The `EXEC` section of the Control module contains 4 kinds of commands and two kinds of assignments. The syntax of an `EXEC` section is as follows:

```
exec ::= "EXEC"
            stmt1 ";"
            stmt2 ";"
              ...

stmt ::= "simulate" "(" integer ")"      /* Simulate command */
```

```
        | "initialize"                       /* Re-initialize all state */
        | "decide" "(" gen-truth-expr ")" /* Decide command */
        | "print" "(" id ")" /* Print the value of a state variable */
        | "print" "(" `"` string `"` ")"  /* Print any arbitrary string
                                             enclosed in double quotes */
        | "printexpr" "(" expr ")" /* Print the value of an expr */
        | id ":=" expr            /* Storage variable assignment */
        | id "[" integer "]" ":=" expr
                                  /* external variable assignment */
```

The *simulate* command takes an integer argument $k$ that specifies the number of steps the symbolic simulation is to be run for, and simulates the system for $k$ steps. The *initialize* command re-initializes all state variables in the system to their initial value. This is useful, for instance, while doing correspondence checking.

The *decide* command takes as argument a "generalized" truth-expression. The syntax of this generalized truth expression is given below:

```
gen-truth-expr ::= truth-expr
                 | forall-truth-expr "=>" truth-expr
                 | forall-truth-expr1 "=>" forall-truth-expr2

forall-truth-expr ::= "FORALL" "(" id1 "," id2 ... "," idn ")"
                            truth-expr
```

A generalized truth expression is either an ordinary truth-expression, as introduced in section A.1.10, or an expression of the form $A \Rightarrow C$ where the antecedent $A$ has some variables (of type `TERM`) universally quantified, while the consequent $C$ may or may not have universally quantified variables (of type `TERM`). The list of arguments to the `FORALL` operator must have at least one element. We will describe how this syntactic feature is used in section A.2.

UCLID version 2.0 provides two commands for printing: *print* and *printexpr*. The *print* command has two variants. The first allows one to print the value of any state variable at any step. The second allows the user to print an arbitrary string enclosed in double quotation marks, primarily for pretty formatting of the output. The *printexpr* command allows one to print the value of any expression (respecting the syntax of `expr`) at the current simulation step.

The size of the output generating by printing the values of state variables and expressions produces blows up very quickly as the number of simulation steps increases; we therefore strongly discourage printing state variables and expressions after a very large number of simulation steps unless they are known to be small.

Assignments to storage variables are similar to macro definitions. The storage variable name appears on the LHS of the assignment, and it can be assigned an expression of its type. Assignments to external variables also need to specify the step of simulation the RHS expression is to be evaluated at. At that step, the expression is evaluated and the value is used wherever the external variable is used. The natural number specifying the simulation step is written in square brackets on the LHS next to the external variable name.

## A.2 Verification with UCLID

UCLID version 2.0 can be used with several verification methods, as was briefly described in Section 7.3. We describe the more commonly used techniques in this section using the language constructs introduced in Section A.1. Using the primitive constructs described in Section A.1, the user can easily develop techniques based on symbolic simulation other than those listed below.

#### Bounded Model Checking

Plain symbolic simulation or *bounded model checking* can be done by simply running the `simulate` command, specifying the number of steps as an argument. The `decide` command can then be used to check the validity of a property of interest in a given state. This can be a very useful bug-finding tool.

Bounded model checking can be used to check safety properties (state invariants) for a bounded number of simulation steps. If the property does not hold for any state, UCLID generates a counterexample that can be used to generate a trace showing how the bug may be exploited. However, if the property holds for all states in the simulation, we cannot make any assertions about whether it will continue to hold for future steps.

Limited checking of liveness properties is also possible. For example, if we wish to check if a process releases a lock eventually (starting from an initial state) and if the symbolic simulation leads to such a state, then, we can assert that the property does indeed hold. However, we cannot find counterexamples for such a liveness property (if it does not hold on a truncated run).

The example in section A.1.2 illustrates the use of UCLID for bounded model checking. In bounded model checking, all state variables are initialized to their initial state values using the `init` statement. To check the validity of safety properties of interest after each step of simulation, the user inserts `decide` commands after the corresponding `simulate` commands. If the formulas are valid at each step up to $k$ steps starting from an initial state $s_0$, then the safety property of interest holds for the first $k$ steps starting from $s_0$. We have found bounded model checking to be useful in catching bugs, especially as a first step before trying to verify the system using techniques such as correspondence checking or inductive invariant checking.

### Correspondence Checking

Correspondence checking involves simulating two different sides of a commutative diagram and checking the validity of the property of interest at the end [34, 158]. Thus, the outline of the verification task, as specified in the Control module of a UCLID specification, will be as follows:

1. Assign values of external variables at specific steps in the simulation, using external variable assignments.
2. Run the simulation for one side of the diagram, using the `simulate` command.
3. Save the values of relevant state variables using storage variables.
4. Re-initialize to the start state, using the `initialize` command.
5. (Re-)Assign values of external variables at different steps.
6. Run the simulation for the other side of the diagram.
7. (Optional) Save the values of relevant state variables in storage variables.
8. Construct a formula for the property of interest, and check its validity by using the `decide` command.

### Deductive Verification

Another verification technique that UCLID can be used on is to prove the inductive invariant of a system. In this technique, the starting state is initialized to a most general state. The system is symbolically simulated for one step. Then, a property of the form $Inv \Rightarrow Next(Inv)$ is checked, where $Inv$ denotes a formula for the invariant property we wish to verify, and $Next(Inv)$ is its next-state version.

In general, the property $Inv$ will need to be augmented by several other auxiliary invariants, just as is often the case in theorem proving. The user has to come up with "lemmas" to prove the inductive invariant, but the process of checking the validity of these lemmas is entirely automatic. The UCLID counterexample generator is very useful in providing hints for lemmas.

**Quantifiers and Antecedent Instantiation**

Formulas that are checked for validity using the `decide` command are formulas in the CLU logic. This logic can express any property in quantifier-free first order logic involving counter arithmetic. It is often the case that properties of interest involve quantifiers. In particular, many properties involve the use of the universal (∀) quantifier. UCLID version 2.0 provides limited support for specifying properties with universal quantifiers.

There are three classes of quantified formulas that UCLID version 2.0 can handle:

1. *Universal Quantification on the outside of a quantifier-free formula:* The general form of a property of this kind is

   $$\forall i_1.\forall i_2 \ldots \forall i_k.P(i_1, i_2, \ldots, i_k)$$

   where $P(i_1, i_2, \ldots, i_k)$ is an arbitrary formula in CLU where the $i_j$s have type `TERM`. Since a universally quantified formula is valid if and only if the a formula without the quantifiers is valid (i.e., a formula in which the $i_j$s appear free), this case can be expressed by simply dropping the quantifiers, and expressing the quantifier-free formula in UCLID syntax.

   For example, consider the formula below:

   $$\forall i.\forall j.(i \neq j) \Rightarrow (f(i) \neq f(j))$$

   This can be expressed in UCLID syntax quite simply as

   ```
   (i != j) => (f(i) != f(j))
   ```

   We have found that most properties fall under this case.

2. *Universal Quantification only over variables appearing in the antecedent:* The general form of a formula $p$ of this kind is

   $$(\forall i_1.\forall i_2 \ldots \forall i_k.A(i_1, i_2, \ldots, i_k)) \Rightarrow C$$

   where $A(i_1, i_2, \ldots, i_k)$ and $C$ are arbitrary formulas in CLU, and $i_1, i_2, \ldots, i_k$ do not appear free in $C$.

   Notice that by pulling out the universal quantifiers, $p$ can be rewritten as

   $$\exists i_1.\exists i_2 \ldots \exists i_k.(A(i_1, i_2, \ldots, i_k) \Rightarrow C)$$

   Formula $p$ can be verified in UCLID in two ways. The first method involves proving a more conservative version of $p$, namely the formula

   $$\forall i_1.\forall i_2 \ldots \forall i_k.(A(i_1, i_2, \ldots, i_k) \Rightarrow C)$$

Notice that the above formula is of the kind handled in item 1 above, and so can be translated to an equivalent formula in CLU.

Often, the more conservative property fails to hold, and other techniques are needed. The second method involves the use of *instantiation*. Instantiation is the process by which the universal quantifier over the antecedent of $p$ is converted to a finite conjunction of instances of the antecedent. Each instance is generated by assigning a symbolic constant to a quantified variable, and dropping the universal quantifier over that variable. For example, the above formula $p$ would get translated to a new formula $p_{inst}$ given below

$$(\bigwedge_{j_1,\ldots,j_k=1}^{n_1,n_2,\ldots,n_k} A(t_{j_1}, t_{j_2}, \ldots, t_{j_k})) \Rightarrow C$$

This procedure is sound, but necessarily incomplete, because it would otherwise imply the decidability of first-order logic. In other words, if $p_{inst}$ is valid, so is $p$, but $p$ could be valid without $p_{inst}$ being valid. We have found that using an instantiation technique is often useful in proving the validity of the property of interest.

UCLID version 2.0 incorporates a simple heuristic strategy to instantiate the antecedent, which has had some success. The strategy essentially involves instantiating each quantified variables with all relevant terms from the consequent formula $C$. Further details of this procedure are available elsewhere [89].

Instantiation may be specified in the UCLID language as follows. For the property $p$ given above, the user would write a corresponding UCLID formula (of type `TRUTH`) as given below (assume $k = 2$)

```
FORALL(i1, i2) A(i1, i2) => C
```

where `A` and `C` are UCLID truth-expressions corresponding to $A$ and $C$ above, respectively.

3. *Universal Quantification performed separately over variables appearing in the antecedent and in the consequent:* The general form of a formula $q$ of this kind is

$$(\forall i_1.\forall i_2 \ldots \forall i_k.A(i_1, i_2, \ldots, i_k)) \Rightarrow (\forall j_1.\forall j_2 \ldots \forall j_n.C(j_1, j_2, \ldots, j_n))$$

where $A(i_1, i_2, \ldots, i_k)$ and $C(j_1, j_2, \ldots, j_n)$ are arbitrary formulas in CLU so that $i_1, i_2, \ldots, i_k$ do not appear free in $C(j_1, j_2, \ldots, j_n)$, and $j_1, j_2, \ldots, j_n$ do not appear free in $A(i_1, i_2, \ldots, i_k)$.

$q$ is equivalent to the following formula

$$\forall j_1.\forall j_2 \ldots \forall j_n.(\forall i_1.\forall i_2 \ldots \forall i_k.A(i_1, i_2, \ldots, i_k) \Rightarrow C(j_1, j_2, \ldots, j_n))$$

which in turn is equivalent to

$$(\forall i_1.\forall i_2 \ldots \forall i_k.A(i_1, i_2, \ldots, i_k) \Rightarrow C(j_1, j_2, \ldots, j_n))$$

Notice that the last formula above is in the form of item 2 above. Therefore, we can handle this formula using the conservative approach and the instantiation techniques described in item 2.

However, UCLID version 2.0 allows the user to be explicit about which variables are being universally quantified in the consequent $C$. Thus, for $k = 1$ and $n = 2$, $q$ may be written in UCLID as

```
FORALL(i1) A(i1) => FORALL(j1, j2) C(j1, j2)
```

In UCLID version 2.0, this will have exactly the same effect as writing

```
FORALL(i1) A(i1) => C(j1, j2)
```

Automatic instantiation is a fairly expensive operation — the formula blows up exponentially with increase in the number of variables to be instantiated. Fortunately, automatic instantiation need not always be done. Consider the class of properties that impose constraints on the values of state variables. In these cases, the user can encode the invariant into the `init` state assignment to those variables. Such an invariant has the form $v = P$, where $v$ is a state variable and $P$ is a case expression enumerating all the possible expressions $v$ can evaluate to along with the conditions under which $v$ can equal them.

In the case of inductive invariant checking, if the invariant formula on a variable $v$ is denoted by $Inv$, then instead of checking the validity of a formula of the form $Inv \Rightarrow Next(Inv)$, we merely encode $Inv$ into the initial state of $v$, simulate for one step, and then check $Next(Inv)$.

Consider the example of section A.1.2. Suppose we wanted to prove the following property using inductive invariant checking:

```
(trafficLight.timer = ZERO) => (trafficLight.light = red)
```

We could encode the invariant into the initial state as follows:

```
init[light] := case
                   timer = ZERO : red;
                   default      : {red, yellow, green};
               esac;
```

**Further Information**

More information on UCLID usage can be found in the user's manual [137].

# Bibliography


[1] L. Aceto, P. Bouyer, A. Burgueno, and K. G. Larsen. The power of reachability testing for timed automata. *Theoretical Computer Science*, 1-3(300):411–475, May 2003.

[2] W. Ackermann. *Solvable Cases of the Decision Problem*. North-Holland, Amsterdam, 1954.

[3] Rajeev Alur. Timed automata. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22. Springer-Verlag, July 1999.

[4] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[5] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[6] Rajeev Alur and Thomas A. Henzinger. Logics and models of real time: A survey. In *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 74–106. Springer-Verlag, July 1991.

[7] Zaher S. Andraus and Karem A. Sakallah. Automatic abstraction and verification of verilog models. In *Proceedings of the 41st Design Automation Conference (DAC)*, pages 218–223, 2004.

[8] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Korniowicz, and Roberto Sebastiani. A SAT based approach for solving formulas over boolean and linear mathematical propositions. In Andrei Voronkov, editor, *Proc. 18th International Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 195–210. Springer-Verlag, July 2002.

[9] Gilles Audemard, Alessandro Cimatti, Artur Korniowicz, and Roberto Sebastiani. Bounded model checking for timed systems. In Doron Peled and Moshe Y. Vardi, editors, *Formal Techniques for Networked and Distributed Systems - FORTE'02*, volume 2529 of *Lecture Notes in Computer Science*, pages 243–259. Springer, November 2002.

[10] T. Ball, B. Cook, S. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *Proc. Computer-Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 457–461, 2004.

[11] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, pages 203–213, June 2001.

[12] C. Barrett, D. L. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, LNCS 1166, pages 187–201. Springer-Verlag, November 1996.

[13] C. Barrett, D. L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In E. Brinksma and K. G. Larsen, editors, *Proc. 14th Intl. Conference on Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 236–249. Springer-Verlag, July 2002.

[14] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proc. 16th International Conference on Computer-Aided Verification (CAV)*, pages 515–518, July 2004.

[15] Gerd Behrmann, Kim Guldstrand Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient timed reachability analysis using clock difference diagrams. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 341–353. Springer-Verlag, July 1999.

[16] Wendy Belluomini. *Algorithms for Synthesis and Verification of Timed Circuits and Systems*. PhD thesis, University of Utah, 1999.

[17] S. Berezin, V. Ganesh, and D. L. Dill. An online proof-producing decision procedure for mixed-integer linear arithmetic. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, LNCS 2619, pages 521–536, April 2003.

[18] Dirk Beyer. Improvements in BDD-based reachability analysis of timed automata. In Jose Nuno Oliveira and Pamela Zave, editors, *Proceedings of the 10th International Symposium of Formal Methods Europe (FME)*, volume 2021 of *Lecture Notes in Computer Science*, pages 318–343. Springer-Verlag, 2001.

[19] Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Springer-Verlag, 1997.

[20] I. Borosh, M. Flahive, D. Rubin, and L. B. Treybig. A sharp bound for solutions of linear Diophantine equations. *Proceedings of the American Mathematical Society*, 105(4):844–846, April 1989.

[21] I. Borosh, M. Flahive, and L. B. Treybig. Small solutions of linear Diophantine equations. *Discrete Mathematics*, 58:215–220, 1986.

[22] I. Borosh and L. B. Treybig. Bounds on positive integral solutions of linear Diophantine equations. *Proceedings of the American Mathematical Society*, 55(2):299–304, March 1976.

[23] I. Borosh and L. B. Treybig. A sharp bound on positive solutions of linear Diophantine equations. *SIAM Journal on Matrix Analysis and Applications*, 13(2):454–458, April 1992.

[24] M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In O. Grumberg, editor, *Computer-Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 179–190. Springer-Verlag, 1997.

[25] Joel Brenner and Larry Cummings. The Hadamard maximum determinant problem. *American Mathematical Monthly*, 79:626–630, June-July 1972.

[26] R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear programming. In *Proceedings of the IEEE VLSI Design Conference*, pages 741–746, 2002.

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

[28] R. E. Bryant, S. German, and M. N. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification (CAV '99)*, LNCS 1633, pages 470–482. Springer-Verlag, July 1999.

[29] R. E. Bryant, S. German, and M. N. Velev. Processor verification using efficient reductions of the logic of uninterpreted functions to propositional logic. *ACM Transactions on Computational Logic*, 2(1):1–41, January 2001.

[30] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In E. Brinksma and K. G. Larsen, editors, *Proc. Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 78–92, July 2002.

[31] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Convergence testing in term-level bounded model checking. In *Correct Hardware Design and Verification Methods (CHARME)*, volume 2860 of *Lecture Notes in Computer Science*, pages 348–362. Springer-Verlag, October 2003.

[32] R. E. Bryant and M. N. Velev. Boolean satisfiability with transitivity constraints. In A. Emerson and P. Sistla, editors, *Computer-Aided Verification (CAV 2000)*, LNCS 1855. Springer-Verlag, July 2000.

[33] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. In $5^{th}$ *IEEE Symposium on Logic in Computer Science (LICS)*, pages 428–439, 1990.

[34] J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In D. L. Dill, editor, *Computer-Aided Verification (CAV '94)*, LNCS 818, pages 68–80. Springer-Verlag, June 1994.

[35] Christopher J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2:121–167, 1998.

[36] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *Proc. 25th International Conference on Software Engineering (ICSE)*, pages 385–395, 2003.

[37] Vijay Chandru. Variable elimination in linear constraints. *The Computer Journal*, 36(5):463–472, August 1993.

[38] Vijay Chandru and John Hooker. *Optimization Methods for Logical Inference*. John Wiley & Sons, 1999.

[39] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proc. IEEE Symposium on Security and Privacy*, May 2005. To appear.

[40] R. Clarisó and J. Cortadella. Verification of timed circuits with symbolic delays. In *Proc. Conference on Asia South Pacific Design Automation (ASP-DAC)*, pages 628–633, 2004.

[41] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[42] CMU SMV. Available at `http://www.cs.cmu.edu/~modelcheck/smv.html`.

[43] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 24. MIT Press, second edition, 2001.

[44] Olivier Coudert and Jean Christophe Madre. A unified framework for the formal verification of sequential circuits. In *Proc. Intl. Conference on Computer-Aided Design (ICCAD'90)*, pages 126–129, 1990.

[45] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic protection from `printf` format-string vulnerabilities. In *Proc. 10$^{th}$ Security Symp.* USENIX, 2001.

[46] CPLEX Optimization Tool. Available from ILOG. `http://www.ilog.com/products/cplex/`.

[47] Colorado University Decision Diagrams Package (CUDD). Available at `http://vlsi.colorado.edu/~fabio/CUDD`.

[48] CVC-Lite: Cooperating Validity Checker. Available at `http://verify.stanford.edu/CVCL/`.

[49] G. B. Dantzig and B. C. Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory A*, 14:288–297, 1973.

[50] M. Dash and H. Liu. Feature selection for classification. *Intelligent Data Analysis*, 1(3):131–156, 1997.

[51] Leonardo de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Proc. 18th International Conference on Automated Deduction (CADE)*, pages 438–455, 2002.

[52] Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. In *Proc. 15th International Conference on Computer-Aided Verification (CAV)*, number 2725 in LNCS, pages 14–26, 2003.

[53] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories Palo Alto, 2003.

[54] M. J. Fischer and M. O. Rabin. Super-exponential complexity of Presburger arithmetic. *Proceedings of SIAM-AMS*, 7:27–41, 1974.

[55] C. Flanagan, K. R. M. Leino, M. Lillibridge, C. G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, June 2002.

[56] Cormac Flanagan, Rajeev Joshi, Xinming Ou, and James B. Saxe. Theorem proving using lazy proof explication. In *Proc. 15th International Conference on Computer-Aided Verification (CAV)*, pages 355–367, 2003.

[57] J. S. Foster, M. Fahndrich, and A. Aiken. A theory of type qualifiers. In *Proc. ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 1999.

[58] Vinod Ganapathy, Sanjit A. Seshia, Somesh Jha, Thomas W. Reps, and Randal E. Bryant. Automatic discovery of API-level exploits. In *Proc. 27th International Conference on Software Engineering (ICSE)*, May 2005. To appear.

[59] V. Ganesh, S. Berezin, and D. L. Dill. Deciding Presburger arithmetic by model checking and comparisons with other methods. In *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, pages 171–186. Springer-Verlag, November 2002.

[60] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In *Proc. 16th International Conference on Computer-Aided Verification (CAV)*, pages 175–188, July 2004.

[61] Steven German. Personal communication.

[62] A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal. BDD based procedures for a theory of equality with uninterpreted functions. In A. J. Hu and M. Y. Vardi, editors, *Computer-Aided Verification (CAV '98)*, LNCS 1427, pages 244–255. Springer-Verlag, June 1998.

[63] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT solver. In *Design Automation and Test in Europe (DATE) 2002*, pages 142–149, 2002.

[64] Mark R. Greenstreet. *STARI: A Technique for High-Bandwidth Communication*. PhD thesis, Princeton University, 1993.

[65] Y. Gurevich. The decision problem for standard classes. *The Journal of Symbolic Logic*, 41(2):460–464, June 1976.

[66] Warwick Harvey and Peter J. Stuckey. A unit two variable per inequality integer constraint solver for constraint logic programming. In *Proceedings of the Twentieth Australasian Computer Science Conference*, pages 102–111, 1997.

[67] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proc. 31st ACM Symposium on Principles of Programming Languages*, pages 232–244, Jan 2004.

[68] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Gregoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. In *Proc. 14th International Conference on Computer-Aided Verification (CAV)*, LNCS 2404, pages 526–538, 2002.

[69] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Proc. 29th ACM Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[70] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Temporal proof methodologies for real-time systems. In *18th Annual ACM Symposium on Principles of Programming Languages*, pages 353–366. ACM press, 1991.

[71] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[72] G. Hinton, M. Upton, D.J. Sager, D. Boggs, D. M. Carmean, P. Roussel, T. I. Chappell, T. D. Fletcher, M. S. Milshtein, M. Sprague, S. Samaan, and R. Murray. A 0.18 CMOS IA-32 processor with a 4-GHz integer execution unit. *IEEE Journal of Solid-State Circuits*, 36(11):1617–1627, November 2001.

[73] D. Hochbaum. *Approximation Algorithms for NP-Hard Problems*, chapter 3. PWS Publishing Company, 1995.

[74] D. Hochbaum and J. Naor. Simple and fast algorithms for linear and integer programs with two variables per inequality. *SIAM Journal on Computing*, 23(6):1179–1192, 1994.

[75] Dorit Hochbaum, N. Megiddo, J. Naor, and A. Tamir. Tight bounds and 2-approximation algorithms for integer programs with two variables per inequality. *Mathematical Programming*, 62:63–92, 1993.

[76] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code*. Addison Wesley, Boston, MA, 2004.

[77] Holger H. Hoos. SAT-encodings, search space structure, and local search performance. In *Proceedings of the International Joint Conference in Artificial Intelligence (IJCAI)*, pages 296–303, 1999.

[78] David Hosmer and Stanley Lemeshow. *Applied Logistic Regression*. Wiley and Sons, New York, 1989.

[79] C-W. Hsu, C-C. Chang, and C-J. Lin. A practical guide to support vector machine classification, 2003. Available electronically as `http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf`.

[80] ICS: Integrated Canonizer and Solver. Available at `http://www.icansolve.com`.

[81] Joxan Jaffar, Michael J. Maher, Peter J. Stuckey, and Roland H. C. Yap. Beyond finite domains. In *2nd International Workshop on Principles and Practice of Constraint Programming (PPCP'94)*, volume 874 of *Lecture Notes in Computer Science*, pages 86–94, 1994.

[82] Mark Jerrum, Leslie G. Valiant, and Vijay V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science*, 43:169–188, 1986.

[83] Thorsten Joachims. Making large-scale SVM learning practical. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*. MIT Press, 1999.

[84] R. Kannan and C. L. Monma. On the computational complexity of integer programming problems. In *Optimisation and Operations Research*, volume 157 of *Lecture Notes in Economics and Mathematical Systems*, pages 161–172. Springer-Verlag, 1978.

[85] H. Kim, P. A. Beerel, and K. S. Stevens. Relative timing based verification of timed circuits and systems. In *8th International Symposium on Asynchronous Circuits and Systems*, pages 115–126. IEEE Press, April 2002.

[86] Manolis Koubarakis. Complexity results for first-order theories of temporal constraints. In Jon Doyle, Erik Sandewall, and Pietro Torasso, editors, *Proc. 4th International Conference on Principles of Knowledge Representation and Reasoning (KR'94)*, pages 379–390. Morgan Kaufmann, 1994.

[87] Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, and Ofer Strichman. Abstraction-based satisfiability solving of Presburger arithmetic. In *Proc. 16th International Conference on Computer-Aided Verification (CAV)*, pages 308–320, July 2004.

[88] Michail Lagoudakis, Michael L. Littman, and Ronald Parr. Selecting the right algorithm. In *Proceedings of the 2001 AAAI Fall Symposium Series: Using Uncertainty within Computation*, November 2001.

[89] S. K. Lahiri, S. A. Seshia, and R. E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, pages 142–160. Springer-Verlag, November 2002.

[90] Shuvendu K. Lahiri and Randal E. Bryant. Deductive verification of advanced out-of-order microprocessors. In *Proc. 15th International Conference on Computer-Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 341–354, 2003.

[91] Shuvendu K. Lahiri, Randal E. Bryant, and Byron Cook. A symbolic approach to predicate abstraction. In *Proc. 15th International Conference on Computer-Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 141–153, 2003.

[92] LASH Toolset. Available at `http://www.montefiore.ulg.ac.be/~boigelot/research/lash`.

[93] Kevin Leyton-Brown, Eugene Nudelman, Galen Andrew, Jim McFadden, and Yoav Shoham. A portfolio approach to algorithm selection. In *Proceedings of the International Joint Conference in Artificial Intelligence (IJCAI)*, pages 1542–1543, 2003.

[94] Oded Maler and Amir Pnueli. Timing analysis of asynchronous circuits using timed automata. In *Proc. Correct Hardware Design and Verification Methods (CHARME)*, pages 189–205, 1995.

[95] P. Manolios and S. K. Srinivasan. Automatic verification of safety and liveness for xscale-like processor models using web refinements. In *Design, Automation, and Test in Europe (DATE)*, pages 168–175, 2004.

[96] Alain J. Martin. Synthesis of asynchronous VLSI circuits. Technical Report CS-TR-93-28, Computer Science Department, California Institute of Technology, 1993.

[97] Stephen McCamant and Michael D. Ernst. Predicting problems caused by component upgrades. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 287–296, 2003.

[98] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1992.

[99] Kenneth L. McMillan. Applying SAT methods in unbounded symbolic model checking. In E. Brinksma and K. G. Larsen, editors, *Proc. 14th International Conference on Computer-Aided Verification (CAV)*, LNCS 2404, pages 250–264. Springer-Verlag, July 2002.

[100] Kenneth L. McMillan. An interpolating theorem prover. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, LNCS 2988, pages 16–30, 2004.

[101] Eric Mercer. *Correctness and Reduction in Timed Circuit Analysis*. PhD thesis, University of Utah, 2002.

[102] J. B. Møller. Simplifying fixpoint computations in verification of real-time systems. In *Proc. Second Workshop on Real-Time Tools*, Copenhagen, Denmark, 1 August 2002.

[103] Moscow ML. Available at `http://www.dina.dk/ ~sestoft/mosml.html`.

[104] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *38th Design Automation Conference (DAC '01)*, pages 530–535, June 2001.

[105] Chris J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Stanford University, 1995.

[106] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, January 1997.

[107] Radu Negulescu. A technique for finding and verifying speed-dependences in gate circuits. In *Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)*, pages 189–198, 1997.

[108] Radu Negulescu and Ad Peeters. Verification of speed-dependences in single-rail handshake circuits. In *4th International Symposium on Asynchronous Circuits and Systems (ASYNC'98)*, pages 159–171. IEEE Press, 1998.

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

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

[111] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*. Wiley-Interscience, New York, 1988.

[112] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization*, chapter I.4: Polyhedral Theory. Wiley-Interscience, New York, 1988.

[113] T. Newsham. Format string attacks. `www.securityfocus.com/guest/3342`.

[114] P. Niebert, M. Mahfoudh, E. Asarin, M. Bozga, N. Jain, and O. Maler. Verification of timed automata via satisfiability checking. In *Proc. FTRTFT'02*, volume 2469 of *Lecture Notes in Computer Science*, pages 225–244, 2002.

[115] Steven M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford University, 1993.

[116] Mika Nyström and Alain Martin. *Asynchronous Pulse Logic*. Kluwer Academic Publishers, 2002.

[117] National Institute of Standards and Technology. The economics impacts of inadequate infrastructure for software testing. Available at `http://www.nist.gov/director/prog-ofc/report02-3.pdf`, May 2002. Prepared by RTI: Health, Social, and Economics Research.

[118] Christos H. Papadimitriou. On the complexity of integer programming. *Journal of the ACM*, 28(4):765–768, 1981.

[119] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[120] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*, chapter 13. Prentice-Hall, 1982.

[121] Marco A. Peña, Jordi Cortadella, Alex Kondratyev, and Enric Pastor. Formal verification of safety properties in timed circuits. In *6th International Symposium on Asynchronous Circuits and Systems (ASYNC'00)*, pages 2–11. IEEE Press, 2000.

[122] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small-domain instantiations. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 455–469. Springer-Verlag, July 1999.

[123] A. Pnueli, M. Siegel, and O. Shtrichman. The code validation tool (CVT) - automatic verification of a compilation process. *Software Tools for Technology Transfer*, 2(2):192–201, 1998.

[124] Vaughan Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, 1977. Cambridge, MA.

[125] M. Preßburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. *Comptes-rendus du Premier Congrès des Mathématiciens des Pays Slaves*, 395:92–101, 1929.

[126] Steven Prestwich. Local search on SAT-encoded colouring problems. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *Lecture Notes in Computer Science*, May 2003.

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

[128] C. Ramchandani. Analysis of asynchronous concurrent systems by timed Petri nets. Technical Report Project MAC Tech. Rep. 120, Massachussets Institute of Technology, February 1974.

[129] J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[130] Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[131] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, 1986.

[132] SecurityFocus. Qualcomm qpopper vulnerability. `www.securityfocus.com/advisories/2271`.

[133] S. A. Seshia, S. K. Lahiri, and R. E. Bryant. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In *40th Design Automation Conference (DAC '03)*, pages 425–430, June 2003.

[134] Sanjit A. Seshia and Randal E. Bryant. Unbounded, fully symbolic model checking of timed automata using Boolean methods. In *Proc. Intl. Conference on Computer-Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 154–166, July 2003.

[135] Sanjit A. Seshia and Randal E. Bryant. Deciding quantifier-free Presburger formulas using parameterized solution bounds. In *19th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 100–109, July 2004. Invited for publication in the journal *Logical Methods in Computer Science*.

[136] Sanjit A. Seshia, Randal E. Bryant, and Kenneth S. Stevens. Modeling and verifying circuits using generalized relative timing. In *11th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, March 2005. To appear.

[137] Sanjit A. Seshia, Shuvendu K. Lahiri, and Randal E. Bryant. *A User's Guide to UCLID Version 2.0*, May 2005.

[138] Sanjit A. Seshia, K. Subramani, and Randal E. Bryant. On solving Boolean combinations of generalized 2SAT constraints. Technical Report CMU-CS-04-179, Carnegie Mellon University, 2004.

[139] Natarajan Shankar and Harald Rueß. Combining Shostak theories. In Sophie Tison, editor, *Proc. Rewriting Techniques and Applications*, LNCS 2378, pages 1–18. Springer-Verlag, July 2002.

[140] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Automated detection of format-string vulnerabilities using type qualifiers. In *Proc. $10^{th}$ Security Symp.*, pages 201–220. USENIX, 2001.

[141] R. E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, 1984.

[142] Siege SAT Solver. Available at `http://www.cs.sfu.ca/~loryan/personal`.

[143] K. S. Stevens, S. Rotem, R. Ginosar, P. Beerel, C. J. Myers, K. Y. Yun, R. Koi, C. Dike, and M. Roncken. An asynchronous instruction length decoder. *IEEE Journal of Solid-State Circuits*, 36(2):217–228, 2001.

[144] Kenneth S. Stevens. *Practical Verification and Synthesis of Low Latency Asynchronous Systems*. PhD thesis, University of Calgary, Calgary, Alberta, September 1994.

[145] Kenneth S. Stevens, Ran Ginosar, and Shai Rotem. Relative timing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(1):129–140, February 2003.

[146] O. Strichman. On solving Presburger and linear arithmetic with SAT. In *Formal Methods in Computer-Aided Design (FMCAD '02)*, LNCS 2517, pages 160–170. Springer-Verlag, November 2002.

[147] O. Strichman. Optimizations in decision procedures for propositional linear inequalities. Technical Report CMU-CS-02-133, Carnegie Mellon University, 2002.

[148] O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding separation formulas with SAT. In E. Brinksma and K. G. Larsen, editors, *Proc. 14th Intl. Conference on Computer-Aided Verification (CAV'02)*, LNCS 2404, pages 209–222. Springer-Verlag, July 2002.

[149] K. Subramani. On deciding the non-emptiness of 2SAT polytopes with respect to first order queries. *Mathematical Logic Quarterly*, 50(3):281–292, 2004.

[150] Ivan Sutherland and Scott Fairbanks. GasP: A minimal FIFO control. In *7th International Symposium on Asynchronous Circuits and Systems*, pages 46–53. IEEE Press, March 2001.

[151] SVM-Light: Support Vector Machine Package. Available at `http://svmlight.joachims.org/`.

[152] Synopsys. *PrimeTime User Guide: Advanced Timing Analysis*, Version T-2002.09.

[153] Muralidhar Talupur, Nishant Sinha, Ofer Strichman, and Amir Pnueli. Range allocation for separation logic. In *Proc. Computer-Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 148–161, 2004.

[154] A. Thuemmel. Analysis of format string bugs. *Manuscript*, 2001. `http://downloads.securityfocus.com/library/format-bug-analysis.pdf`.

[155] ATACS Verification Tool. Available at `http://www.async.ece.utah.edu/tools/`.

[156] UCLID Verification System. Available at `http://www.cs.cmu.edu/~uclid`.

[157] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag, New York, 1995.

[158] M. N. Velev and R. E. Bryant. Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions. In *Correct Hardware Design and Verification Methods (CHARME '99)*, pages 37–53, September 1999.

[159] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea, and Seth Copen Goldstein. C to asynchronous dataflow circuits: An end-to-end toolflow. In *International Workshop on Logic synthesis (IWLS)*, pages 501–508, June 2004.

[160] J. von zur Gathen and M. Sieveking. A bound on solutions of linear integer equalities and inequalities. *Proceedings of the American Mathematical Society*, 72(1):155–158, October 1978.

[161] Farn Wang. Efficient verification of timed automata with BDD-like data-structures. In *Proc. 4th Intl. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 189–205, New York, January 2003.

[162] Farn Wang. Formal verification of timed systems: A survey and perspective. *Proceedings of the IEEE*, 92(8):1283–1305, August 2004.

[163] Ryan Williams, Carla Gomes, and Bart Selman. Backdoors to typical case complexity. In *Proceedings of the International Joint Conference in Artificial Intelligence (IJCAI)*, pages 1173–1178, 2003.

[164] Wisconsin Safety Analyzer Project. `http://www.cs.wisc.edu/wisa`.

[165] Steven A. Wolfman and Daniel S. Weld. The LPSAT engine and its application to resource planning. In *Proceedings of the International Joint Conference in Artificial Intelligence (IJCAI)*, pages 310–317, 1999.

[166] Pierre Wolper and Bernard Boigelot. An automata-theoretic approach to Presburger arithmetic constraints. In *Proc. Static Analysis Symposium*, LNCS 983, pages 21–32, September 1995.

[167] Tomohiro Yoneda and Hiroshi Ryu. Timed trace theoretic verification using partial order reduction. In *5th International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 108–124. IEEE Press, 1999.

[168] Sergio Yovine. Kronos: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1/2):123–133, October 1997.

[169] zChaff Boolean Satisfiability Solver. Available at `http://ee.princeton.edu/~chaff/zchaff.php`.

[170] Hao Zheng, Chris J. Myers, David Walter, Scott Little, and Tomohiro Yoneda. Verification of timed circuits with failure directed abstractions. In *21st International Conference on Computer Design (ICCD)*, pages 28–35. IEEE Press, 2003.