Efficient SMT Solving for Hardware Model Checking

by

Hyondeuk Kim

B.E., Ajou University, Korea, 2002

M.S., University of Colorado at Boulder, USA, 2005

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Electrical and Computer Engineering

2010

Kim, Hyondeuk (Ph.D., Electrical and Computer Engineering)

Efficient SMT Solving for Hardware Model Checking

Thesis directed by Professor Fabio Somenzi

The Satisfiability Modulo Theories (SMT) problem is a decision problem for the satisfiability of first-order formulas with background theories. In the last few years, decision procedures for SMT have been studied intensively, and they are applied successfully to hardware and software verification, compiler optimization, scheduling, and other design automation areas. In particular, during our study, we have found that they are also applicable to constrained random simulation.

SMT solvers have been effectively applied to software verification with predicate abstraction [BMMR01, LNO06] and bounded model checking [GG08, AMP06]. Only to a lesser extent, they have been applied to hardware verification. In today's hardware designs, bit-level and word-level operations are often tightly intermingled. On some designs, a bit-level model checker may perform better than a word-level model checker or vice versa.

In my dissertation, we study several efficient SMT solving techniques that can be applied to hardware model checking and constrained random simulation. In particular, we present a hybrid approach [KJS07a, KS06] for integer difference logic that combines finite instantiation method with Bellman-Ford algorithm. In addition, we present an efficient term-ITE conversion method [KSJ09] that improves SMT solving by word-level simplifications. Efficiency of these techniques have been shown in our SMT solver SatEEn that won the 1st places in **Integer Difference Logic** (IDL) and **Linear Integer Arithmetic Logic** (LIA) divisions of SMT Competition 2009.

In SMT-based model checking, an efficient encoding plays an important role along with the efficient SMT solving. For hardware model checking, we propose an SMT-based model checking system that consists of modeling and constraint solving components. The modeling component selectively decides the encoding method by analyzing the model, and the constraint solving component uses either **Linear Integer Arithmetic Logic** (LIA) or **Bit-Vector** (BV) solver for the encoding. On the other hand, hardware modeling is nontrivial since the behavior of hardware is described with the detailed event semantics of Standard Verilog [IEE06]; hence we define a subset of Verilog with restrictions that guarantee behavioral equivalence between verification condition and simulation of synchronous hardware. The restrictions lead to a concise verification condition and allow controlled nondeterminism that can be easily eliminated for synthesis. In addition, we propose an encoding method that improves SMT solving by maximizing the use of word-level information. For constrained random simulation, we propose to use word-level simplification [KJR$^+$08] that reduces the bit-width of each variable in the design.

Acknowledgements

I am extremely grateful to my advisor Professor Fabio Somenzi for his guidance, encouragement and patience during my graduate studies. I started to learn how to do research by conquering several peaks and taking extra credits in the Rocky Mountains with him. Whenever I faced an obstacle during my research, he has always guided me to the right direction to find a solution to the problem. I also would like to thank Professor Clark Barrett, Professor Aaron Bradley, Professor Sriram Sankaranarayanan and Professor Manish Vachharajani for kindly serving as my thesis committee and providing valuable suggestions that improved the quality of my thesis.

During my study, I had great opportunities to work on real-world problems in formal verification area. I would like to thank Dr. Chao Wang and Dr. Aarti Gupta for giving such an opportunity to work on hardware model checking problems in NEC Laboratories. I also would like to thank Dr. Kavita Ravi, Dr. HoonSang Jin, and Dr. Robert P. Kurshan for giving an opportunity to apply my work to constrained random simulation in Cadence Design Systems.

The members of our research group made my work more interesting and fun through many discussions. I would like to thank HoonSang Jin, Hyojung Han, and Saqib Sohail for sharing interesting ideas about my work. Many thanks to HoonSang Jin and Hyojung Han for enjoying the research and hiking together and encouraging me to finish my thesis.

Finally, I would like to thank my parents for their love and support. Without their sacrifice, I would not be able to finish this thesis. I dedicate this thesis to them.

Contents

Chapter

1 Introduction 1

1.1 Background 1

1.2 Thesis Contribution 3

1.3 Thesis Organization 5

2 Preliminaries 7

2.1 Satisfiability Modulo Theories 7

2.1.1 Bit-Vector Logic 8

2.1.2 Linear Integer Arithmetic Logic 9

2.1.3 $\mathsf{BV} \cup \mathsf{LIA}$ Logic 10

2.1.4 Integer Difference Logic 10

2.1.5 DPLL(T) 11

2.2 Model Checking 13

2.3 Hardware Description Language 16

3 Finite Instantiation for Integer Difference Logic 17

3.1 Introduction 17

3.2 Preliminaries 18

3.3 Minimizing the Abstract Models 20

3.4 Bounds on Solutions 22

3.4.1 Bound Computation . . . 22
3.4.2 Inconsistency Check for Partial Interpretations . . . 23
3.4.3 Consistency Check for Abstract Models . . . 24
3.5 Algorithm . . . 25
3.5.1 The Theory Solver . . . 26
3.6 Related Work . . . 32
3.7 Experimental Results . . . 33
3.8 Conclusions . . . 36

4 Efficient Term-ITE Conversion 38
4.1 Preliminaries . . . 40
4.2 Term-ITE Conversion . . . 41
4.2.1 Two Methods for Term-ITE Conversion . . . 41
4.2.2 Term-ITE Conversion with Cofactors . . . 43
4.3 Simple Preprocessing . . . 45
4.4 Algorithm . . . 46
4.5 Related Work . . . 50
4.6 Experimental Results . . . 50
4.7 Conclusions . . . 53

5 Avoiding Mismatches in Verification of Verilog Designs 54
5.1 Introduction . . . 54
5.2 Verification Conditions for Hardware . . . 55
5.3 Correctness . . . 60
5.4 Discussion . . . 67
5.5 Related Work . . . 69
5.6 Experimental Results . . . 70
5.7 Conclusions . . . 70

6 Selective SMT Encoding for Hardware Model Checking 72

6.1 Introduction 72

6.2 From Hardware Description to BV 73

6.3 SMT Encoding for Hardware Designs 75

6.3.1 LIA Encoding 75

6.3.2 SMT Encoding with Combined Theories (BV $\cup$ LIA) 78

6.4 Model Analysis 80

6.4.1 Analysis of Bit-Select Operations 80

6.4.2 Analysis of Bit-Wise Operation 81

6.4.3 Scoring System 82

6.4.4 Experimental Evaluation 82

6.5 Related Work 84

6.6 Conclusions 84

7 Application of Formal Word-Level Analysis to Constrained Random Simulation 86

7.1 Introduction 86

7.2 Constraint Solving in Simulation 87

7.3 **DomRed**: Technical Details 88

7.4 Experimental Results 89

8 Conclusions 91

8.1 Thesis Conclusions 91

8.2 Future Work 92

Bibliography 94

Appendix

A Tables for Comparison 101

Tables

Table

3.1 Number of Calls and Conflicts 37

4.1 Number of term-ITE reduction with simple preprocessing 51

5.1 Result of Vl2smt on Verilog designs 71

6.1 Comparison of using selective, SAT, BV, and LIA encodings on evaluation set 85

7.1 Comparison Table of without and with Bound Reduction 89

A.1 Comparison of using BV solvers and LIA solvers on Verilog design 102

A.2 Comparison of using BV solvers and LIA solvers (Bit-Blast) on training set of Verilog designs 103

A.3 Comparison of LIA encodings with and without value enumeration 104

A.4 Comparison of LIA encoding with Term-ITEs and LIA encoding with fresh variables 105

A.5 Comparison of LIA with Bit-Blast, pure LIA and BV $\cup$ LIA encoding 106

A.6 Comparison of using BV solvers and LIA solvers (Bit-Blast) on evaluation set of Verilog designs 106

Figures

Figure

2.1 DPLL(T) algorithm 11
2.2 Theory solver algorithm 12
2.3 4x4 Shidoku problem 15

3.1 Theory solver algorithm 27
3.2 Theory solver algorithm (continued) 28
3.3 SCC without any negative cycle 30
3.4 BARCELOGICTOOLS vs. Sateen on QF_IDL 34
3.5 YICES vs. Sateen on QF_IDL 34
3.6 MATHSAT vs. Sateen on QF_IDL 35
3.7 BARCELOGICTOOLS vs. Sateen on Job Shop Scheduling and Queen suites 36
3.8 YICES vs. Sateen on Job Shop Scheduling and Queen suites 36
3.9 MATHSAT vs. Sateen on Job Shop Scheduling and Queen suites 37
3.10 Sateen with splitting disequalities vs. Sateen on Job Shop Scheduling and Queen suites . . . 37

4.1 Verification condition $\mathcal{F}$ with term-ITEs 39
4.2 Term-ITE conversion 42
4.3 Term-ITE conversion with cofactor 43
4.4 Term-ITE conversion with simple check 45
4.5 Term-ITE conversion algorithm 47

4.6 Term-ITE conversion algorithm 48

4.7 Term-ITE conversion 49

4.8 Z3 vs. Sateen on QF_LIA 52

4.9 MATHSAT vs. Sateen on QF_LIA 52

4.10 YICES vs. Sateen on QF_LIA 52

4.11 Z3 WITH PREPROCESS vs. Z3 on QF_LIA 52

4.12 MATHSAT WITH PREPROCESS vs. MATHSAT on QF_LIA 52

4.13 YICES WITH PREPROCESS vs. YICES on QF_LIA 52

4.14 SATEEN vs. Sateen without Theory-Simp on QF_LIA 53

4.15 SATEEN vs. Sateen with naive approach on QF_LIA 53

5.1 Conversion from **MSV** description to intermediate form 57

5.2 Verilog code for a test bench 61

5.3 Nondeterministic behavior of Verilog simulation (a) 63

5.4 Nondeterministic behavior of Verilog simulation (b) 64

5.5 Deterministic Verilog design 69

6.1 BV vs. LIA 73

6.2 Conversion from HDL to SSA form 74

6.3 VAL ENUM vs. No Val Enum 78

6.4 TERM-ITE vs. Fresh Variable 78

6.5 LIA WITH BIT-BLAST vs. LIA 81

6.6 BV vs. LIA for Training Set 83

6.7 BV vs. LIA for Evaluation Set 83

Chapter 1

Introduction

1.1 Background

The Satisfiability Modulo Theories (SMT) problem has been the subject of intense scrutiny in the last few years. On the one hand, emerging applications like model checking of infinite state systems rely on such decision procedures for tasks like predicate abstraction [BMMR01]. On the other hand, algorithmic advances have significantly increased the range of problems that can be tackled, and hence have stimulated interest.

Recently, a dramatic performance increase in propositional satisfiability (SAT) solvers has led to the development of decision procedures that rely on the **propositional abstraction** [BDS02] of formulae from more expressive logics like the logic of **Linear Arithmetic** (LA) constraints, Presburger arithmetic, the logic of array, the logic of bit-vector, or the logic of equality and uninterpreted function symbols (EUF). In the propositional abstraction of a formula, atomic formulae of the specific theory (e.g., $x - y \leq 5$ or $f(x) = f(y)$, where f is an uninterpreted function symbol) is replaced with fresh propositional variables. Each model of the abstraction maps to a conjunction of literals in the original formula that can be checked for consistency with theory-specific procedures. If such a procedure establishes consistency, then the given formula is satisfiable and the enumeration terminates. Otherwise, from the proof of inconsistency a refinement of the propositional abstraction is extracted and the search is resumed.

There are several ways to combine the propositional reasoning engine with the theory-specific procedures. One broad classification is into **lazy** and **eager** approaches. A lazy solver produces an initial propositional approximation that is concise and possibly quite coarse; it relies on refinements during the enumeration of solutions. By contrast, an eager solver adds constraints to the initial propositional abstraction that embody known relationships among the literals. An example is given by the constraints that encode transitivity of equality. The most effective solvers often adopt elements of both approaches and tailor their strategies to the theory (theories) at hand.

Despite the recent progress in SMT solving, several challenges still remain to be solved. The challenges in SMT can be broadly divided into two major parts: enhancement of SMT solving and applicability of the solver. As mentioned in [NORCR07], one of the big challenges in SMT solving is to obtain hybrid procedures that combine the benefits of both lazy and eager approaches. Depending on the problem, one may perform well, and the other may not. One simple way to combine these two approaches is that we analyze the problem features and apply adaptively one of the two approaches. For the adaptive method, an intelligent problem analysis method will be required.

Another challenge in SMT solving is on the simplification of the problem. In practice, SMT instances contain a lot of redundancies and retaining them in SMT solving resulted in poor performance of the solver. Recently, SAT preprocessing techniques [EB05] have been intensively studied, and the techniques are widely used in most SAT solvers. These techniques are also used in SMT, but they have some limitations since the theories are not considered for the simplification. In LA logic, the solvers are required to handle the infinite precision numbers for the soundness of the results. As a result, the solvers use infinite precision libraries such as GMP [GMP]; however, the cost of using the library is expensive due to the complex computations with the cumbersome numbers. Finding the practical way to avoid using the infinite precision library, or to lessen the burden for the library is a big challenge in LA logic.

Handling quantifiers and dealing with the combination of logics in SMT still remain to be as interesting research topics. In real world problems, one is often required to use quantifiers or multiple logics to describe the problem. Although there are several works [BCF$^+$06, GBT07, GdM09] on quantifiers and the combination of logics, only a few SMT solvers support these features; thus there are still more room for the improvement.

Although SMT solvers have been widely used in software verification, the challenge still remains in the hardware verification. In today's hardware designs, bit-level and word-level operations are often tightly intermingled to describe the model behavior. Recently, bit-vector solvers [BB09, Bru08] have been applied to hardware verification; however the most bit-vector solvers are based on the eager approach that encodes the bit-vector variables and the operations into SAT, and only utilize partial word-level information.

Among these challenges in SMT, we study an efficient SMT solver that combines lazy and eager approaches, and adopts word-level preprocessing technique to simplify the problem. As an application of SMT solver, we study an effective SMT-based model checking for hardware verification, and a formal word-level analysis to constrained random simulation.

1.2 Thesis Contribution

In this section, we describe the contributions of my thesis to SMT solving and its application.

- Finite Instantiations for Integer Difference Logic [KS06, KJS07b]: We describe a theory solver for **Integer Difference Logic** (IDL) that is effective when the formula to be decided contains equality and disequality (negated equality) constraints so that the decision problem partakes of the nature of the pigeonhole problem. Atomic formulae in IDL constrain the difference between the values of pairs of integer variables. This logic finds extensive application to problems involving timing and scheduling constraints, resource allocation, and program analysis. IDL is closely related to **Real Difference Logic** (RDL), to the point that a decision procedure for the latter based on propositional abstraction also works for the former, as long as the coefficients are integers. It is sufficient to rewrite each **equality** constraint (of the form $x - y = n$) as the conjunction of two inequalities. However, if an equality constraint is negated, then the conjunction turns into a disjunction, which requires case splitting in the enumeration of the propositional solutions. In contrast, we propose an approach that does not decompose equalities and their negations; rather, it converts the problem of checking satisfiability of a conjunction of arithmetic atomic formulae into a set of propositional satisfiability checks—whose cardinality is bounded by the number of strongly connected components (SCC) of a suitable constraint graph. The conversion to propositional satisfiability that we have proposed is based on the ability to bound the values of the integer variables that appear in the formula. While in general such bounds do not exist, we have shown that to decide satisfiability of a set of constraints whose graph is a single SCC, it is sufficient to consider a subset of the solutions for which bounds are easily established. We also showed how the general case can be efficiently solved given solutions for the individual SCCs of the constraint graph. Experimental study shows that our new approach greatly improves the efficiency of our decision procedure for problem instances in which disequalities play a significant role, and makes it very competitive with respect to state-of-the-art tools.

- Efficient Term-ITE Conversion for SMT [KSJ09]: We describe how *term-if-then-else* (*term-ITE*) is handled in SMT. Term-ITEs allow one to conveniently express verification conditions; hence, they are very common in practice. However, the theory provers of SMT solvers are usually designed to work on conjunctions of literals; therefore, the input formulae are rewritten so as to eliminate term-ITEs. The challenge in rewriting is to avoid introducing too many new variables, while avoiding as often as possible the exponential explosion that is frequent when a naive approach is applied. We proposed a solution that is based on the computation of cofactors and theory propagation, and the experimental shows that the conversion method often produces orders-of-magnitude speedups in several SMT solvers for LIA problems.

- Avoiding Mismatches in Verification of Verilog Designs: We present a subset of Verilog with restrictions that guarantee behavioral equivalence between verification condition and simulation of synchronous hardware. The restrictions lead to a concise verification condition and allow controlled nondeterminism that can be easily eliminated for synthesis. Under a cycle-based simulation environment, we prove that every execution trace that may be produced by a standard-compliant simulator for synchronous hardware is captured in the verification condition, and vice versa.

- Selective SMT Encoding for Hardware Model Checking: We present a selective SMT encoding for hardware model checking. In particular, we introduce a model analysis method that considers each bit-vector operation in the design and selects the encoding based on the analysis. In addition, we present some enhancements to SMT encoding for hardware designs. Our experiments show that our approach selects the right encoding for most of the hardware designs and improves the efficiency of hardware model checking.

- Application of Formal Word-Level Analysis to Constrained Random Simulation [KJR$^+$08]: We have presented a new application of using SMT to constrained random simulation. In the constrained random simulation, the word-level analysis with SMT solver on word-level model enables the bit-level solver to avoid size explosion problem. Our main objective is to give bound reduction to the variables that are used in bit encoding. For bound computation, we use Bellman-Ford algorithm for the **difference** constraints and use simple coefficients checking for other linear arithmetic constraints. We can also detect a overconstraint from the set of **difference** constraints using Bellman-Ford algorithm. From the experiment, we found that our simple and fast algorithm can give huge amount of reduction to the variables in the real problem.

1.3 Thesis Organization

The rest of this thesis is organized as follows.

Chapter 2 introduces Satisfiability Modulo Theories and model checking. In addition, we review the Verilog hardware description language.

Chapter 3 presents an approach to solve IDL problem that contains many disequality constraints. We describe a theory solver that employs clique generation and finite instantiations to check the feasibility of a conjunction of inequality and disequality constraints. We present a bound computation algorithm that computes the bounds of integer variables in the constraints.

Chapter 4 presents an efficient term-ITE conversion method for SMT. We present a term-ITE conversion method that is based on cofactoring and theory simplification. We show the effectiveness of our approach by applying the method to LIA instances that make extensive use of the term-ITE operator.

Chapter 5 we present a subset of Verilog with restrictionsw that guarantee behavioral equivalence between verification condition and simulation of synchronous hardware. We show that the restrictions lead to a concise verification condition and allow controlled nondeterminism that can be easily eliminated for synthesis.

In chapter 6, we present a selective SMT encoding method for hardware model checking that predicts the encoding for a hardware design based on model analysis method. We describe the model analysis method that considers several characteristics of the design. We also present several enhanced encoding techniques for LIA solvers. We show the experimental evaluation to show the effectiveness of the approach.

In chapter 7, we presents a word-level pre-processor, **DomRed**, that simplifies the constraints in constrained random simulation. A bound reduction algorithm is presented that reduces the bound of the variables that are used in bit-encoding.

In chapter 8, conclusions and some future research directions are presented.

Chapter 2

Preliminaries

In this chapter, we introduce Satisfiability Modulo Theories (SMT) solving and model checking. In addition, we review the Verilog hardware description language (HDL) that is commonly used in verification of hardware.

2.1 Satisfiability Modulo Theories

The Satisfiability Modulo Theories (SMT) problem is a decision problem that decides the satisfiability of first-order formulas with background theories. SMT solvers find increasing applications in areas like formal verification in which one needs to reason about complex Boolean combinations of numerical constraints. The most common approach to this problem leverages the efficiency of modern propositional satisfiability solvers that work on a propositional abstraction of the given formula. At the same time, they interact with theory solvers, which check conjunctions of literals for consistency and learn consequences (new lemmas) from them. This approach has come to be known as DPLL(T) [NO05].

Recently, word-level model checking [Bje09, Joh01, CKZ96] has received growing attention. In particular, SMT solvers have been effectively applied to software verification with predicate abstraction [BMMR01, LNO06] and bounded model checking [GG08, AMP06]. Only to a lesser extent, they have been applied to hardware verification. The most natural SMT encodings for hardware description are bit-vector (BV) and linear integer arithmetic (LIA) encodings. LIA encoding for RTL constructs is presented in [BBC$^+$06], where control variables are encoded as Boolean variables and datapath variables as integer variables. In [Bru08], the author presents a bit-vector (BV) solver with a layered approach for RTL design verification.

In this section, we recall the definitions of the logics BV, LIA, BV $\cup$ LIA, and IDL which we use to encode hardware. In addition, we review the DPLL(T) framework and discuss its algorithm.

2.1.1 Bit-Vector Logic

Let $V_B(n)$ for $n \in \mathbb{Z}^+$ be the set of BV variables whose domains are bit-vectors with n bits. Let V_P be the set of propositional variables. We assume that $i \neq j \rightarrow V_B(i) \cap V_B(j) = \emptyset$. Let $T_B(n)$ be a set of BV terms whose values are bit-vectors with n bits. The formulae in BV logic are inductively defined as follows.

- If $c \in \mathbb{N}$ and $c < 2^n$, then $c[n] \in T_B(n)$.
- If $x \in V_B(n)$, then $x[n] \in T_B(n)$.
- If $x \in V_B(n)$ and $0 \leq j \leq i < n$, then $x[i : j] \in T_B(i - j + 1)$, and if $t[n] \in T_B(n)$, then $\sim t[n] \in T_B(n)$. ($\sim$ is the bit-wise negation operator.)
- If $t_1[n], t_2[n] \in T_B(n)$, and $\circ$ is an arithmetic or bit-wise operator in $\{+, -, \cdot, /, \%, \&, |\}$, then $t_1[n] \circ t_2[n] \in T_B(n)$.
- If $t_1[i] \in T_B(i)$ and $t_2[j] \in T_B(j)$, then $concat(t_1[i], t_2[j]) \in T_B(i + j)$.
- A propositional variable $a \in V_P$ is a formula.
- If $t_1[n], t_2[n] \in T_B(n)$, and $\diamond$ is a relational operator in $\{=, \neq, <, \leq, >, \geq\}$, then $t_1[n] \diamond t_2[n]$ is a formula.
- If f_1, f_2, and f_3 are formulae, then $\neg f_1$, $f_1 \wedge f_2$, $f_1 \vee f_2$ and *ite*(f_1, f_2, f_3) are formulae, and if $t_1[n], t_2[n] \in T_B(n)$ and f is a formula, then *tite* $(f, t_1[n], t_2[n]) \in T_B(n)$.

Further formulae can be defined as abbreviations. For instance, $x[n] \ll k$, a left shift of $x[n]$ by a constant k, is defined as $concat(\ x[n - k - 1 : 0], 0[k])$. An **atomic formula** is one of the form $t_1[n] \diamond t_2[n]$, where $\diamond$ is a relational operator. The semantics are defined in the usual way; in particular, arithmetic is modular, $x[i : j]$ is the subfield of $x[n]$ comprising the bits from i to j included, $concat(t_1[i], t_2[j])$ concatenates $t_1[i]$ and $t_2[j]$, and *ite*(f_1, f_2, f_3) is equivalent to $(f_1 \wedge f_2) \vee (\neg f_1 \wedge f_3)$. In addition, the **term if-then-else** (**tite**) operator is defined by the equivalence, for all formulae f and g and for all terms $t_1[n]$ and $t_2[n]$, of $f($*tite*$(g, t_1[n], t_2[n]))$ and *ite*$(g, f(t_1[n]), f(t_2[n]))$.

For $A, B, C, D, E \in V_B(2)$, (2.1) is a BV formula.

$$(C[2] = A[2] \ \& \ B[2]) \wedge (D[2] = C[2] + E[2]) \ . \tag{2.1}$$

2.1.2 Linear Integer Arithmetic Logic

Let V_Z be the set of integer-valued variables. The formulae in LIA logic are inductively defined as follows.

- An integer number $c \in \mathbb{Z}$ is a (constant) LIA term, and a variable $x \in V_Z$ is an LIA term.
- A variable $x \in V_Z$ is an LIA term, and the product $c \cdot x$ of an integer number $c \in \mathbb{Z}$ and a variable $x \in V_Z$ is an LIA term.
- If t_1 and t_2 are LIA terms, so are $t_1 + t_2$ and $t_1 - t_2$.
- A propositional variable $a \in V_P$ is a formula.
- If t_1 and t_2 are LIA terms, and $\diamond$ is a relational operator in $\{=, \neq, <, \leq, >, \geq\}$, then $t_1 \diamond t_2$ is a formula.
- If f_1, f_2, and f_3 are formulae, then $\neg f_1$, $f_1 \wedge f_2$, $f_1 \vee f_2$ and *ite*(f_1, f_2, f_3) are formulae.
- If t_1 and t_2 are LIA terms, and f is a formula, then *tite*(f, t_1, t_2) is an LIA term.

For $A, B, C, D, E \in V_Z$, (2.2) is an LIA formula:

$$(C = A - B) \wedge (D = C + E) \ . \tag{2.2}$$

2.1.3 BV $\cup$ LIA Logic

Let R_B be a set of rules for BV logic and R_Z be a set of rules for LIA logic. The formulae in BV $\cup$ LIA are inductively defined as the largest set that satisfies the rules in $R_B \cup R_Z$.

For $A \in V_B(2)$ and $C \in V_Z$, (2.3) is a BV $\cup$ LIA formula:

$$C = \textit{tite}(A[1\!:\!1] = 1[1], 2, 0) + \textit{tite}(A[0\!:\!0] = 1[1], 1, 0) \ . \tag{2.3}$$

With the use of V_P in BV and LIA logics, a BV formula can be easily converted into a Boolean formula. The conversion is called bit-blasting in which a set of propositional variables replaces each bit-vector. Through bit-blasting, a BV $\cup$ LIA formula can be converted into an LIA formula, which is often decided more efficiently.

Given $A_0, A_1 \in V_P$ and $C \in V_Z$, Eq. (2.4) shows the LIA formula obtained from Eq. (2.3) by bit-blasting $A[2]$:

$$C = \textit{tite}(A_1, 2, 0) + \textit{tite}(A_0, 1, 0) \ . \tag{2.4}$$

2.1.4 Integer Difference Logic

We define inductively **Integer Difference Logic** (IDL) formulae as follows.

- A propositional variable $a \in V_P$ is a formula.
- $x - y \leq n$ and $x - y = n$ are formulae, for $x, y \in V_P$, $n \in Z$.
- If φ and ψ are formulae, so are $\varphi \wedge \psi$ and $\neg\varphi$.

The following abbreviations are also defined:

$$\begin{array}{rclcrcl} x - y < n & \doteq & x - y \leq n - 1 & \qquad & x - y \neq n & \doteq & \neg(x - y = n) \\ x = y & \doteq & (x - y = 0) & & x \neq y & \doteq & \neg(x = y) \ . \end{array}$$

In SMT, a literal is an atomic formula, or the negation of an atomic formula. A **clause** is the disjunction of a set of literals, and a formula in **conjunctive normal form** (CNF) is the conjunction of a set of clauses.

2.1.5 DPLL(T)

DPLL(T) architecture [GHN$^+$04, NO05] combines DPLL(X), the propositional reasoning engine, with the theory specific procedure. Given an SMT formula φ with a specific theory T, DPLL(T) computes a **propositional abstraction** φ^b of φ by replacing the atomic formulae of T with fresh propositional variables. A model for φ^b maps to a conjunction of literals in φ that is checked for consistency with the theory solver. If the model is consistent in T, φ is satisfiable and the enumeration of the model terminates. Otherwise, the theory solver returns the explanation of the inconsistency for the refinement of the propositional abstraction, and the search is resumed. Checking consistency of the partial interpretation enables the solver to detect the inconsistency earlier and learn so-called **theory consequences** in T that often improve the efficiency of the search.

```
DPLL_T () {
    while (ChooseNextAssignment () == FOUND)
        while ($\top$) {
            if (Deduce () == $CONFLICT$ || TheorySolver () == $CONFLICT$) {
                blevel = AnalyzeConflict ();
                if (blevel $<$ 0) return $UNSAT$;
                else Backtrack (blevel);
                continue;
            }
            if (TheoryConseq () == $\emptyset$) break;
        }
    return $SAT$;
}
```

Figure 2.1: DPLL(T) algorithm

The pseudo-code of DPLL(T) procedure is presented in Fig. 2.1. The algorithm is not much different from the David-Putnam-Logemann-Loveland (DPLL) procedure [DP60, DLL62]. It works as the DPLL procedure if the condition **TheorySolver () ==** $CONFLICT$ is removed and the condition **TheoryConseq () ==** $\emptyset$ is converted into $\top$. The DPLL(T) algorithm is applied to a **propositional abstraction** φ^b of φ where φ is an SMT formula in CNF. It maintains an **assignment stack** that records all the assignments currently in effect and an **assignment queue** that records the assignments that are not in effect yet. The procedure

ChooseNextAssignment checks if the queue is empty and selects an unassigned variable to make a decision on the value of the variable if it is empty. If no unassigned variable is selected, the algorithm returns SAT, which means φ is satisfiable. The newly assigned variable, if it exists, is entered into the queue, and its implications are added in the queue by the **Deduce** procedure. If **Deduce** does not cause a conflict, the procedure **TheorySolver** checks if the conjunction of the atomic formula is consistent or not. If either **Deduce** or **TheorySolver** returns $CONFLICT$, then **AnalyzeConflict** analyzes the reason of the conflict. The procedure **AnalyzeConflict** returns the backtracking level, and if it is less than zero, the algorithm terminates by giving the $UNSAT$ result for φ; otherwise, the procedure **Deduce** resumes in the backtracking level. If there is no conflict in both **Deduce** and **TheorySolver**, the algorithm checks if **TheorySolver** generated theory consequences. If theory consequences are generated, the algorithm continues with the while loop in line 3; otherwise, it continues with the while loop in line 2.

```
1   TheorySolver () {
2       foreach $l \in I^b$ {
3           if ($I \models_T \neg l$) {
4               $L_E$ = Explanation ($I$, $\neg l$);
5               $\varphi^b$ = $\varphi^b \wedge \neg L_E$;
6               return $CONFLICT$;
7           } else {
8               $I = I \cup l$;
9           }
10      }
11      foreach $l \in L \setminus I$ {
12          if ($I \models_T l$) $I^b = I^b \cup l$;
13      }
14      return $NULL$;
15  }
```

Figure 2.2: Theory solver algorithm

The procedure **TheorySolver** in Fig. 2.2 is called with a conjunction of literals in T whose corresponding propositional literals are true in a (partial) interpretation I^b of the propositional formula φ^b. It decides whether there is an interpretation to the variables in the atomic formula that satisfies the conjunction of all those literals. Let L be the set of all the literals in φ^b and I be the set of literals that is a (partial) interpretation of φ. The set I is initially empty, and the negation of $l \in I^b$ is checked with I for a theory consequence. A literal l is a theory consequence of I, denoted $I \models_T l$, if l is true in I. If $I \models_T \neg l$ is true, an explanation L_E of the theory consequence is generated, where L_E is a conjunction of literals. The negation of L_E is conjoined with φ^b to prevent this to be happen again. Since a conflict is found in I with l, the procedure returns with the $CONFLICT$ result. If $I \models_T \neg l$ is not true, the literal $l \in I^b$ is added to I. The procedure continues to check $I \models_T \neg l$ for each $l \in I^b$ until it finds a conflict or all the literals in I^b are added to I, a new (partial) interpretation of φ. With the new interpretation I, each $l \in L \setminus I^b$ is checked for a theory consequence to deduce more literals. The literal $l \in L \setminus I^b$ is added to I^b if $I \models_T l$. After checking all the theory consequences, the procedure returns with $NULL$.

2.2 Model Checking

Model checking [CE81, CGP99] is an algorithmic approach to verify the correctness properties of a finite state system automatically. Given a model M of a hardware or software system, the transition system of M is explored with a temporal property φ to check if the property holds in the model, denoted $M \models \varphi$. If the model does not meet the property, denoted $M \not\models \varphi$, the model checking algorithm provides a counterexample trace that demonstrates how the property can be violated.

Traditionally, explicit-state model checking [CE81] approach has been widely used, where the set of states and the transition relations are explicitly represented and the search algorithm explores the states to check if the state violates the property. Due to the explicit representation of the states, the method often suffers with the state explosion problem. As an alternative approach, symbolic model checking [McM94, BCCZ99] approach uses a Boolean formula to represent the set of states and the transition relations, where the Boolean formula is often represented with Binary Decision Diagrams (BDDs) [Bry86] or propositional satisfiability (SAT) [MMZ$^+$01, GN02, JS04]. Since BDDs are canonical representation, the BDD-based model checking may suffer with the size explosion problem; however, once the BDDs are built, the model checking problem can be solved efficiently. On the other hand, SAT- based model checking avoids the size explosion problem by not using the canonical representation. It converts the Boolean formula into a **Conjunctive Normal Form** (CNF) to be solved by propositional SAT solvers.

In SAT-based Bounded Model Checking (BMC), the transition relation of a model is unrolled k

times and conjoined with the set of initial states and the negation of the Linear Time Logic (LTL) property [WVS83, LP85]. The conjoined Boolean formula in CNF is solved by a propositional SAT solver and is satisfiable if there exists a counterexample of the length k to the property. In contrast to BDD-based model checking, SAT-based BMC suffers less to the size explosion problem and produces counterexamples of minimum length for all LTL properties.

Given a model $\mathcal{M}$, an LTL property ϕ, and a bound k, BMC constructs a Boolean formula denoted by $[\![\mathcal{M}, \neg\phi]\!]_k$, that is satisfiable if and only if there exists a counterexample of the length k to ϕ; $[\![\mathcal{M}, \neg\phi]\!]_k$ is defined as follows:

$$[\![\mathcal{M}, \neg\phi]\!]_k = I(s_0) \wedge \bigwedge_{0 \leq i < k} T(s_i, s_{i+1}) \wedge [\![\neg\phi]\!]_k \quad , \tag{2.5}$$

where I is the predicate describing the initial states, T is the transition relation, and $[\![\neg\phi]\!]_k$ expresses the satisfaction of $\neg\phi$ along that path defined by $s_0, s_1, \ldots s_k$.

In recent years, SMT-based model checking has received growing attention. In SMT-based BMC, a model is encoded into an SMT formula that is more concise and that preserves more structure of the model compare to the corresponding Boolean formula. In terms of efficiency of the solver, representing the model in SMT gives more flexibility to choose a suitable approach for the problem and often increases the deductive power of the solver. The following example compares SMT and SAT encodings for a **Shidoku problem** and shows the effectiveness of the SMT encoding.

Consider a **4×4 Shidoku problem** in Fig. 2.3. The objective of the **4×4 Shidoku problem** is to fill a **4×4** grid so that each column, each row, and each of the four **2×2** blocks contains the digits from 0 to 3 only one time each. It is well known that Shidoku problem can be encoded into either a SAT or an SMT problem. Suppose the values $x_0 = 0, x_1 = 1, x_2 = 3$ in the first column are given for the problem in Fig. 2.3. If we encode the problem into a SAT problem, we introduce Boolean variables x_i^1, x_i^0 for each integer variable x_i. The partial encoded SAT problem for $x_0 = 0, x_1 = 1, x_2 = 3$ and $x_0 \neq x_3, x_1 \neq x_3, x_2 \neq x_3$ is given below.

$$(x_3^1 \vee x_3^0) \wedge (x_3^1 \vee \neg x_3^0) \wedge (\neg x_3^1 \vee \neg x_3^0) \quad . \tag{2.6}$$

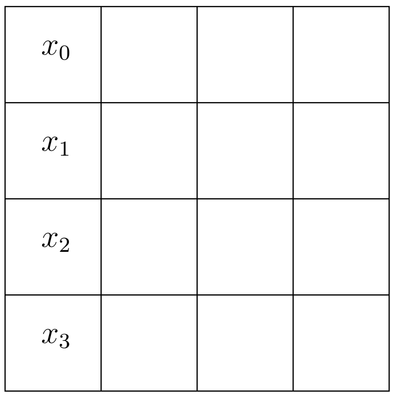


Figure 2.3: 4x4 Shidoku problem

As Eq. (2.6) shows, one of the Boolean variables in the clause should be decided to make assignments to the variables x_3^1 and x_3^0.

On the other hand, if the problem is encoded into a LIA formula, the formula is

$$\neg(x_3 = 0) \wedge \neg(x_3 = 1) \wedge \neg(x_3 = 3) \wedge (x_3 \geq 0) \wedge (x_3 \leq 3) \ \ . \tag{2.7}$$

The equalities in Eq. (2.7) can be converted into inequalities, and the converted formula is

$$((x_3 < 0) \vee (x_3 > 0)) \wedge ((x_3 < 1) \vee (x_3 > 1)) \wedge ((x_3 < 3) \vee (x_3 > 3)) \wedge (x_3 \geq 0) \wedge (x_3 \leq 3) \ \ . \tag{2.8}$$

By applying theory propagation with $(x_3 \geq 0)$ and $(x_3 \leq 3)$ in the unit clauses, Eq. (2.8) is simplified into

$$(x_3 > 1) \wedge (x_3 < 3) \ \ . \tag{2.9}$$

From Eq. (2.8), we can infer $(x_3 = 2)$.

The comparison of SAT and SMT encodings shows that SMT encoding introduces fewer number of variables and clauses for the problems that require word-level reasoning. As a result, the size of SMT encoding is much smaller than the size of SAT encoding. In addition, SMT encoding often gives more deductive power to the solver by considering the problem in word level.

2.3 Hardware Description Language

Verilog is a hardware description language (HDL) used to describe digital systems. Verilog HDL is the most commonly used language in verification, synthesis, and testing of hardware designs. Verilog describes a hardware design as a hierarchy of modules, where modules communicate each other through a set of declared inputs, outputs, and bidirectional ports. Each module contains net, variable, function, and task declarations, procedural and parallel blocks, and instances of other modules. A net can be of type **supply0**, **supply1**, **tri**, **triand**, **trior**, **tri0**, **tri1**, **wire**, **wand**, or **wor**. A variable can be of type **reg**, **integer**, **real time**, or **realtime**. A constant is an integer or real number; expressions are made of variables, constants, and operators, which are categorized into arithmetic, concatenation, reduction, bit-selection, shift, bit-wise, logical, conditional, and relational operators.

In Verilog, a blocking assignment (=) updates the target variable immediately, while the update of a nonblocking assignment ($\Leftarrow$) is deferred. A continuous assignment updates the target wire whenever the values of the operands in the right-hand side of the assignment is changed. A statement may be an assignment, an **if / else** conditional statement, a **case** statement, a looping statement, or a sequence of statements enclosed by the keywords **begin** and **end**.

A procedural block in Verilog can be either **inital** or **always**. An **initial** block is executed only once, and is used to describe the initial values and the updates of memory elements. On the other hand, an **always** block is executed repeatedly, and is used to describe combinational and sequential logics. The statements in a procedural block are executed sequentially in the given order, whereas the statements in a parallel block such as **fork-join** block are executed concurrently.

The statement in either a procedural or parallel block is controlled by explicit timing controls such as a delay control (**#d**) and an event control (**@ event_identifier**, **@ (event_expression)**, **@ (*)**, or **@ ***). The delay control specifies the time duration for executing a statement and the event control defers the execution of a statement until there is an occurrence of a declared event or value change on a net or variable.

Chapter 3

Finite Instantiation for Integer Difference Logic

3.1 Introduction

Decision procedures for fragments of first-order logic have been the subject of intense scrutiny in the last few years. On the one hand, emerging applications like model checking of infinite state systems rely on such decision procedures for tasks like predicate abstraction [BMMR01]. On the other hand, algorithmic advances have significantly increased the range of problems that can be tackled, and hence have stimulated interest.

In this chapter, we focus on **Integer Difference Logic** (IDL), in which arithmetic atomic formulae constrain the difference between the values of pairs of integer variables. This logic finds extensive application to problems involving timing and scheduling constraints, resource allocation, and program analysis. IDL is closely related to **Real Difference Logic** (RDL), to the point that a decision procedure for the latter based on propositional abstraction also works for the former, as long as the coefficients are integers. It is sufficient to rewrite each **equality** constraint (of the form $x - y = n$) as the conjunction of two inequalities. However, if an equality constraint is negated, then the conjunction turns into a disjunction, which requires case splitting in the enumeration of the propositional solutions. In contrast, we propose an approach that does not decompose equalities and their negations; rather, it converts the problem of checking satisfiability of a conjunction of arithmetic atomic formulae into a set of propositional satisfiability checks—whose cardinality is bounded by the number of strongly connected components (SCC) of a suitable constraint graph.

The conversion to propositional satisfiability that we propose is based on the ability to bound the values of the integer variables that appear in the formula. While in general such variables are not bounded, we show that to decide satisfiability of a set of constraints whose graph is a single SCC it is sufficient to consider a subset of the solutions for which bounds are easily established. We also show how the general case can be efficiently solved given solutions for the individual SCCs of the constraint graph. Experiments show that our new approach, which combines techniques typical of both the lazy and the eager approaches, greatly improves the efficiency of our decision procedure for problem instances in which disequalities play a significant role, and makes it very competitive with respect to state-of-the-art tools.

This chapter is organized as follows: Section 3.2 reviews background and introduces notation. Section 3.3 and Section 3.4 discuss the minimizing the abstract models and the bounds on solutions, while Sect. 3.5 deals with the implementation of our theory solver. After a survey of related work in Sect. 3.6, experiments are presented in Sect. 3.7, and conclusions are offered in Sect. 3.8.

3.2 Preliminaries

Propositional logic is the fragment of IDL obtained by omitting the rule that defines arithmetic atomic formulae. Efficient algorithms to decide the satisfiability of propositional logic formulae are based on the DPLL procedure [DP60, DLL62], and exploit techniques like clause recording, conflict analysis, nonchronological backtracking, and fast Boolean constraint propagation [MS96, MMZ$^+$01].

In recent times, decision procedures for IDL, and other fragments of quantifier-free first-order logic, have been based on the DPLL procedure as well. Given a set of propositional variables B such that $B \cap P = \emptyset$, one obtains a propositional formula φ^b from an IDL formula φ by replacing each arithmetic atomic subformula of φ with a distinct variable from B. The resulting formula φ^b is unsatisfiable only if φ is unsatisfiable. Each model of φ^b corresponds to a conjunction of literals of φ. Given a decision procedure for the conjunction of arithmetic atomic propositions in IDL (a **theory solver**), one therefore derives a complete decision procedure for IDL by enumerating the models of φ^b, extracting from each of them the corresponding conjunction of arithmetic atomic propositions and their negations, and checking these conjunctions for satisfiability using the theory solver. In the following, we refer to the conjunction of a set of arithmetic literals as a **set of IDL constraints**.

The theory solver rewrites the IDL constraints to be checked according to their form:

(1) $x - y \leq n$: unchanged;

(2) $x = y$: unchanged;

(3) $x - y = n$, with $n \neq 0$: split into $(x - y \leq n) \wedge (y - x \leq -n)$;

(4) $\neg(x - y \leq n)$: rewritten as $y - x \leq -n - 1$;

(5) $\neg(x = y)$: rewritten as $x \neq y$;

(6) $\neg(x - y = n)$, with $n \neq 0$: rewritten as $x - y \neq n$.

Constraints of type 1, 3, and 4 are **inequalities** (I). Constraints of type 2 are **equalities** (Q), and finally, constraints of type 5 and 6 are **disequalities** (D). Specifically, constraints of type 5 form the set $D_0 \subseteq D$. Let $C = I \cup Q \cup D$.

An edge integer-labeled directed graph is a triple $G = (V, E, \lambda)$, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, and $\lambda : E \rightarrow Z$ is an edge labeling function. A **strongly connected component** (SCC) of G is a maximal subgraph G' of G such that every two nodes of G' are connected by a path in G'. An SCC is **trivial** if it consists of one vertex and no arcs. The SCCs of G define a partition of V. The **SCC quotient graph** $\widehat{G} = (\widehat{V}, \widehat{E})$ of G is a directed acyclic graph with one vertex for each SCC of G and an edge $(A, B) \in \widehat{E}$ if and only if there exist $x \in A$ and $y \in B$ such that $(x, y) \in E$.

Given a distinguished source vertex $s \in V$, distances of all vertices from s are well defined provided there exists no **negative cycle** in G; that is, no cycle such that the sum of the labels on the edges along the cycle is negative. The Bellman-Ford algorithm [CLR90] reports negative cycles if they are present, and computes the distance $\delta(x)$ of each vertex in V from the source s otherwise. The **slack** of an edge $(x, y) \in E$ is given by $\sigma((x, y)) = \lambda((x, y)) - (\delta(y) - \delta(x))$. It is easy to see that for all $e \in E$, $\sigma(e) \geq 0$ and that $\sigma((x, y)) = 0$ if and only if (x, y) is on a shortest path from s to y in G. Distances and slacks obviously depend on the choice of source vertex.

Given a (finite) set I of inequality constraints (i.e., of the form $x - y \leq n$), their **constraint graph** $G = (V, E, \lambda)$ is a labeled directed graph defined as follows:

- $V \subseteq V_Z$ is the set of variables appearing in the constraints in I.

- There is an arc $(x, y) \in E$ with $\lambda((x, y)) = n$ if and only if there is a constraint $y - x \leq n$ in I.

It is well known [CLR90] that I is satisfiable if and only if G contains no negative cycle. In fact, adding both sides of the constraints forming a cycle of length w, one gets $0 \leq w$, which is not satisfiable when $w < 0$. If, on the other hand, no negative cycle exists in G, then one can find a model for I by solving a single-source shortest-path problem on an augmented graph G_a, obtained from G by adding a new reference vertex x_r and arcs labeled 0 from x_r to all the other vertices. Let $\delta(x)$ be the distance of $x \in V$ from x_r in G_a. Then δ is a model for I. It is also well known that, given a model of I, $\alpha : V \to Z$, and a constant, $c \in Z$, the interpretation $\alpha' : V \to Z$ defined by $\alpha'(x) = \alpha(x) + c$ is also a model of I, because $\alpha'(x) - \alpha'(y) = \alpha(x) - \alpha(y)$. This observation allows an easy encoding of **range constraints** in IDL. A set of constraints $\{l_i \leq x_i \leq u_i\}$ is translated to $\{x_i - y \leq u_i\} \cup \{y - x_i \leq -l_i\}$, where y is a fresh variable. The solution α obtained from the constraint graph is then translated so that $\alpha'(y) = 0$. One fresh variable suffices for multiple range constraints.

Since integer labels imply integer distances, if the right-hand sides of the constraints are integer-valued, and the constraints are satisfiable when the variables range over the real numbers, then an integer-valued solution is also guaranteed to exist. Loosely speaking, the satisfiability problem for **inequalities** is the same for IDL and real difference logic (RDL). Adding **equality** constraints to the inequalities does not change this state of affairs: Given a constraint $x - y = n$, one replaces x by $y + n$; if no immediate inconsistencies arise, one continues with the construction of the constraint graph. In contrast, if disequality constraints (i.e., negations of equalities) are allowed, an unsatisfiable conjunction of IDL constraints may be satisfiable when regarded as an RDL formula. An example is given by $\bigwedge_{1 \leq i \leq p}(1 \leq x_i \leq h) \wedge \bigwedge_{1 \leq i < j \leq p}(x_i \neq x_j)$, which exemplifies the pigeonhole principle.[1]

3.3 Minimizing the Abstract Models

Given the set of clauses φ^b and a complete model for them produced by the propositional reasoning engine, we consider now the problem of identifying a minimal (partial) model such that at least one literal

[1] This does not contradict what was observed in Sect. 3.1 because $x \neq y$ translates into $(x < y) \vee (y < x)$ for RDL, but translates into $(x \leq y - 1) \vee (y \leq x - 1)$ for IDL.

for each clause is true. The intent of finding such minimal model is twofold: to alleviate the task of the theory solver and to make the exploration of the models of φ^b more efficient. A greedy solution to our problem is easily obtained by considering each variable in turn and removing it from the model if no clause becomes unsatisfied as a result. We now describe how such a solution can be implemented efficiently in the context of the algorithm that enumerates the solutions to the propositional abstraction. That is, we show how we can take advantage of the information gathered by the propositional SAT solver to significantly speed up the choice of the minimal model.

Two observations from [RS04] provide the foundation for our method. The first is that no variable that received its value by implication (rather than decision) by the SAT solver can be removed from the model. This fact greatly reduces the number of variables that are candidates for removal. The second observation concerns the list of watched literals and assumes that only two literals are watched by the SAT solver [MMZ$^+$01]. It can then be shown that when a complete model is found, at least one watched literal in every clause is true. Therefore, when considering a variable for removal it is sufficient to check if it provides the only true literal in the clauses in which the satisfied literal of the variable is watched. The clauses in which that literal is not watched can be safely ignored. Moreover, conflict clauses recorded by the SAT solver do not need to be examined because they are known to be satisfied whenever the original clauses are satisfied.

When a clause in which the candidate literal is watched is examined, a substitute literal that is true is sought so as to maintain the invariant. If there is no substitute and the other watched literal is false, the candidate is rejected. On the other hand, if this process manages to empty the watched-literal list of the candidate (except possibly for conflict clauses), the candidate is removed from the model.

The effect of the minimization procedure is to alter the watched-literal lists of the solver. However, the enumeration process can resume from the modified lists without any adverse consequence. The algorithm that we have described runs in polynomial time, but only guarantees a minimal set of variables. Reduction from set covering shows that deciding whether a model of size k exists for a set of propositional clauses is NP-complete.

The order in which literals are considered for removal depends on the constraints they represent.

The check for consistency of a set of constraints tends to be easier if disequality constraints are the first candidates for elimination. They are followed by inequality constraints, and finally equality constraints, in that order.

3.4 Bounds on Solutions

In this section we show how bounds to the solutions of a set of constraints are computed and how those bounds are used in checking for consistency of (partial) interpretations of φ^b. Two cases must be distinguished depending on whether the interpretation to be checked is known to be a model of φ^b: If it is not known to be a model, a cheap check is applied, which can only report inconsistency. Otherwise, a more expensive, complete check is applied in addition, which decides consistency and computes a model of φ if it exists.

3.4.1 Bound Computation

It was recalled in Sect. 3.2 that from a solution α to a set of inequality constraints, one can derive a family of solutions $\{\alpha + c\}$. In general, however, not all solutions are obtained one from the other by **translation**. Consider the constraints $\{(x - y \leq 1), (y - x \leq 0)\}$. The two interpretations $\alpha_1(x) = 0$, $\alpha_1(y) = 0$ and $\alpha_2(x) = 1$, $\alpha_2(y) = 0$ satisfy the constraints, though there is no c such that $\alpha_1 = \alpha_2 + c$. Such solutions are called **independent**. In general, there may be several families of independent solutions, and therefore, multiple distinct solutions that assign a given value to a distinguished variable. The following result characterizes these sets of solutions and forms the basis for our treatment of disequality constraints in IDL.

Theorem 3.1. *Let I be a set of inequality constraints. Let $G = (V, E, \lambda)$ be the constraint graph associated to I. Suppose that G contains no negative cycle and consists of one SCC. Let δ_{ab} be the distance from a to b in G. For $x \in V$ and $n \in Z$, let S_x^n be the set of solutions $\alpha : V \to Z$ to I such that $\alpha(x) = n$. Then, for each vertex $y \in V$, there exist bounds $y_l = n - \delta_{yx}$ and $y_u = n + \delta_{xy}$, such that for every solution in S_x^n, $y_l \leq \alpha(y) \leq y_u$.*

Proof. By definition of SCC, every vertex in V is reachable from x in G; likewise, x is reachable from any vertex in G. Let δ_{xy} be the distance of y from x (the length of a shortest path). Such a distance is defined because there are no negative cycles in G. Adding both sides of all the constraints along the path yields $y - x \leq \delta_{xy}$. Therefore, for every solution $\alpha \in S_x^n$, it must be $\alpha(y) \leq n + \delta_{xy}$. Said otherwise, $y_u = n + \delta_{xy}$. For the lower bound, if δ_{yx} be the distance of x from y in G, then, for every solution $\alpha \in S_x^n$, it must be $\alpha(y) \geq n - \delta_{yx}$; that is, $y_l = n - \delta_{yx}$. $\square$

Satisfaction of disequalities is not affected by translation. Therefore, a set of constraints including both inequalities and disequalities is satisfiable if and only if there exists a solution α such that $\alpha(x) = n$. This allows us to limit the search to the set S_x^n for an arbitrarily chosen n. Theorem 3.1 asserts that solutions in this set are bounded. The way this result is exploited depends on whether the set of constraints corresponds to a model of φ^b. The next two subsections discuss the two cases.

3.4.2 Inconsistency Check for Partial Interpretations

Given a partial abstract interpretation that is not known to be a model of φ^b, we want to check the corresponding constraints for inconsistency to prune the search space (as in theory propagation) or to possibly avoid the more expensive check of Section 3.4.3. A set of constraints is assumed to be given along with ranges for every variable in them. It is also assumed that the graph has one SCC. If that is not the case, each SCC is checked in turn: The constraints are inconsistent if at least one SCC is inconsistent. Though the check described in the next section could be applied in this case, we are interested in a cheaper criterion.

The quick check for inconsistency is based on two observations: The first is that if all variables in the SCC have the same range, then the disequalities define a graph whose chromatic number must not exceed the size of the range for the constraints to be satisfiable. (The chromatic number is the least number of colors needed to assign different colors to adjacent vertices in the graph.) The second observation is that the chromatic number of a graph is bounded from below by the size of a clique of the graph and from above by the number of vertices. From these observations, it is easy to prove the following theorem.

Theorem 3.2. *Let D_0 be a set of disequality constraints of the form $x_i - x_j \neq 0$. Let $X = \{x_1, \ldots, x_n\}$ be* *the set of variables in D_0. Let $L = \{l_1, \ldots, l_n\} \in Z^n$ and $U = \{u_1, \ldots, u_n\} \in Z^n$ be the bounds on the variables in X ($l_i \leq x_i \leq u_i$). For $y_l, y_u \in Z$, let $\Gamma = \{\gamma_1, \ldots, \gamma_p\}$ be the subset of X such that*

$$\Gamma = \{x_i \in X \mid y_l \leq l_i \wedge u_i \leq y_u\} \ .$$

Let $\rho = y_u - y_l + 1$. Let $G_D = (V, E)$ be the disequality graph associated to D_0, such that $V = \{v_1, \ldots, v_n\}$ and $\{v_i, v_j\} \in E$ if and only if $x_i - x_j \neq 0 \in D_0$ or $x_j - x_i \neq 0 \in D_0$. If G_D contains a clique of size greater than ρ then D_0 is inconsistent.

Example 3.3. *Consider the set of disequality constraints $D_0 = \{(x - y \neq 0), (y - z \neq 0), (z - x \neq 0)\}$ with variables y, z that have the same range, $0 \leq y, z \leq 1$, and variable x that has range $0 \leq x \leq 0$ which is a subset of the common range. Let $y_l = 0$ and $y_u = 1$; then $\Gamma = \{x, y, z\}$. A clique consisting of variables x, y, z is present in G_D. Since $|\Gamma| = 3 > 2 = \rho$, the constraints are inconsistent. An explanation of inconsistency consists of the disequality constraints $\{(x - y \neq 0), (y - z \neq 0), (z - x \neq 0)\}$ and the inequality constraints that generated the range, $0 \leq y, z \leq 1$, $0 \leq x \leq 0$.*

The check based on Theorem 3.2 results in one of three outcomes: A suitable clique has been found and inconsistency is declared; a large enough clique was not found because of the heuristic nature of the algorithm; a large enough clique is known not to exist. In the first case, an explanation of inconsistency is derived from the disequalities forming the clique and the inequalities responsible for the bounds. In the last two cases, the result is inconclusive, because the chromatic number of a graph can be arbitrarily larger than the size of the largest cliques. However, if a large enough clique does not exist in the graph, and the interpretation is not known to be a model, we avoid a full check for inconsistency, which is rather expensive and likely to fail. (If the interpretation is a model, on the other hand, the consistency check must be performed for the whole decision procedure to be sound.)

3.4.3 Consistency Check for Abstract Models

If the constraints correspond to a model of φ^b, we want to decide consistency and compute a model of φ in case the answer is affirmative. For this, we resort to finite instantiation. Specifically, we can encode each integer variable with enough binary variables to span its range and translate the satisfiability problem for a conjunction of inequality and disequality constraints into a propositional satisfiability problem.

Theorem 3.1 applies when the constraint graph consists of one SCC. If that is not the case, we examine the SCC quotient graph one SCC at the time. If there is no negative cycle in the constraint graph G, the only reason for unsatisfiability is the inability to satisfy the disequalities within some SCC of G. Therefore, if the finite instantiation of each SCC is satisfiable, the entire set of constraints is satisfiable. This can be shown as follows.

Let G be the constraint graph. Extend G by adding one edge for every disequality constraint $x - y \neq n$ (where n may be 0) such that x and y belong to different SCCs. Let $\preceq$ be the preorder defined by $u \preceq v$ if there is a path in G from u to v. (The preorder is updated after each edge addition.) If $x \preceq y$, add $y - x \leq -n - 1$ to E; if $y \preceq x$, add $x - y \leq n - 1$. If x and y are not comparable in the preorder, add either $y - x \leq -n - 1$ or $x - y \leq n - 1$, but not both. Note that adding these edges does not create cycles, and therefore does not change the SCCs of G. (See Sect. 3.5.)

Let $\widehat{G} = (\widehat{V}, \widehat{E})$ be the SCC quotient graph of the extended G. Consider the vertices in $\widehat{V}$ starting from the minimal SCCs (those with no predecessors) and proceeding in a chosen topological order. Let A_i be the i-th SCC in that order and let α_i be a solution for the constraints corresponding to its edges. Inductively assume that β_{i-1} is a solution for the constraints in the subgraph induced by $\bigcup_{0<j<i} A_j$. Let k be the maximum amount by which any constraint corresponding to an edge into A_i is violated. (Let $k = 0$ if no such violation exists.) Finally, let $\alpha'_i = \alpha_i - k$. Then, $\beta_i = \beta_{i-1} \cup \alpha'$ is a solution for the constraints in the subgraph induced by $\bigcup_{0<j\leq i} A_i$.

3.5 Algorithm

We assume a decision procedure for IDL based on propositional abstraction. The given IDL formula φ is translated into a propositional formula φ^b as described in Sect. 3.2. A **propositional reasoning engine** enumerates the models to φ^b and calls the **theory solver** to determine whether that abstract model corresponds to a consistent interpretation of the integer-valued variables.

The theory solver for IDL is relatively efficient. Therefore, it is advantageous to call it also on a partial interpretations to terminate the fruitless search of part of the state space, or to learn so-called **theory consequences** [NO05]. Our implementation follows this approach, though the equality constraints ($x - y = n \neq 0$) are split and the full check for inconsistencies due to disequalities is applied only to abstract models. (See lines 38–42 of Fig. 3.1.) We omit the details of the incremental implementation of the Bellman-Ford algorithm. The interested reader is referred to [WIGG05].

3.5.1 The Theory Solver

The theory solver is called with a collection of arithmetic literals whose corresponding propositional literals are true in a (partial) interpretation of the propositional formula φ^b; it then decides whether there is an interpretation to the integer-valued variables that satisfies the conjunction of all those literals. The first step is to obtain a set of arithmetic atomic formulae (without negations) from the given set of literals. The given literals are rewritten and divided into Q, I, and D as described in Sect. 3.2.

The theory solver, whose pseudocode is shown in Figures 3.1 and 3.2, adopts the **layered** approach of MathSAT [BBC$^+$05b]. For IDL, it considers three main layers: equalities, inequalities, and disequalities. Let $X_= \subseteq X$ be the set of integer-valued variables appearing in the equalities in Q. The theory solver creates an undirected equality graph $\mathcal{Q} = (X_=, \Gamma)$, where

$$\Gamma = \{\{x_i, x_j\} : x_i = x_j \in Q\} \enspace .$$

The vertices of $\mathcal{Q}$ are in the same class if they are made equivalent by the equality constraints. The feasibility of Q with D_0 is checked by comparing the equivalence class of the two vertices of each disequality constraint in D_0. If two vertices are in the same class, an explanation of infeasibility is returned. If the set of equality constraints is feasible, the variables in the same class are merged into a single variable, and some simplified constraints in D and I are dropped from the set.

The algorithm continues by checking the feasibility of the set of inequality constraints. Let $V \subseteq V_Z$ be the set of integer-valued variables appearing in I. The theory solver creates a constraint graph $G = (V, E, \lambda)$ from I as explained in Sect. 3.2. The Bellman-Ford algorithm is run on G. If a negative cycle is found, the set I is infeasible; a negative cycle with a subset of Q provides the explanation of infeasibility.

```
TheorySolver ($C$) {
    $Explanation$ = EqualitySolver ($Q$, $D_0$);
    if ($Explanation = SAT$) $Explanation$ = InequalitySolver ($I$);
    if ($Explanation = SAT$) $Explanation$ = DisequalitySolver ($D$);
    return $Explanation$
}

EqualitySolver ($Q$, $D_0$) {
    $\mathcal{Q}$ = CreateEqualityGraph ($Q$);
    return $Explanation$ = CheckFeasibilityOfEqualityConstraints ($\mathcal{Q}$, $D_0$);
}

InequalitySolver ($I$) {
    $G$ = CreateConstraintGraph ($I$);
    $NegCycle$ = BellmaFordAlgorithm ($G$);
    if ($NegCycle$) return GenerateExplanationFromNegCycle ($NegCycle$);
    else return $SAT$
}

DisequalitySolver ($D$) {
    $SCC$ = GenerateZeroSlackSCCOfConstraintGraph ($G$);
    $Explanation$ = CheckFeasibilityOfZeroSlackSCC ($SCC$, $D$);
    if ($Explanation \neq SAT$) return $Explanation$;
    else {
        $SCC'$ = GeneratePositiveSlackSCCOfConstraintGraph ($G$);
        return CheckFeasibilityOfPositiveSlackSCC ($SCC'$, $D$);
    }
}

CheckFeasibilityOfZeroSlackSCC ($SCC$, $D$) {
    For each $d \in D$ {
        $Explanation$ = CheckFeasibilityOfDisequalityConstraint ($SCC$, $d$);
        if ($Explanation \neq SAT$) return $Explanation$;
        else DropValidConstraint ($d$,$D$);
    }
    return $SAT$;
}
```

Figure 3.1: Theory solver algorithm

```
CheckFeasibilityOfPositiveSlackSCC ($SCC'$, $D$) {
    for each $SCC' \in SCC'$ {
        $(L, U)$ = GenerateBoundsForEachVariableInSCC ($SCC'$);
        $Explanation$ = CheckFeasibilityOfBoundsWithClique($SCC'$, $D$, $L$, $U$);
        if (($Explanation = UNDECIDED$ or
           $Explanation = PROB\_SAT$) and interpretation is a model) {
            $CNF$ = SmallDomainEncodingForConstraintsInSCC ($SCC'$, $D$, $L$, $U$);
            $Explanation$ = SatSolver ($CNF$);
            if ($Explanation \neq SAT$) return $Explanation$;
        }
        else return $Explanation$;
    }
    return $SAT$;
}

GenerateBoundsForEachVariableInSCC ($SCC'$) {
    $x$ = FixValueOfOneVertexInSCC ($SCC'$);
    $U$ = ComputeUpperBoundForEachVariableInSCC ($SCC'$,$x$);
    $L$ = ComputeLowerBoundForEachVariableInSCC ($SCC'$,$x$);
    return $(L, U)$;
}

CheckFeasibilityOfBoundsWithClique ($SCC'$, $D$, $L$, $U$) {
    $\Gamma$ = GatherVariablesWithSameBounds ($D$, $L$, $U$);
    $\rho$ = GetBoundForGatheredVariables ($\Gamma$);
    $D'$ = CollectRelevantDisequalityConstraints ($D$,$\Gamma$);
    $\Gamma'$ = RemoveIrrelevantVariableByCheckingDegree ($\Gamma$, $D'$);
    if ($n(\Gamma') \leq \rho$ and $n(Var(D)) = n(\Gamma)$) return $PROB\_SAT$;
    else if ($n(\Gamma') \leq \rho$ and $n(Var(D)) \neq n(\Gamma)$) return $UNDECIDED$;
    if ($n(D') < (\rho \cdot (\rho + 1))/2$ and $n(Var(D)) = n(\Gamma)$) return $PROB\_SAT$;
    else if ($n(D') < (\rho \cdot (\rho + 1))/2$ and $n(Var(D)) \neq n(\Gamma)$) return $UNDECIDED$;
    $C$ = GenerateMaxClique ($\Gamma'$, $D'$);
    if ($n(Var(C)) < \rho$ and $n(Var(D)) = n(\Gamma)$) return $PROB\_SAT$;
    else if ($n(Var(C)) < \rho$ and $n(Var(D)) \neq n(\Gamma)$) return $UNDECIDED$;
    else return GenerateExplanationFromMaxClique ($SCC'$,$C$);
}

SmallDomainEncodingForConstraintsInSCC ($SCC'$, $D$) {
    return EncodingForBoundsOfEachVariableInSCC ($SCC'$) $\cup$
        EncodingForInequalityConstraintsInSCC($SCC'$) $\cup$
        EncodingForDisequalityConstraints ($D$);
}
```

Figure 3.2: Theory solver algorithm (continued)

Equality constraints are involved in the explanation if the constraints on the negative cycle were obtained by simplification in the equality layer. If there is no negative cycle in G, the set $I \cup Q$ is feasible; therefore a solution $\delta : V \rightarrow Z$ is returned by the Bellman-Ford algorithm.[2]

The (simplified) set I combined with D is considered in the next step. Let G_0 be the subgraph of G such that the edges with non-zero slacks for solution δ are removed from G. Since the slacks of the edges of G_0 are zero, the difference between the values of two variables in the same SCC of G_0 is the same in all solutions to the constraints. In fact, each cycle in G_0 is of length 0 [LM05]; hence, if x and y are on one cycle of G_0 and the distance from x to y along the cycle is k, then the distance from y to x is $-k$. It follows that every solution to I must satisfy $y - x \leq k$ and $x - y \leq -k$, that is, $y - x = k$. In other words, an SCC of G such that its vertex set induces also an SCC of G_0 has only one family of solutions. (See Sect. 3.4.)

Each disequality constraint $d \in D$ is checked for feasibility against each SCC of G_0. If the two variables x, y in $x - y \neq n$ (where n may be 0) are in the same SCC of G_0 and $\delta(x) - \delta(y) = n$, then the set $I \cup Q \cup D$ is infeasible. The violated disequality d, together with a cycle that contains x and y and an appropriate subset of Q constitutes the explanation of infeasibility. If the two variables x and y in d are in the same SCC of G_0 and $\delta(x) - \delta(y) \neq n$, then d is redundant and is dropped from D. Disequalities connecting variables in different SCCs of G_0 are simply passed on to the next phase of the procedure. If no infeasibility is detected with G_0, a final feasibility check is performed by the small domain encoding method discussed in Sect. 3.4. For each SCC of G, Theorem 3.1 is used to compute bounds for each variable as follows.

To compute the upper bound for each variable, a variable in the SCC is chosen arbitrarily as source. (Variable x in Theorem 3.1.) The distance from it is computed for each variable in the SCC by the Bellman-Ford algorithm. The lower bound for a variable is computed as its distance from the same source variable used to compute the upper bound after reversing the edges in the SCC. (Note that one cannot replace the distances computed by these invocations of the shortest path algorithm with those computed on G_a.)

Some inequalities and disequalities may be automatically satisfied for all values of the variables in

[2] The algorithm is, in principle, applied to the augmented graph G_a described in Sect. 3.2. In practice, no augmentation of G is required: it suffices to initialize all distances to 0.

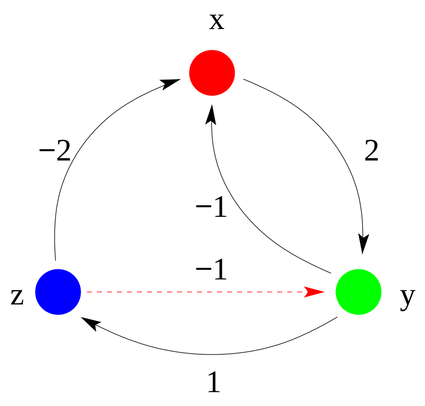


Figure 3.3: SCC without any negative cycle

their ranges. For instance, if $0 \leq x \leq 1$ and $2 \leq y \leq 3$, then $x \neq y$ and $y - x \leq 4$ are both satisfied. These constraints are therefore ignored in the successive steps, which consist of a quick check based on finding a clique of the disequality graph, possibly followed by propositional encoding and satisfiability check.

Some disequalities may be strengthened by converting them into a disjunction of inequalities and dropping one disjunct that is always false due to the ranges of the variables. For instance, $x - y \neq 1$, where $1 \leq x \leq 2, 0 \leq y \leq 0$ can be strengthened to $x - y \geq 2$ because $x - y \leq 0$ is false for x and y in the given ranges. The range of x therefore shrinks to $2 \leq x \leq 2$.

Example 3.4. *Consider the SCC without any negative cycle in Fig. 3.3. The edges correspond to the inequality constraints* $\{(x - y \leq -1), (y - x \leq 2), (z - y \leq 1), (x - z \leq -2)\}$*. Additionally, there is a set of disequality constraints* $\{(x - y \neq 0), (y - z \neq 0), (z - x \neq 0), (z - x \neq 1), (y - z \neq -1)\}$*. Variable* x *is chosen as source; hence both bounds of* x*,* x_l *and* x_u*, are given value 0. Using the Bellman-Ford algorithm,* y_u *is assigned 2 and* z_u *is assigned 3. Reversing the edges in the SCC,* y_l *is assigned 1 and* z_l *is assigned 2. Therefore, the ranges are* $\{0 \leq x \leq 0, 1 \leq y \leq 2, 2 \leq z \leq 3\}$*. The inequalities* $\{(x - y \leq -1), (y - x \leq 2)\}$ *and the disequalities* $\{(x - y \neq 0), (z - x \neq 0), (z - x \neq 1)\}$ *are automatically satisfied for all values of the variables in their ranges. The disequality* $(y - z \neq 0)$ *is strengthened to* $(y - z \leq -1)$*. Consequently,* $(y - z = -1)$ *and the disequality* $(y - z \neq -1)$ *cannot be satisfied.*

The application of Theorem 3.2 is described in lines 53–66 of Fig. 3.2. We identify sets of variables for which Theorem 3.1 produces the same bounds and we check whether there are enough disequalities among the variables in one such set to cause inconsistency.

Specifically, suppose a set $\Gamma = \{\gamma_1, \ldots, \gamma_p\}$ of variables is found such that all variables in Γ have the same bounds y_l and y_u. Variables whose range is a subset of the common range are added to Γ.

Let $\rho = y_u - y_l + 1$. If $|\Gamma| < \rho$, disequalities cannot cause inconsistency of this set of variables. If, on the other hand, the number of variables exceeds their common range, we check whether the disequalities form a clique of size greater than ρ. We first eliminate from Γ all variables that appear in fewer than ρ disequalities of the form $\gamma_i \neq \gamma_j$ ($\gamma_i, \gamma_j \in \Gamma$). If Γ is not empty after this process, we greedily grow a clique, adding every time the variable appearing in the largest number of disequalities among the surviving members of Γ. This greedy algorithm does not always find the largest clique, but is fast and works well in practice.

In the final step of the theory solver, the constraints and the bounds are converted to a set of clauses whose satisfiability is established by calling a propositional SAT solver.[3] If the clauses are satisfiable, an interpretation for the integer variables is extracted from the solution. Otherwise, an explanation for the unsatisfiability is derived as follows from the proof of unsatisfiability returned by the SAT solver, which consists of a subset of the clauses that are found to be unsatisfiable. (The **unsatisfiable core**.)

Every propositional clause in the unsatisfiable core is derived from some arithmetic constraint. If a clause appears in the unsatisfiable core, the parent constraint is included in the explanation. The bound constraints on the integer variables also contribute to unsatisfiability. They are accounted for by including the constraints that form the two shortest path spanning trees found during the computation of the bounds.

Example 3.5. *If Example 3.4 continues without disequality strengthening, the constraints* $\{(z-y \leq 1), (x-z \leq -2), (y-z \neq 0), (y-z \neq -1)\}$ *and the bounds* $\{0 \leq x \leq 0, 1 \leq y \leq 2, 2 \leq z \leq 3\}$ *are converted to the set of clauses below. The variable* y *is substituted by* $\iota + 1$*, and the variable* z *is substituted by* $\zeta + 2$*. As a result, the range of* ι *and* ζ *are* $0 \leq \iota \leq 1$, $0 \leq \zeta \leq 1$*, and the number of bits used for* ι *and* ζ *during*

[3] Our current encoding of the ranges is rather unsophisticated. We are implementing a heuristic approach to minimizing the total number of encoding bits required.

the encoding is one instead of two.

$$\varphi = (\neg\zeta_0 \vee \iota_0) \wedge (\neg\iota_0 \vee \zeta_0) \wedge (\neg\iota_0 \vee \neg\zeta_0) \wedge (\iota_0 \vee \zeta_0).$$

With the set of clauses φ, a propositional SAT solver is called. Since the set of clauses is unsatisfiable, the unsatisfiable core Ω is returned:

$$\Omega = (\neg\zeta_0 \vee \iota_0) \wedge (\neg\iota_0 \vee \zeta_0) \wedge (\neg\iota_0 \vee \neg\zeta_0) \wedge (\iota_0 \vee \zeta_0).$$

The inequality constraints that are responsible for the bounds are extracted as an explanation from the SCC in Fig. 3.3. For the variables x, y, z in the SCC, the edges that lie on the forward and backward shortest paths from each variable to the fixed variable x are gathered. Therefore, we get $\{(y - x \leq 2), (z - y \leq 1), (x - z \leq -2)\}$ as an explanation for the bounds $\{0 \leq x \leq 0, 1 \leq y \leq 2, 2 \leq z \leq 3\}$. The parent constraints of the clauses left in Ω are finally gathered; they are $\{(z - y \leq 1), (y - z \neq 0), (y - z \neq -1)\}$. As a result, the full explanation for the infeasibility is

$$\{(y - x \leq 2), (z - y \leq 1), (x - z \leq -2), (y - z \neq 0), (y - z \neq -1)\}.$$

Five constraints suffice to explain the infeasibility of the original nine constraints.

3.6 Related Work

Propositional abstraction as an approach to satisfiability modulo theories was proposed in [BDS02]. Notable solvers based on that principle are MathSAT [BBC+05b, BBC+05a], ICS and Yices [dMR02, DdM06a, DdM06b], Verifun [FJOS03], BarcelogicTools [GHN+04, NO05], SLICE [WIGG05], and SATORI [IPC03]. ASAP [KOSS04] takes a dual approach, in which satisfiability of the propositional abstraction guarantees satisfiability of the original quantifier-free Presburger formula, while UCLID [LS04] is an eager solver. Our propositional enumeration engine is the one of [JHS05, JS05].

Finite instantiations for equality logic are studied in [PRSS02] and extended to difference logic in [TSSP04]; this last work has several points of contact with ours, but also important differences. The approach of [TSSP04] is eager, and the ranges are computed once and for all before invoking the propositional

SAT solver. In contrast, we advocate a lazy approach and a computation of the ranges that takes place in the theory solver. Because of that, we may compute ranges more than once, but the size of the range for each variable in our algorithm is bounded by the sum of the slacks in the SCC, which is much smaller than $n + maxC$, where $maxC$ is the sum of absolute constants in the formula. In practice, ranges are much smaller in our algorithm. Moreover, we compute ranges by simply finding shortest paths in the constraint graph. The algorithm of [TSSP04], on the other hand, enumerates paths in the constraint graph and is exponential in the worst case.

Recent work by Ganai **et al.** [GTG06] presents a polynomial algorithm for the computation of ranges, which improves over the one of [TSSP04], but shares the basic approach: Ranges are allocated initially, so as to be adequate for every formula built from the given set of difference constraints. Disequalities are converted to disjunctions of inequalities, instead of being retained as such in the formulation of the problem. The theory consistency problem is never converted to propositional satisfiability. Instead, range propagation allows the solver to refine the initial ranges.

MathSAT introduced the notion of layered, incremental theory solver, and that of delayed theory combination; DPLL(t) the idea of exhaustive theory propagation, both of which are included in our implementation. The importance of considering zero-slack SCCs was first pointed out in [LM05], which deals with RDL. Finally, [WIGG05] discusses an efficient way to implement a recursive, backtrackable Bellman-Ford algorithm.

3.7 Experimental Results

We have implemented the algorithm presented in Sect. 3.5 in Sateen, a theorem prover for quantifier-free first-order logic that combines the propositional reasoning engine of [JHS05, JS05] with theory-specific procedures. A first set of experiments were done with the full set of QF_IDL (Quantifier free integer difference logic) benchmarks from SMT-COMP (Satisfiability Modulo Theories Competition [SMTa]). The experiments were performed on a 1.7 GHz Pentium 4 with 2 GB of RAM running Linux. Time out was set at 3600 seconds. Sateen was compared with BarcelogicTools [DPL], Yices-0.1.1 [Yic] and MathSAT 3.3.1 [Mat]. The compared solvers are the ones that were submitted to SMT-COMP in 2005.

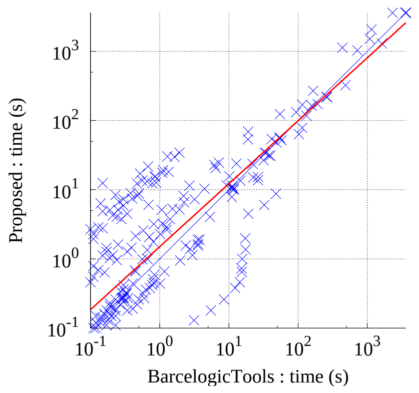


Figure 3.4: BARCELOGICTOOLS vs. Sateen on QF_IDL

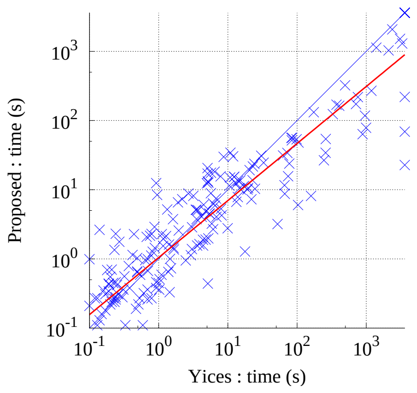


Figure 3.5: YICES vs. Sateen on QF_IDL

Figures 3.4–3.6 show scatterplots comparing BarcelogicTools, Yices and MathSAT to Sateen. Points below the diagonal represent wins for Sateen. Each scatterplot shows two lines: The main diagonal, and $y = \kappa \cdot x^{\eta}$, where κ and η are obtained by least-square fitting. Figure 3.4 shows that Sateen is comparable to BarcelogicTools. In Figures 3.5 and 3.6, Sateen shows better results compared to Yices and MathSAT, especially on hard problems. The SMT-COMP benchmark formulae are such that usually the sets of constraints passed to the theory solver either contain few disequality constraints, or are such that the disequality constraints are dealt with by the zero-slack SCC algorithm. The main purpose of these experiments is therefore not to show the effectiveness of the newly proposed algorithm for finite instantiations, but to establish that Sateen is, overall, a competent solver for IDL, comparable to some of the best tools in the field.

To assess the effectiveness of the finite instantiation approach, we have generated two benchmark suites where disequality constraints play a significant role: the Queens Suite and the Job Shop Scheduling Suite. The Queens Suite contains n-Queens problem and n-Super-Queens problem. The n-Queens problem is a classical combinatorial search problem which consists of placing n queens on a $n \times n$ board so that they do not attack each other. In the n-Super-Queens problem, each queen's placement is more restricted by allowing it also the knight's moves. The Job Shop Scheduling problem is a randomly generated problem which checks the feasibility of processing a number of jobs, each consisting of several tasks, on a given set of machines in a given amount of time. These two sets of benchmarks have disequality constraints that cause

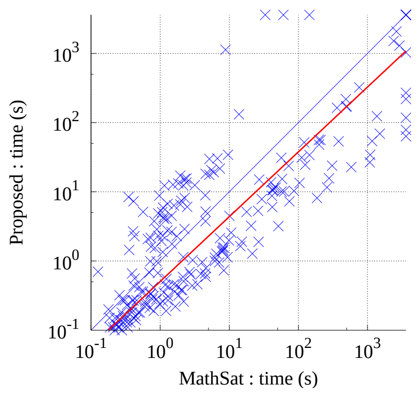


Figure 3.6: MATHSAT vs. Sateen on QF_IDL

pigeonholing problems. In the experiment on these benchmarks, the timeout was set to 1000 seconds.[4]

Figures 3.7–3.9 shows that Sateen is often orders of magnitude faster than the other solvers on these problems. The $\times$ symbols denotes the experiments on the Queens benchmarks, and the $+$ symbols denotes the experiments on the Job Shop Scheduling benchmarks. We also provide the comparison between Sateen with our proposed algorithm and a version of Sateen that splits disequalities. Figure 3.10 shows that the finite instantiation algorithm works significantly better than the splitting method.

Table 3.1 shows the number of calls and conflicts involving the equality layer (EQ), the Bellman-Ford layer (INEQ), the zero-slack SCC layer (ZS), the clique generation layer (CLQ) and the finite instantiation layer (FI) on selected benchmarks. **BV** and **AF** correspond to the number of propositional variables and atomic formulae, respectively. In the entries of the form X/Y, X is the number of conflicts and Y is the number of calls. The data show that each layer contributes to finding conflicts. In particular, the clique generation layer is very effective in finding conflicts in the Job Shop Scheduling benchmarks, which enables the solver to avoid the finite instantiation layer.

[4] Although, the results of SMT-COMP [SMTb] in 2006 show that Sateen is still behind the three other solvers above, Sateen gives significantly better result on the n-Queens and Job Shop Scheduling benchmarks.

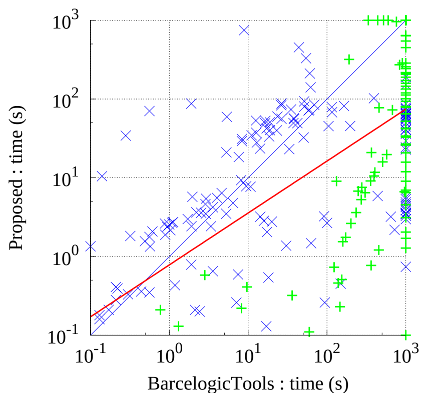


Figure 3.7: BARCELOGICTOOLS vs. Sateen on Job Shop Scheduling and Queen suites

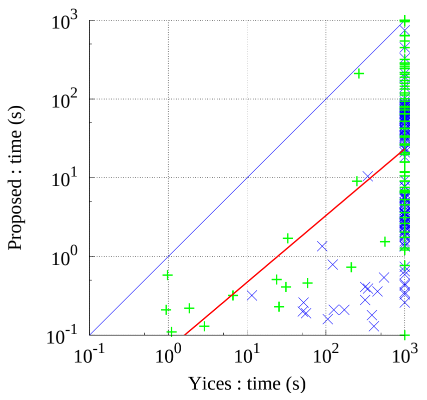


Figure 3.8: YICES vs. Sateen on Job Shop Scheduling and Queen suites

3.8 Conclusions

We have presented an approach to solving integer difference logic that is particularly effective when the constraints to be solved are rich in disequalities. By restricting consideration to a small sufficient set of solutions, we are able to compute bounds for the integer variables occurring in the constraints. Experiments indicate that this approach is more effective than splitting disequalities into the disjunction of inequalities. Further improvements in efficiency are expected from a more sophisticated encoding scheme for the finite instances that we are currently developing.

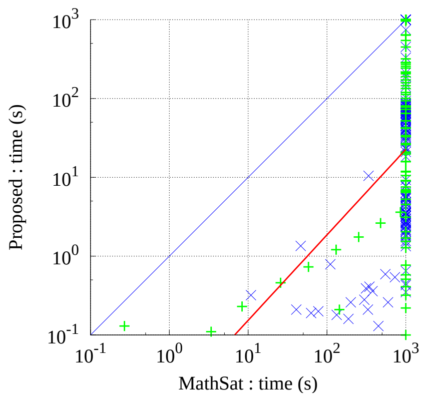


Figure 3.9: MathSAT vs. Sateen on Job Shop Scheduling and Queen suites

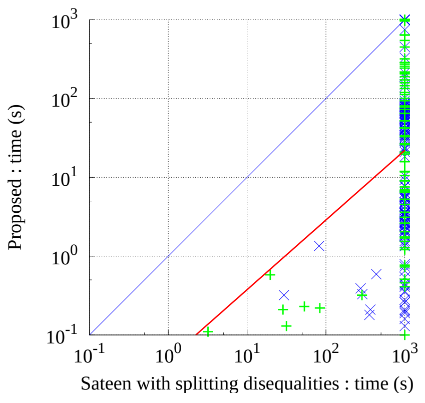


Figure 3.10: Sateen with splitting disequalities vs. Sateen on Job Shop Scheduling and Queen suites

Benchmark	**BV**	**AF**	**SAT**	**EQ**	**INEQ**	**ZS**	**CLQ**	**FI**
diamonds.10.5.i.a.u	0	121	UNSAT	0/0	90/1199	0/0	0/0	0/0
DTP_k2_n35_c245_s2	0	490	SAT	0/0	709/7200	0/0	0/0	0/0
inf-bakery-mutex-18	76	328	UNSAT	71/1498	84/1533	25/2070	0/0	0/0
FISCHER9-10-ninc	1146	686	SAT	54/55	0/1	0/0	0/0	0/0
queen30-1	0	1365	SAT	0/0	0/1	0/1	0/0	0/1
super_queen60-1	0	5664	SAT	0/0	0/1	0/1	0/0	0/1
jobshop30-2-20-20-4-4-12	0	2820	UNSAT	0/0	632/1264	0/631	1/631	0/1
jobshop40-2-20-20-4-4-12	0	4960	SAT	109/258	3/1343	109/1282	58/1172	0/1
jobshop50-2-25-25-4-4-11	0	7700	UNSAT	0/0	1802/3604	0/1801	1/1801	0/0
jobshop60-2-30-30-4-4-12	0	11040	SAT	239/538	3/2773	239/2682	88/2442	0/1

Table 3.1: Number of Calls and Conflicts

Chapter 4

Efficient Term-ITE Conversion

Satisfiability Modulo Theories (SMT) solvers find increasing applications in areas like formal verification in which one needs to reason about complex Boolean combinations of numerical constraints. The most common approach to this problem leverages the efficiency of modern propositional satisfiability solvers that work on a propositional abstraction of the given formula. At the same time, they interact with theory solvers, which check conjunctions of literals for consistency and learn consequences (new lemmas) from them. This approach has come to be known as DPLL(T) [NO05].

Among the logics for which theory solvers have been developed in recent times, linear arithmetic is one of the most useful and well-researched. Many current solvers adopt some variant of the simplex algorithm. In particular, the backtrackable version of [DdM06a, DdM06b] fits well in the DPLL(T) scheme and has shown good results in practice for both integer and real-valued variables.

The Boolean dimension of many SMT instances, however, continues to pose a challenge to solvers. In this chapter, we address this problem. In particular, we focus on those instances that make extensive use of the **term-if-then-else** (ITE) operator. This operator facilitates the analysis of problems in which paths through control-flow graphs must be translated into SMT formulae. It is not surprising, therefore, that many of the available benchmark instances for linear arithmetic are rich in term-ITEs. Given a code fragment that contains **if** statements, a verification condition can be naturally formulated with ITEs as shown in Fig. 4.1.

Two major approaches can be envisioned to deal with term-ITEs. On the one hand, one can modify the theory solver to deal with conditional expressions. Without ITEs, every assignment to an atom of the SMT formula adds to a conjunction of literals that is analyzed by the theory solver. With ITEs, this is no

```
main(void){
      .
      .
      .
   if(x = 0){
      y = 1;
   }else if(x = 1){
      y = 2;
   }else if(x = 2){
      y = 3;
   }else {
      y = 4;
   }
   assert(y ≤ 2);
}
```

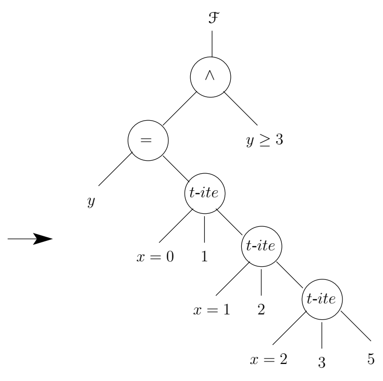


Figure 4.1: Verification condition $\mathcal{F}$ with term-ITEs

longer the case. In order to analyze the atom, the conditional expressions of the ITEs need to be assigned. On the other hand, one can eliminate all the ITEs from the formula by rewriting. The problem here is that the rewritten formula may retain a lot of redundancies depending on how one rewrites it. We address this problem by a procedure based on cofactoring and theory simplification. Although our approach may cause a blow-up, it often simplifies the formula in practice. Our approach is applied to linear arithmetic logic in this chapter; however, it can be easily applied to other logics like the logic of equality and uninterpreted function symbols (EUF), the logic of bit-vector, or the logic of arrays. Only the terminal cases are different in each logic. Our experiments show that our approach is promising and often speeds up a solver by orders of magnitude. The experiments also demonstrate the effectiveness of theory simplification.

The rest of this chapter is organized as follows. Section 4.1 defines notation and summarizes the main concepts. Section 4.2 discusses motivation and outlines our approach to the problem. Section 4.3 presents the simplifications applied before invoking the term-ITE conversion. Section 4.4 presents an algorithm for term-ITE conversion with theory reasoning. After a survey of related work in Sect. 4.5, experiments are presented in Sect. 4.6, and conclusions are offered in Sect. 4.7.

4.1 Preliminaries

We consider the satisfiability problem for linear arithmetic logic, which is the quantifier-free fragment of first-order logic that deals with linear arithmetic constraints. Let V_R be the set of real-valued variables. The formulae in linear arithmetic logic are inductively defined as the largest set that satisfies the following rules.

- A propositional variable $a \in V_P$ is a formula.
- A real number $c \in \mathbb{R}$ is a (constant) LA term.
- The product cx of a real number $c \in \mathbb{R}$ and a real-valued variable $x \in V_R$ is an LA term.
- If t_1 and t_2 are LA terms, so are $t_1 + t_2$ and $t_1 - t_2$.
- If t_1 and t_2 are LA terms, and f is a formula, then *tite*(f, t_1, t_2) is an LA term.
- If t_1 and t_2 are LA terms, and $\sim$ is a relational operator in $\{=, \neq, <, \leq, >, \geq\}$, then $t_1 \sim t_2$ is a formula.
- If f_1, f_2, and f_3 are formulae, then $\neg f_1$, $f_1 \wedge f_2$, $f_1 \vee f_2$ and *ite*(f_1, f_2, f_3) are formulae.

The semantics are defined in the usual way; in particular, *ite*(f_1, f_2, f_3) is equivalent to $(f_1 \wedge f_2) \vee (\neg f_1 \wedge f_3)$. An **atomic formula** is one of the form $t_1 \sim t_2$. A **positive literal** is an atomic formula or a propositional variable; a **negative literal** is the negation of a positive literal.

A model for a formula f is an assignment of values to the variables in the formula that is consistent with the type of each variable and that makes the formula true. A formula that has at least one model is **satisfiable**. In recent years, decision procedure for *LA*, and other fragments of quantifier-free first-order logic, have been based on the DPLL procedure. Given a formula $\mathcal{F}$, the propositional abstraction $\mathcal{F}_b$ of $\mathcal{F}$ is built by substituting each atomic formula with a new propositional variable. As the DPLL procedure provides a model for $\mathcal{F}_b$, a **theory solver** for *LA* is invoked with the set of atomic formulae that are assigned. The theory solver checks the feasibility of the set. If the set is feasible, then the model is also a model in theory. If the set is infeasible, then the explanation of the infeasibility is returned to the DPLL procedure.

The procedure continues until it finds a complete model, or decides that $\mathcal{F}$ is **unsatisfiable** in the given theory.

4.2 Term-ITE Conversion

An *LA* formula can often be expressed more concisely by using term-ITEs. For example, Fig. 4.2 shows that the formula f in (a) is equivalent to the more verbose formula f' in (b). Despite the conciseness afforded by term-ITEs, a *LA* formula with term-ITEs is often converted into a formula without them, so that the formula may be solved by an SMT solver based on the propositional abstraction.

4.2.1 Two Methods for Term-ITE Conversion

A common way to eliminate these term-ITEs is to introduce a fresh constant that replaces the term-ITE. In particular, an *LA* formula $f(\mathit{tite}(g, t_1, t_2))$ is converted to the equisatisfiable

$$f(c) \wedge \mathit{ite}(g, t_1 = c, t_2 = c) \ , \tag{4.1}$$

where c is a constant that does not appear in the given formula. The advantage of this conversion is that it does not blow up; however, it often retains redundancies in the converted formula. For example, the formula $\mathit{tite}(g, 1, 2) = \mathit{tite}(h, 3, 4)$ can be reduced to $\perp$, whereas the conversion generates $\mathit{ite}(g, c = 1, c = 2) \wedge \mathit{ite}(h, c = 3, c = 4)$ that contains a redundancy. To remove the redundancy, additional theory reasoning is required. A naive approach to the term-ITE conversion will be to combine every term in the left-hand side of the relational operator with the terms in the right-hand side depending on the conditional terms of term-ITEs. In particular, an *LA* formula $f(\mathit{tite}(g, t_1, t_2))$ is converted according to following conversion rule [JDB95].

$$f(\mathit{tite}(g, t_1, t_2)) \iff \mathit{ite}(g, f(t_1), f(t_2)) \ . \tag{4.2}$$

This approach removes the redundancy in the above example on the fly; however, as Fig. 4.2 illustrates, the converted formula may grow exponentially large in the worst case.

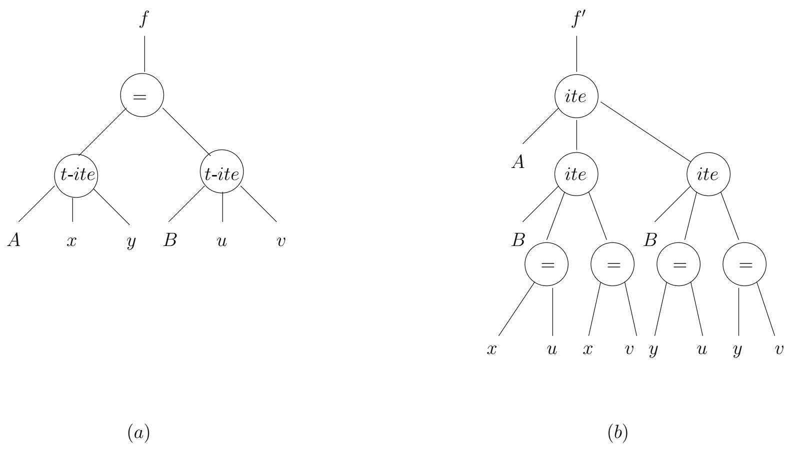


Figure 4.2: Term-ITE conversion

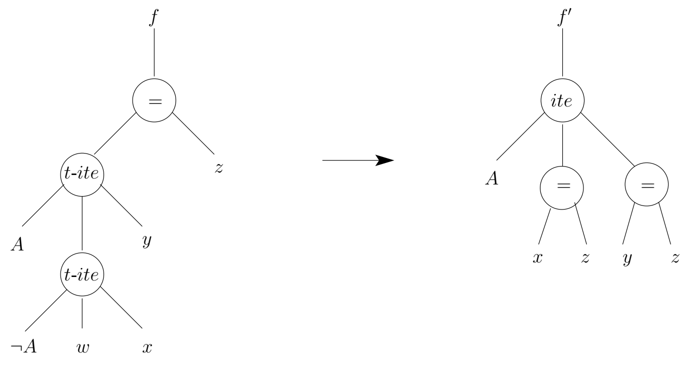


Figure 4.3: Term-ITE conversion with cofactor

4.2.2 Term-ITE Conversion with Cofactors

As an alternative to the approaches described in Sect. 4.2.1, term-ITE conversion can be done by computing cofactors.

Definition 4.1. *Let $f(x_1, ..., x_n)$ be an LA formula, where each x_i is a positive literal. Then,*

$$f_{x_i} = f(x_1, ..., x_{i-1}, \top, x_{i+1}, ..., x_n)$$

$$f_{\neg x_i} = f(x_1, ..., x_{i-1}, \bot, x_{i+1}, ..., x_n)$$

are the positive and negative cofactors of f with respect to x_i.

Theorem 4.2 (Boole)**.** *Let $f(x_1, ..., x_n)$ be an LA formula. Then $f(x_1, ..., x_n) = (x_i \wedge f_{x_i}) \vee (\neg x_i \wedge f_{\neg x_i}) =$ ite$(x_i, f_{x_i}, f_{\neg x_i})$.*

According to Theorem 4.2, the following rule can be used to rewrite an *LA* formula:

$$f(\mathit{tite}(g, t_1, t_2)) \iff \mathit{ite}(x, f_x(\mathit{tite}(g, t_1, t_2)), f_{\neg x}(\mathit{tite}(g, t_1, t_2))) \ . \tag{4.3}$$

By computing the cofactors of f, the conversion may greatly simplify the converted formula. In Fig. 4.3, f is simplified to $\bot$ using (4.3). In particular, the cofactors $f_A \iff (\mathit{tite}(B, 3, 5) = 4)$ and $f_{\neg A} \iff (5 = 4) \iff \bot$ are first computed. Then f is simplified to $(A \wedge f_A)$, and finally reduced to $\bot$ by cofactoring f_A with respect to B.

This kind of simplification can often be applied to the *LA* problems in SMT-LIB [SMTa]. As the previous example shows, the simplification for equality is easily done by comparing two constants. On the other hand, if fresh constants are introduced, redundancy may remain in the converted formula: a fresh constant c replaces the term $\textit{tite}(\textit{ite}(A, B, \bot), \textit{tite}(\neg A, x, 3), 5)$ in f. Then f is rewritten in two steps: first as

$$(c = 4) \wedge \textit{ite}(\textit{ite}(A, B, \bot), c = \textit{tite}(\neg A, x, 3), c = 5) \ \ ,$$

and then as

$$(c = 4) \wedge (c' = c) \wedge \textit{ite}(\textit{ite}(A, B, \bot), \textit{ite}(\neg A, c' = x, c' = 3), c = 5) \ \ ,$$

where c' is another fresh constant. Removing the redundancy from the converted formula requires theory reasoning. While such reasoning is uncomplicated in this example, in general the new constants may make it cumbersome. Although the cofactoring method may give a huge reduction, it may blow up if there is little simplification. Compared to the approach that introduces a fresh constant, it is more aggressive.

Definition 4.3. *Let x be a literal and h be a formula. We write $x \models_T h$ if h is a consequence of x in theory T, and we call h a **theory consequence** of x.*

The cofactoring method can be further extended with theory reasoning. Using the theory propagation method [NO05], an assignment to an atomic predicate may entail assignments to other atomic predicates. For example, in *LA*, if we make an assignment to $(x < 0) = \top$, then $(x < 3) = \top$ and $(x > 1) = \bot$. The following rules show how theory propagation may help in the simplification of the converted formula:

$$\frac{x \models_T h}{f_x(\textit{tite}(h, t_1, t_2)) \iff f_x(t_1)} \tag{4.4}$$

$$\frac{x \models_T \neg h}{f_x(\textit{tite}(h, t_1, t_2)) \iff f_x(t_2)} \ . \tag{4.5}$$

As we compute the cofactors in the term-ITE conversion, we make an assignment to the cofactoring literal. If the cofactoring literal is an atomic formula and the computed cofactor is also an atomic formula, then theory reasoning can be invoked to check the relation between these two atoms. The following consequence of Theorem 4.2 gives an idea of how this simplification can be done; it will be used in Sect. 4.4.

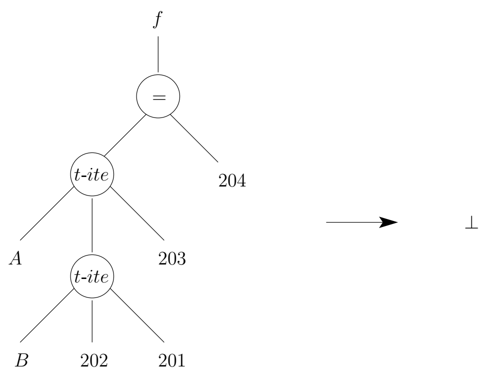


Figure 4.4: Term-ITE conversion with simple check

Theorem 4.4. *Given a formula f of theory T and a literal x_i, if $x_i \models_T f_{x_i}$, then $f \iff x_i \vee f_{\neg x_i}$. If $x_i \models_T \neg f_{x_i}$, then $f \iff \neg x_i \wedge f_{\neg x_i}$.*

4.3 Simple Preprocessing

Before we execute term-ITE conversion for an *LA* formula f, terminal cases for term-ITE are detected and basic simplification is carried out. Let $a \in V_P$; let t_1, t_2, and t_3 be terms and let c_1, c_2, and c_3 be constants. In the *LA* formula, we detect special cases like *tite*$(\top, t_1, t_2) \iff t_1$, *tite*$(\perp, t_1, t_2) \iff t_2$, *tite*$(a, t_1, t_1) \iff t_1$. We also simplify nested term-ITEs such as *tite*$(a,$ *tite*$(a, t_1, t_3), t_2) \iff$ *tite*(a, t_1, t_2), *tite*$(a,$ *tite*$(\neg a, t_3, t_2), t_1) \iff$ *tite*(a, t_2, t_1). For arithmetic terms, $(0 + t_1) \iff t_1, (0 \cdot t_1) \iff 0, (1 \cdot t_1) \iff t_1, (-(-t_1)) \iff t_1, (c_1 + c_2) \iff c_3$, where c_3 is the sum of c_1 and c_2.

Furthermore, if a formula f has a root node that is a relational operator applied to term-ITEs and has leaves that are all constants, then it can be simplified. For simplicity, we only check the case where either of the children of the root node is a constant. Example 4.5 shows such a case.

Example 4.5. *Let f be a formula shown in Fig. 4.4. The formula f is an equality with term-ITEs. As Fig. 4.4 shows, the terms on the left-hand side of the root node are all constants and the one on the right-hand side is also a constant. In such a case, we compare all the constants in the left hand side for equality with the constant on the right,* 204*. Clearly,* $(202 = 204) \iff \perp, (201 = 204) \iff \perp$ *and* $(201 = 203) \iff \perp$*; hence* $f = \perp$.

4.4 Algorithm

We assume that an SMT solver adopts the rewriting procedure. Given an *LA* formula $\mathcal{F}$ with term-ITEs, an SMT solver converts $\mathcal{F}$ into $\mathcal{F}'$ by removing all term-ITEs in $\mathcal{F}$. The SMT solver then decides the satisfiability of $\mathcal{F}'$. In this section, we describe how $\mathcal{F}$ is converted into $\mathcal{F}'$.

As the pseudocode of Fig. 4.5 shows, the main function of term-ITE conversion is called with an *LA* formula $\mathcal{F}$. The formula $\mathcal{F}$ is represented as a directed acyclic graph (DAG), where each node is a Boolean operator, a relational operator, an arithmetic operator, a term-ITE, or an atom. The conversion is applied to each relational operator in the DAG, and the procedure ends when $\mathcal{F}'$ no longer contains term-ITEs. The main function starts by selecting the candidates for the conversion in the DAG. Each candidate is a relational operator that has a term-ITE as a descendant, and the candidates are gathered in F. As Line 4 in Fig. 4.5 shows, the term-ITE conversion is invoked with $f \in F$, and all the term-ITEs are removed from f. After the conversion of f, the converted formula f' is either a Boolean ITE or an atom. The procedure ends when all $f \in F$ have been considered. At that point, $\mathcal{F}$ has been converted into $\mathcal{F}'$, which does not contain any term-ITEs.

As TermIteConversion is invoked with $f \in F$, a cofactoring variable v is searched for in f at Line 10. We select an atom as a cofactoring variable that resides in the conditional term of the term-ITE. With v, we recursively compute the cofactor of f. In general, the cofactors are computed for the children of f with respect to v, and a new formula f_v is created with new children. As shown in Line 38 of Fig. 4.6, if f is a relational operator, we compute the cofactors l_v and r_v for the children of f. After computing the cofactors, we check for simple cases with l_v and r_v. The simple check detects terminal cases for the terms l_v and r_v with respect to the type $(=, <, \leq, >, \geq)$ of f. Figure 4.4 shows an example of simplification. If a terminal case is not found, a new formula f_v is generated with $\mathrm{type}(f)$, l_v and r_v. The newly generated formula, f_v is either an atom or a relation operator with term-ITEs. In the latter case, term-ITE conversion is called with f_v, again. In Line 47 of Fig. 4.6, if f_v is an atom, theory reasoning is done with v. As Theorem 4.4 shows, if $v \models_T f_v$, then f in Line 13 of Fig. 4.5 is simplified to $v \vee f_{\neg v}$. Likewise, if $v \models_T \neg f_v$, then f is simplified to $\neg v \wedge f_{\neg v}$. When f is either a term-ITE or a Boolean ITE, the cofactor for each term of f is

```
TermIteConversionMain ($\mathcal{F}$) {
    $F$ := GatherCandidateForTermIteConversion ($\mathcal{F}$);
    for (each $f \in F$ in topological order) {
        $f'$ := TermIteConversion ($f$);
        $\mathcal{F}'$ := UpdateFormula ($\mathcal{F}$, $f'$);
    }
    return $\mathcal{F}'$;
}

TermIteConversion ($f$) {
    while ( $v$ := GetCofactorVariable ($f$) ) {
        $f_v$ := CofactorRecur ($f$, $v$);
        $f_{\neg v}$ := CofactorRecur ($f$, $\neg v$);
        $f$ := Ite ($v$, $f_v$, $f_{\neg v}$);
    }
    return $f$;
}

CofactorRecur ($f$, $v$) {
    if ( $f$ = $v$ ) {
        $f_v$ := $\top$;
    } else if ( $f$ = $\neg v$ ) {
        $f_v$ := $\perp$;
    } else if ( is_relation($f$) ) {
        $f_v$ := CofactorRelRecur ($f$, $v$);
    } else if ( is_term_ite($f$) ) {
        $f_v$ := CofactorTiteRecur ($f$, $v$);
    } else { /* $+, -, \times$ */
        $C$ := children($f$);
        For each $c \in C$ {
            $d$ := CofactorRecur ($c$, $v$);
            Add($D$, $d$);
        }
        $f_v$ := NewFormula (type($f$), $D$); /* type($f$) is either $+, -, \times$. */
        SimplifyArithFormula($f_v$);
    }
    return $f_v$;
}
```

Figure 4.5: Term-ITE conversion algorithm

```
CofactorRelRecur ($f$, $v$) {
    $l_v$ := CofactorRelRecur ($\mathrm{left}(f)$, $v$);
    $r_v$ := CofactorRelRecur ($\mathrm{right}(f)$, $v$);
    $f_v$ := SimpleCheckWithTerms (type($f$), $l_v$, $r_v$);
    if ( $f_v$ = NoSimplification ) {
        $f_v$ := NewFormula (type($f$), $l_v$, $r_v$);
        if ( is_term_ite($l_v$) $or$ is_term_ite($r_v$) ) {
            $f_v$ = TermIteConversion ($f_v$);
        }
    }
    if ( is_atom($f_v$) ) {
        if ( $v \models_T f_v$ ) { /* theory reasoning */
            $f_v$ := $\top$
        } else if ( $v \models_T \neg f_v$ ) { /* theory reasoning */
            $f_v$ := $\bot$
        }
    }
    return $f_v$;
}

CofactorTiteRecur ($f$, $v$) {
    $f_c$ := CondTerm($f$);  $f_t$ := ThenTerm($f$);  $f_e$ := ElseTerm($f$);
    if ( $f_c$ = $\top$ ) {
        return CofactorRecur ($f_t$, $v$);
    } else if ( $f_c$ = $\bot$ ) {
        return CofactorRecur ($f_e$, $v$);
    } else if ( is_pred($f_c$) ) {
        if ( $v \models_T f_c$ ) { /* theory reasoning */
            return CofactorRecur ($f_t$, $v$);
        } else if ( $v \models_T \neg f_c)$ ) { /* theory reasoning */
            return CofactorRecur ($f_e$, $v$);
        }
    }
    $c_v$ := CofactorRecur ($f_c$, $v$);
    $t_v$ := CofactorRecur ($f_t$, $v$);
    $e_v$ := CofactorRecur ($f_e$, $v$);
    $f_v$ := Ite ($c_v$, $t_v$, $e_v$);
    return $f_v$;
}
```

Figure 4.6: Term-ITE conversion algorithm

computed as shown in Line 58 of Fig. 4.6. As in the cofactoring on the relational operator, a terminal case is checked for the conditional term f_c. If f_c is an atomic predicate, theory reasoning is done with v and f_c using Rules 4.4–4.5 of Sect. 4.2.2. If a terminal case is not found, then the cofactors for the terms of f are computed to obtain f_v.

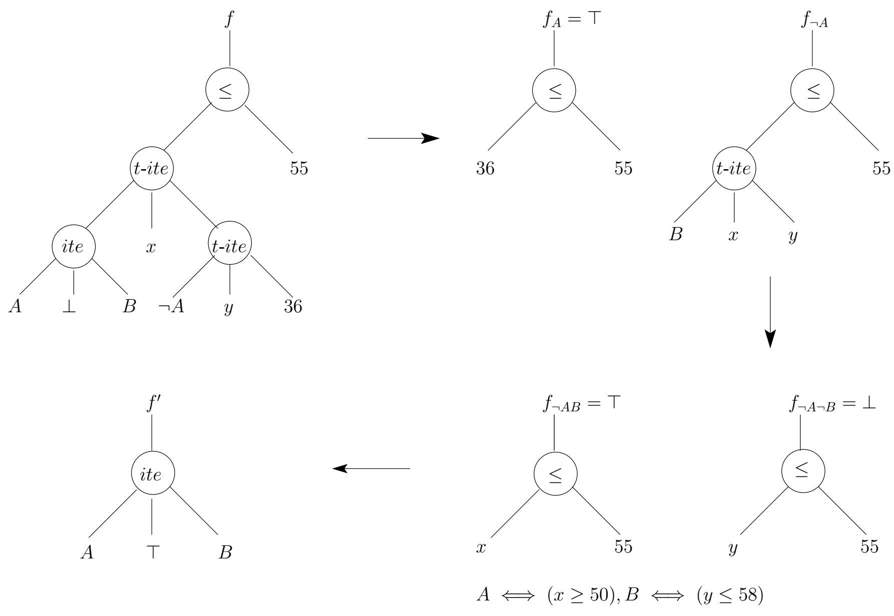


Figure 4.7: Term-ITE conversion

Example 4.6. *If f is a relational operator such that $D(f)$ contains term-ITEs, we convert f into f' such that there is no term-ITE in $D(f')$. In Fig. 4.7, let $A \leftrightarrow (x \geq 50)$ and $B \leftrightarrow (y \leq 58)$. We first traverse $D(f)$ to find a cofactoring variable. We pick an atomic formula A as cofactoring variable and compute the cofactors of f with respect to A. As we proceed, $f_A = (36 \leq 55) = \top$ and $f_{\neg A}$ is constructed with a new term-ITE. Since there still exists a term-ITE in $D(f_{\neg A})$, we look for another cofactoring variable in $f_{\neg A}$. We select B and compute the cofactors for $f_{\neg A}$. As a result, we get $f_{\neg AB} = (x \leq 55)$ and $f_{\neg A \neg B} = (y \leq 55)$. Since $A \models_T f_{\neg AB}$ and $\neg B \models_T \neg f_{\neg A \neg B}$, $f_{\neg AB} = \top$ and $f_{\neg A \neg B} = \perp$. Finally, the converted formula f' gets reduced to $ite(A, \top, B)$ as shown in Fig. 4.7.*

4.5 Related Work

Early references on the treatment of ITEs are [Kar88], [BRB90] and [JDB95]. For SMT preprocessing, HTP [Roe06] introduces several preprocessing techniques such as unate predicate detection, variable substitution and symmetry breaking. Yices [DdM06a, DdM06b] uses a Gaussian elimination to reduce the size of initial tableau of equality constraints. In [YM06], Yu **et al.** describes a static learning technique that analyzes the relationship of the linear constraints. In Karplus's technical report [Kar88], a new canonical form for *ITE* DAGs is introduced using two-cuts, and *ITE* normalization using recursive transformation is shown in [NO08].

4.6 Experimental Results

We have implemented the algorithm presented in Sect. 4.4 in Sateen [KJS07b, KJR$^+$08, VIS], a theorem prover for quantifier-free first-order logic that combines the propositional reasoning engine of [JHS05, JS05] with theory-specific procedures. Experiments are done with the full set of QF_LIA (Quantifier free linear integer arithmetic logic) benchmarks from SMT-COMP (Satisfiability Modulo Theories Competition) [SMTa]. The experiments were performed on an Intel 2.4 GHz Quad Core with 4 GB of RAM running Linux. Time out was set at 1000 seconds. Sateen was compared with Z3.2 [SMTa], MathSAT-4.2[BBC$^+$05b, SMTa] and Yices-1.0.16 [Yic]. Z3.2 and MathSAT-4.2 are the ones that were submitted to SMT-COMP in 2008. We used most recent version of Yices that is available.

In QF_LIA benchmarks, there are two benchmark sets, **nec-smt** and **rings**, that are rich in term-ITE operators. More than 90 percent of the QF_LIA benchmarks belong to those two sets. The instances in the **nec-smt** set are generated by the SMT-based BMC engine of F-Soft [IYG$^+$05]; the instances in **rings** encode associativity properties on modular arithmetic.

Figures 4.8–4.10 show scatterplots comparing Z3, MathSAT and Yices to Sateen. Points below the diagonal represent wins for Sateen. Each scatterplot shows two lines: The main diagonal, and $y = \kappa \cdot x^{\eta}$, where κ and η are obtained by least-square fitting. Figure 4.8 shows that Sateen is often an order of magnitude faster than Z3. In Fig. 4.9 and 4.10, Sateen is often a few orders of magnitude faster than

Table 4.1: Number of term-ITE reduction with simple preprocessing

Benchmark	**Before S.P.**	**After S.P.**	**rate(%)**
bftpd_login/prp-74-50.smt	38773	34085	12
checkpass/prp-10-46.smt	17240	14949	13
checkpass/prp-63-50.smt	25376	21893	14
checkpass_pwd/prp-38-42.smt	12196	10354	15
getoption/prp-2-200.smt	11269	9791	13
getoption_directories/prp-0-110.smt	72892	62457	14
getoption_group/prp-72-49.smt	15021	12094	20
handler_sigchld/prp-20-46.smt	7800	6824	13
int_from_list/prp-34-41.smt	7184	5888	18
user_is_in_group/prp-23-48.smt	22549	17939	20

MathSAT and Yices.

We further evaluated our preprocessor by generating simplified formulae from the **nec-smt** benchmarks and running Z3, MathSAT, and Yices on them. All solvers took less than a second on each simplified problem. Figures 4.11–4.13 show scatterplots comparing Z3, MathSAT and Yices with preprocessor and without preprocessor. The times for the solvers with preprocessor include preprocessing time. As Figures 4.11–4.13 show, our preprocessor is also effective for other solvers.

Table 4.1 shows the number of term-ITE reductions with the simple preprocessing on randomly selected benchmarks. The first column gives the name of the benchmarks, the second one is the initial number of term-ITEs, and the third one is the number of term-ITEs after the simple preprocessing. The last column gives the rate of the reduction. On average, we achieved 15% term-ITE reduction with the simple preprocessing of Section 4.3.

Finally, we compared our approach to the naive approach of Eq. 4.2. As Fig. 4.15 shows, our approach is significantly better. In addition, we disabled theory simplification in the algorithm and ran the experiment on the problems where the simplifications play a significant role. Figure 4.14 shows that Sateen with theory simplification is consistently better than the one without simplification.

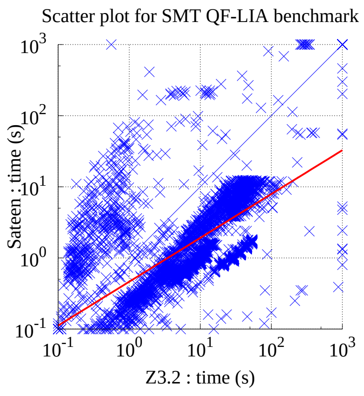


Figure 4.8: Z3 vs. Sateen on QF_LIA

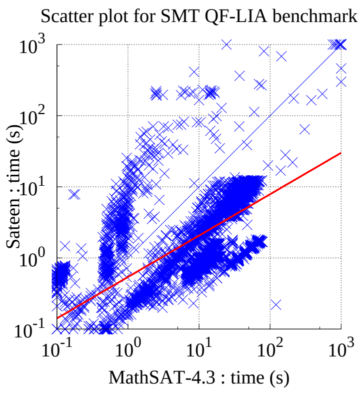


Figure 4.9: MathSAT vs. Sateen on QF_LIA

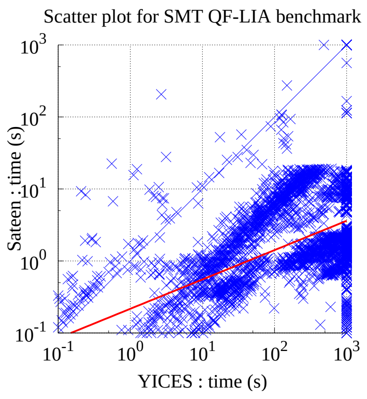


Figure 4.10: Yices vs. Sateen on QF_LIA

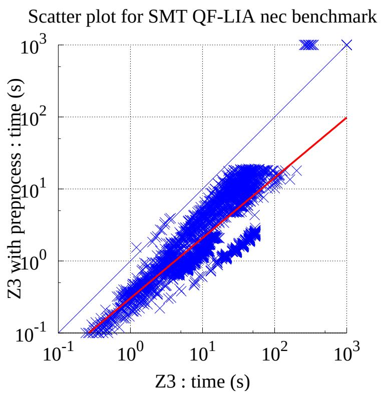


Figure 4.11: Z3 with preprocess vs. Z3 on QF_LIA

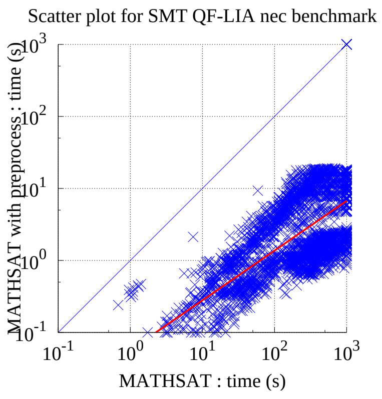


Figure 4.12: MATHSAT with preprocess vs. MATHSAT on QF_LIA

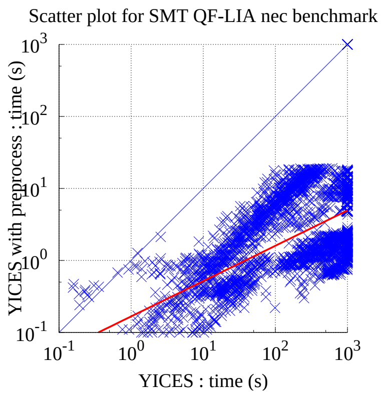
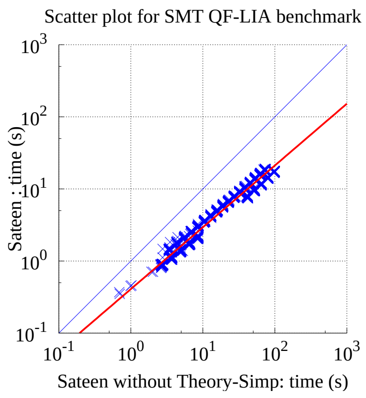


Figure 4.13: YICES with preprocess vs. YICES on QF_LIA



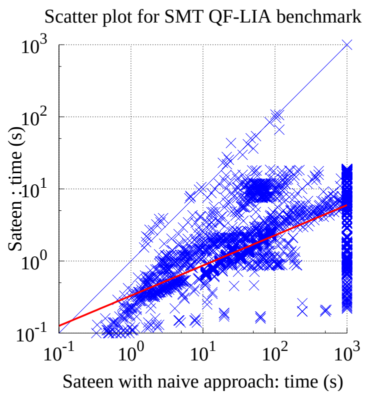


Figure 4.14: SATEEN vs. Sateen without Theory-Simp on QF_LIA

Figure 4.15: SATEEN vs. Sateen with naive approach on QF_LIA

4.7 Conclusions

We have presented an algorithm for the term-ITE conversion in first-order theories like the theory of linear arithmetic. The approach is based on the computation of cofactors and theory simplification. The simplification is done by detecting special cases in the formula or using theory propagation on the atomic predicates. Experiments show that this approach is very effective in most QF_LIA benchmarks and often speeds up SMT solvers. On the other hand, since our approach may still blow up in general, we are working on combining it with a less aggressive approach, based on (4.1), that does not blow up.

Chapter 5

Avoiding Mismatches in Verification of Verilog Designs

5.1 Introduction

There have been numerous efforts to put the Verilog hardware description language (HDL) on a rigorous semantic basis for simulation, synthesis, and formal verification. On the one hand, several different semantics have been proposed to describe the execution of a subset of Verilog. On the other hand, Verilog coding guidelines have been practically used to avoid the mismatches between pre- and post-synthesis simulations.

A verification condition for a Verilog design may be described in terms of event semantics; however, expressing the event semantics in a logical formula often leads to a complex condition to verify. On the other hand, cycle-based semantics describe the execution of Verilog in terms of sequences of stable states attained in every clock cycle. With cycle-based semantics, we show that a concise verification condition for a hardware model may be generated that captures exactly the set of execution trace that may be produced by a standard-compliant simulator. In the past, several cycle-based approaches have been proposed; however, the semantics are often not completely defined and do not guarantee to avoid the mismatches between the verification condition and the simulation of the model.

We define a subset of Verilog that describes synchronous hardware under appropriate semantic restrictions. The restrictions are compatible with common coding guidelines. They guarantee that formal verifiers, simulators, and synthesis tools all interpret a model in the same way. (We prove behavioral equivalence between the verification condition and the simulation model.) The restrictions allow controlled nondeterminism, which is useful for high-level verification, but can be easily eliminated for synthesis. Finally, they lead to a concise verification condition for the model as an SMT formula.

The rest of this chapter is organized as follows. Section 5.2 presents a subset of Verilog called **MSV**. Section 5.3 proves the correctness of our translation. Section 5.4 discusses strengths and limitations. After a survey of related work in Sect. 5.5, conclusions are offered in Sect. 5.7.

5.2 Verification Conditions for Hardware

In this section, we define a language **MSV** (Minimal Synchronous Verilog) that is a subset of the Verilog hardware description language (HDL) [Ver] suitable for the modeling of synchronous hardware. A description in **MSV** consists of a single module that contains variable declarations and procedural blocks. A variable x with a width of $n \in Z^+$ bits can be of type **input** or **reg**, and a variable of type **reg** can be designated as output. A constant c is a natural number; expressions are made of variables, constants, and operators, which are categorized into arithmetic, concatenation, reduction, bit-selection, shift, bit-wise, logical, conditional, and relational operators. All Verilog operators are supported except case equality (===) and inequality (!==). Although the subset we consider includes essential features of Verilog, it does not support delays, strengths, and other features that are not needed for RTL verification of synchronous designs.

In **MSV**, as in Verilog, a blocking assignment (=) updates the target variable immediately, while the update of a nonblocking assignment (⇐) is deferred. A statement may be an assignment, an **if / else** conditional statement, or a sequence of statements enclosed by the keywords **begin** and **end**. A procedural block consists of a trigger and a statement.

Procedural block triggers are restricted to three types in **MSV**:

- **always** $@ *$
- **initial** $\#0\ \#0$
- **always** $@$ (**posedge clock**)

The purpose of **always** $@ *$ blocks is to describe combinational logic, while **initial** $\#0\ \#0$, and **always** $@$ (**posedge clock**) blocks are used to describe the initial values and the updates of memory elements. In **always** $@$ (**posedge clock**), **clock** is a distinguished input.

Valid **MSV** descriptions obey semantic constraints, some of which are best described in terms of an intermediate form. Let V be the set of variables in a description. Let $\bar{V}$ be a set of variables of type **reg** disjoint from V and let $\beta : V \to \bar{V}$ be an injective function. We write $\bar{v}$ for $\beta(v)$. An **MSV** description is put in intermediate form by replacing each nonblocking assignment $a \Leftarrow b$ with $\bar{a} = b$. The result is converted into Static Single Assignment (SSA) form [CFR+89].

The example in Fig. 5.1 shows the conversion from **MSV** description to the intermediate form. The three procedural blocks at the top are converted into the intermediate form at the bottom by replacing each nonblocking assignment with a blocking assignment (i.e, from $z \Leftarrow y$ to $\bar{z} = y$), and the result is converted into SSA form. We assume henceforth that descriptions are in intermediate form.

Let B_C be the set of combinational blocks of type **always** $@*$. Let B_A be the set of sequential blocks of type **always** @(**posedge clock**), B_I be the set of initial blocks of type **initial** $\#0\ \#0$, and $B_S = B_A \cup B_I$ be the set of sequential blocks. Let $V_C \subseteq V$ be the set of target variables in B_C and $V_S \subseteq V$ be the set of target variables in B_S. Let $V_A \subseteq V_S$ be the set of target variables in B_A, and $V_I \subseteq V_S$ be the set of target variables in B_I where $V_S = V_A \cup V_I$. Let $V_R \subseteq V \cup \bar{V}$ be the set of variables of type **reg**. We define several terms useful to describe the semantics of **MSV**.

Definition 5.1. *The condition for an assignment is the predicate that has to be true for the assignment to execute.*

Definition 5.2. *Let V_C^+ be the set of variables in the intermediate form of B_C. The dependency graph for B_C is a directed graph $G_D = (V_C^+, E)$. If an assignment α has a target variable $d_i \in V_C^+$ and if a variable $s_j \in V_C^+$ appears in the right-hand side of α, or in the condition of an* ***if / else*** *statement containing α, then $(s_j, d_i) \in E$.*

We impose the following restrictions for B_C and B_S.

(1) $V_C \cap V_S = \emptyset$.

(2) The dependency graph G_D is acyclic.

```
initial $\#0$ $\#0$ $z = 0$;

always @(posedge clk) begin
    $z \Leftarrow 0$;
    if ($v$)
        $z \Leftarrow 1$;
end
```

```
always @(posedge clk) begin
    if ($u$)
        $z = x$;
    else
        $z \Leftarrow y$;
    $w = z$;
end
```

```
initial $\#0$ $\#0$ $z_1 = 0$;

always @(posedge clk) begin
    $\bar{z}_1 = 0$;
    if ($v_0$)
        $\bar{z}_2 = 1$;
    $\bar{z}_3 = \phi(\bar{z}_2, \bar{z}_1)$;
end
```

```
always @(posedge clk) begin
    if ($u_0$)
        $z_2 = x_0$;
    else
        $\bar{z}_4 = y_0$;
    $z_3 = \phi(z_2, z_1)$;
    $\bar{z}_5 = \phi(z_1, \bar{z}_4)$;
    $w_1 = z_3$;
end
```

Figure 5.1: Conversion from **MSV** description to intermediate form

(3) Let B_C^+ be the set of blocks in B_C in intermediate form. If variable v_i is defined in $b \in B_C^+$ and v_j is used in b, $j \geq i$. If a variable v is in V_R, v_0 is not used in the intermediate form of $B_C \cup B_S$.

(4) If a target variable $v \in V_C$ occurs in a block $b_1 \in B_C$, v does not occur as a target in another block $b_2 \in B_C$.

(5) All the assignments to a variable v in a block $b \in B_C$ are of the same type: either all blocking, or all nonblocking.

(6) If a target variable $v \in V_A \setminus \bar{V}$ is in $b_1 \in B_A$, v cannot be used in $b_2 \in B_A$ or $b_3 \in B_C$.

(7) If a target variable $v \in V_I \setminus \bar{V}$ is in $b_1 \in B_I$, v cannot be used in $b_2 \in B_I$ or $b_3 \in B_A$.

We impose these restrictions to enable B_C to describe combinational logic and to allow B_S to have nondeterminism that can be easily controlled by the designer. In particular, Restrictions 1, 2, 3, 4, and 5 enable B_C to describe combinational logic. Restrictions 6 and 7 are imposed to limit nondeterministic behavior caused by the interleaving of sequential blocks. The restrictions are compatible with common design guidelines [Cum02] used in industry (e.g., blocking assignments for combinational logic and nonblocking assignments for memory elements) and allow us to produce concise verification conditions. The role of each restriction is made clear in Sect. 5.3.

The semantics of **MSV** descriptions complying with the restrictions above are defined with respect to a finite state machine. Let $V_S' = \{v_1', \ldots, v_m'\}$ be the primed version of $V_S = \{v_1, \ldots, v_m\}$, where V_S and V_S' are the current and next state variables in B_S. Let $W = \{w_1, \ldots, w_p\}$ be the variables of type **input** and $Z = \{z_1, \ldots, z_n\} \subseteq V_C \cup V_S$ be the variables that are designated as output. A finite state model is a 7-tuple $\langle V_S, W, V_S', Z, I, T, Q\rangle$, where $I(V_S)$ is the initial state predicate, $T(V_S, W, V_S')$ is the transition relation, and $Q(V_S, W, Z)$ is the output relation. The initial state predicate is defined by

$$I(V_S) = \exists W \,.\, \varphi(V_S, W) \ , \tag{5.1}$$

where

$$\varphi(V_S, W) = \textstyle\bigwedge_{1 \leq i \leq m} (v_i = \rho_i(W)) \ , \tag{5.2}$$

that is, the initial value of each state variable is a function of the input variables. The transition relation is defined by

$$T(V_S, W, V_S') = \exists V_C \,.\, \textstyle\bigwedge_{1 \le i \le m} (v_i' = \delta_i(V_S, V_C, W)) \ , \tag{5.3}$$

that is, the next value of each state variable is a function of the current state and the input. The output relation is defined by

$$Q(V_S, W, Z) = \textstyle\bigwedge_{1 \le i \le n} (z_i = \gamma_i(V_S, W)) \ , \tag{5.4}$$

that is, the value of each output variable is a function of the current state and the input variables.

We use the intermediate form of an **MSV** description that is in SSA form to derive BV formulae for (5.2), (5.3), and (5.4). The SSA form of a sequential program enables us to convert each assignment in the program into an equality with an enabling condition. In contrast to a sequential program, an **MSV** description contains multiple blocks and two different types of assignments. In the following, we describe how to generate the BV formulae that describe the conflict arbitration of two different types of assignments in multiple blocks.

Let $B_B \subseteq B_A$ be the set of blocks that contain blocking assignments to a state variable $v \in V_A$; let $B_N \subseteq B_A$ be the set of blocks that contain nonblocking assignments to v. Suppose $|B_A| = k$, $|B_B| = r$, and $|B_N| = s$. We generate r equalities for v and s equalities for $\bar{v}$. In each $b_j \in B_B$, we introduce a new variable v_j for v and generate a BV equality for v that is defined by

$$v_j[n] = \textit{tite}(c_1, e_1[n], \textit{tite}(c_2, e_2[n], \ldots, \textit{tite}(c_p, e_p[n], v[n]))) \ , \tag{5.5}$$

where each c_i ($1 \le i \le p$) is a condition to assign e_i to v_j by a blocking assignment. Likewise, for each $b_k \in B_N$, we introduce a new variable $\bar{v}_k$ for $\bar{v}$ and generate a BV equality for $\bar{v}$,

$$\bar{v}_k[n] = \textit{tite}(d_1, f_1[n], \textit{tite}(d_2, f_2[n], \ldots, \textit{tite}(d_q, f_q[n], v[n]))) \ , \tag{5.6}$$

where each d_i ($1 \le i \le q$) is a condition to assign f_i to $\bar{v}_k$ by a nonblocking assignment. Finally, we generate a BV formula

$$\begin{aligned} &\textit{ite}(\textstyle\bigvee_{1 \le k \le s} D_k, \bigvee_{1 \le k \le s}(D_k \wedge v'[n] = \bar{v}_k[n]), \textit{ite}(\bigvee_{1 \le j \le r} C_j, \\ &\qquad\qquad \textstyle\bigvee_{1 \le j \le r}(C_j \wedge v'[n] = v_j[n]), v'[n] = v[n])) \ , \end{aligned} \tag{5.7}$$

where each D_k is the disjunction of conditions to assign to $\bar{v}_k$ in each $b_k \in B_N$ and each C_j is the disjunction of conditions to assign to v_j in each $b_j \in B_B$. The formula describes the conflict arbitration among r blocking and s nonblocking assignments to v in B_A, where the nonblocking assignment takes precedence over the blocking assignment. The formula (5.7) conjoined with r equalities generated by (5.5) and s equalities generated by (5.6) is the transition relation for v.

The BV formula for each state variable is generated and the conjunction of these formulae is the transition relation that is equivalent to (5.3). The output relation (5.4) and (5.2) of the initial state predicate are generated in a similar manner.

Continuing the example of Fig. 5.1, suppose $u, v \in V_B(1)$ and $w, x, y, z \in V_B(4)$. For the target z, we generate BV formulae

$$(\bar{z}_1[4] = 0[4]) \wedge \textit{ite}(v_0[1] = 1[1], \bar{z}_2[4] = 1[4] \wedge \bar{z}_3[4] = \bar{z}_2[4], \bar{z}_3[4] = \bar{z}_1[4]) \tag{5.8}$$

in the first procedural block and

$$\textit{ite}(u_0[1] = 1[1], z_2[4] = x_0[4], \bar{z}_4[4] = y_0[4]) \wedge (z_3[4] = \textit{tite}(u_0[1] = 1[1], z_2[4], z_1[4])) \wedge \\ (\bar{z}_5[4] = \textit{tite}(u_0[1] = 1[1], z_1[4], \bar{z}_4[4])) \tag{5.9}$$

in the second procedural block. Then, we generate an *ite* formula *ite*$(\top \vee v_0[1] = 1[1] \vee u_0[1] = 0[1], z'[4] = \bar{z}_3[4] \vee (u_0[1] = 0[1] \wedge z'[4] = \bar{z}_5[4]), z'[4] = z_3[4])$, which is simplified to $z'[4] = \bar{z}_3[4] \vee (u_0[1] = 0[1] \wedge z'[4] = \bar{z}_5[4])$. The simplified formula conjoined with (5.8) and (5.9) is the transition relation for z where z_1 is the current state variable.

5.3 Correctness

In this section we show that for an **MSV** module operated in synchronous mode, the set of behaviors that may be produced by a standard-compliant Verilog simulator [IEE06] that satisfies an atomicity requirement to be introduced shortly is captured by the BV formulae described in (5.2), (5.3), and (5.4).

The assumption of synchronous operation is enforced by having a suitable Verilog test bench drive the module under consideration. A template for the test bench is shown in Fig. 5.2. It consists of the

```
module Testbench;
        <input declaration list>; // e.g., reg [2:0] a; reg [3:0] b; ...
        <output declaration list>; // e.g., wire q; wire [4:0] r; ...
        reg clock;
        initial begin
                clock = 0;
                #0 <input list> = inputF (0);
                $strobe($time,<input list>,<output list>);
                #1 forever begin
                        clock=0;
                        <input list> = inputF (0);
                        $strobe($time,<input list>,<output list>);
                        #1 clock=1;
                        #1;
                end
        end
        function [NBITS-1:0] inputF (input dummy);
        begin: _inputF
                // returns input values for current $time
        end
        endfunction
        dut dut0 (clock,<input list>,<output list>);
endmodule
```

Figure 5.2: Verilog code for a test bench

instantiation of the **MSV** module **dut**, the declarations of its inputs and outputs, a variable **clock**, an **initial** block that applies stimulus to **dut** and samples its outputs, and a function **inputF**, which produces the input values—either 0 or 1, but not $\times$.

At simulation time $t = 0$, the simulator sets the **clock** to zero in the **initial** block. The simulator then calls the function **inputF** that generates new inputs. The target variables in **initial** $\#0$ $\#0$ blocks of **dut** are updated first and the target variables in combinational blocks of **dut** are updated to reflect the new inputs and the updates. The zero delays in the **initial** block of the test bench and in **initial** $\#0$ $\#0$ blocks of **dut** impose the update order. After the updates, the **strobe** task reports the values of the inputs and the outputs of **dut**. Then the simulation time is increased to $t = 1$ and the the **forever** loop is evaluated: the **clock** stays at zero and the function **inputF** generates new inputs. The target variables in combinational blocks of **dut** are updated to reflect the new inputs. After the updates, the **strobe** task reports the values of the inputs and the outputs of **dut**. Now, the simulation time is increased to $t = 2$: the **clock** is changed to one and all the sequential blocks in **dut** are triggered, causing updates of their target variables. The evaluation of the **forever** loop repeats as the simulation proceeds. In this simulation environment just described, the following lemma holds.

Lemma 5.3. *Every variable of* ***dut*** *attains a stable value at every simulation time; hence the simulation time always advances.*

Proof. By Restriction 2, there is no cycle in the dependency graph of combinational logic. By induction, the number of evaluations of each combinational block is finite because it only depends on the finite number of changes on its inputs. Hence, the outputs of each combinational block stabilize.

On the other hand, since all the sequential blocks are evaluated only once, there is a finite number of update events in the sequential blocks. As a result, simulation time can always advance. $\square$

Corollary 5.4. *Just before the time is advanced from t to $t + 1$ ($t = 0, 1, \ldots$), there is only one evaluation event scheduled and it is for the always block in the test bench.*

Proof. Since there is no delay in **dut**, the only event scheduled before the time is advanced from t to $t + 1$ is the evaluation event of the always block in the test bench. $\square$

```
1   initial #0 #0 begin              9    always @ * begin
2       $monitor ($time, a, b, c);   10       d = b;
3       a = 0;                       11       c = a ^ d;
4       b = 0;                       12   end
5       #1 a = 1;
6       b = 1;
7       #1 $finish;
8   end
```

Figure 5.3: Nondeterministic behavior of Verilog simulation (a)

Thanks to Lemma 5.3, the notions of initial and final values of a variable at a certain time are well defined.

Definition 5.5. *The* ***initial value*** *of a variable $v \in V_C \cup V_S$ at time t is the value of v when time advances to t. The* ***final value*** *of v at time t is the value of v immediatley before time advances to $t + 1$.*

While Lemma 5.3 shows that the **dut** model evolves from one stable state to another, it says nothing about what states may be produced. Standard-compliant simulators are allowed to produce different results for a variety of reasons. While the ability to describe nondeterministic behavior is sometimes an advantage, it also poses significant challenges to designers and tool implementors.

According to the standard, assignments in different always blocks that are triggered simultaneously may be interleaved arbitrarily, as long as sequential consistency is preserved. If at time t the simulator executes at least one nonblocking assignment to v, the final value of v at time t is assigned by the nonblocking assignment that is executed last. Otherwise, if any blocking assignment to v is executed, the final value of v is assigned by the blocking assignment that is executed last. If no assignment to v is executed, the final value of v is its initial value. For example, the value of c in Fig. 5.3 is either 0 or 1 at time 1. If the assignments are executed in the order of $5 \rightarrow 10 \rightarrow 6 \rightarrow 11$, the value of c is 1; if they are executed in the order of $5 \rightarrow 6 \rightarrow 10 \rightarrow 11$, the value of c is 0.

This freedom to interleave processes makes it very difficult for designers to describe behavior that is unambiguously combinational. In practice, most synthesis tools partly limit and partly ignore the nondeter-

```
always @ * begin
    $a = 0$;
    if (b & c) $a \Leftarrow 1$;
end
```

Figure 5.4: Nondeterministic behavior of Verilog simulation (b)

ministic behavior of a model and always derive a deterministic netlist. Ignoring a nondeterministic behavior in the synthesis tools may cause pre- and post-synthesis simulation mismatches [MC99]; however, this is often not detected since most simulators execute blocks atomically. If a block is not executed atomically, the user of a synthesis tool may have a problem describing combinational logic, and the verification condition for a model may get unnecessarily cumbersome; hence, we impose an atomicity rule.

Definition 5.6 (Atomic evaluation)**.** *A block, either **initial** or **always**, conforming with **MSV** is **evaluated atomically** if the simulator executes the events in the block without any suspension until it reaches the end of the block.*

Assumption 1 (Atomicity rule)**.** *Every block conforming with **MSV** is evaluated atomically.*

In practice, most standard-compliant simulators implement the atomicity rule and most synthesis tools assume it for models restricted to **MSV**.

Although the atomicity rule prevents some of the undesired outcomes, a simulator may still generate a nondeterministic outcome without restriction 5, which says that the assignments in a combinational block are either all blocking, or all nonblocking. If different assignment types are allowed, the block may not describe combinational logic. In Fig. 5.4, suppose the value of b changes from 0 to 1 first, and the value of c then changes from 1 to 0 at the same simulation time. Then, the always block may be evaluated twice, with the nonblocking assignment $a \Leftarrow 1$ executed in the first evaluation, but not in the second. The value of a stabilizes to 1, which is assigned by the nonblocking assignment. Since the value of a is determined by the unstable input, the combinational block does not describe combinational logic. Restriction 5 prevents this.

Given the simulation environment for **dut**, we show that the relation between the input and output values extracted by the **strobe** task and the values of the state variables in the simulation is captured by the BV formulae described in (5.2), (5.3), and (5.4). As we assumed for generating the BV formula, the only possible source of nondeterminism in simulation is the interleaving of sequential blocks. A variable of type **reg** may have an initial value $\times$ if it is not assigned in an **initial** block. Restriction 3 guarantees that the initial value $\times$ is not propagated to other variables; hence we can ignore the value $\times$.

We first show that the final value of each target variable in a combinational block is uniquely determined by the final values of the inputs to that block. Then, we show that the value assigned to each target variable by a sequential block is uniquely determined by the initial values of the inputs to that block, and the final value of each target in sequential blocks may be any of the values assigned by the blocks that assign to the target. This argument is captured in the following lemmas.

Lemma 5.7. *The final value of a variable $v \in V_C$ at time t is uniquely determined by the final values at time t of the inputs to the unique block $b \in B_C$ that assigns to v.*

Proof. By Lemma 5.3, every target variable in a combinational block attains a stable value. By the atomicity rule, when the input of a combinational block changes, the block is always triggered; hence the last evaluation of the block occurs after all the inputs attain the stable values. By Restriction 3, every target variable in a combinational block gets assigned whenever the block is evaluated. According to the standard, the sequential order of the nonblocking (blocking) assignments in a block is preserved when they are executed by a simulator; hence by Restriction 5, the update event during the last evaluation is not superseded by any subsequent update. By Restriction 4, b is the only block that assigns to v. $\square$

Lemma 5.8. *The final value assigned by a block $b \in B_A$ to a target $v \in V_A$ at time t is uniquely determined by the initial values of the inputs to b at time t.*

Proof. By Restrictions 6 and 7, a target variable in a sequential block is not dependent on the target assigned by a blocking assignment in another sequential block, and the value of a target assigned by a nonblocking assignment is only available in the next clock cycle; hence the inputs of a sequential block do not change while the block is evaluated and the execution order of the blocks does not affect the value of the target in the block. Since every sequential block is evaluated once in one clock cycle, the final value assigned by a block $b \in B_A$ to $v \in V_A$ is uniquely determined by the initial values of the inputs. □

Lemma 5.9. *Let $B_N \subseteq B_A$ be a set of sequential blocks such that each $b \in B_N$ contains a nonblocking assignment to a target $v \in V_A$ that is executed by the simulator at time t. Let $B_B \subseteq B_A$ be a set of sequential blocks such that each $b \in B_B$ contains a blocking assignment to v that is executed by the simulator at time t. If $B_N \neq \emptyset$, the final value of v is the final value assigned by one of the blocks in B_N. If $B_N = \emptyset$ and $B_B \neq \emptyset$, the final value of v is the final value assigned by one of the blocks in B_B. If $B_N \cup B_B = \emptyset$, the final value of v is its initial value.*

Proof. By Lemma 5.8, each $b \in B_N \cup B_B$ assigns a final value to v. In addition, all the blocks in $B_N \cup B_B$ are triggered at the same time and can be evaluated in arbitrary order by the simulator. Therefore, any block in $B_N \cup B_B$ can be the last block that is evaluated. If $B_N \neq \emptyset$, the final value of v is the final value assigned by the block that is executed last among the blocks in B_N; hence the final value of v is the final value assigned by one of the blocks in B_N. If $B_N = \emptyset$ and $B_B \neq \emptyset$, the final value of v is assigned by the block that is executed last among the blocks in B_B; hence the final value of v is the final value assigned by one of the blocks in B_B. If $B_N \cup B_B = \emptyset$, no block assigns a final value to v; hence the final value of v is its initial value. □

Now, we describe a cycle-based transition relation that is valid under the simulation environment. At every clock cycle when the **strobe** task is evaluated, the values of new inputs and the outputs are reported and the current state values of the state variables can be extracted from **dut**. The next state value of each state variable is determined by the new inputs and the current state variables, and the cycle-based transition relation is the conjunction of the transition relations for the state variables. The initial value of each state variable is determined by the new inputs, and the cycle-based initial state predicate is the conjunction of the initial state predicates for the state variables.

Theorem 5.10. *The transition relation between the input and output values extracted by the* **strobe** *task and the values of the state variables in each clock cycle is captured by the* BV *formula of* **dut***, which is the conjunction of* (5.3) *and* (5.4).

Proof. By Lemma 5.8 and Lemma 5.9, the next state value of a state variable v is determined by new inputs and the current state variables. In a sequential block $b \in B_A$ that contains an assignment to v, a cycle based transition relation for v in b is defined. If all the assignments to v in b are blocking, the transition relation is equivalent to (5.5). If there exists a nonblocking assignment to v in b, the transition relation for $\bar{v}$ is defined by (5.6) where $\bar{v} = \beta(v)$. If there are r blocks that contain blocking assignments to v and s blocks that contain nonblocking assignments to v, the transition relation for v is equivalent to (5.7) conjoined with (5.5) and (5.6). The conjunction of all the transition relations for the state variables is the cycle based transition relation that is equivalent to (5.3). The value of a target variable in V_C that is designated as output is determined by new inputs and the current state variables as described in Lemma 5.7. The value of a target variable in V_S that is designated as output is determined as the the normal target variable in V_S is determined as described above. The cycle based output relation for **dut** is equivalent to (5.4), where each $z_i = \gamma(V_S, W)$ is an output relation for an output variable $z_i \in V_C \cup V_S$. $\square$

The relation between the input values and the initial state values of the state variables at $t = 0$ is captured by (5.2). This can be proved by reasoning similar to that used in Theorem 5.10.

5.4 Discussion

We have shown that the verification condition for synchronous hardware is encoded concisely into a BV formula that agrees with standard-compliant simulators. The only nondeterministic behavior described in the BV formula is the one that is caused by the interleaving of the sequential blocks. Other nondeterministic behaviors that are cumbersome to describe in the BV formula are avoided by Restrictions 4, 6, 7, and the atomicity rule.

For instance, without Restriction 4, the target of multiple assignments from different blocks may change its value nondeterministically every time one of those blocks is evaluated. While all sequential blocks are evaluated once per clock cycle, a combinational block may be evaluated at time t even when the final values of its inputs are the same at times $t-1$ and t. With just one block assigning to a target, it does not matter whether it is re-evaluated in such a case, because the assigned value does not change. With multiple assignments, it is hard to tell whether a new value may be assigned if only final values are considered.

Restrictions 6 and 7 guarantee that the values computed by sequential blocks do not depend on their order of evaluation. Hence, they restrict the nondeterministic behaviors that are arise from the interleaving of sequential blocks. Furthermore, the atomicity rule prevents the nondeterministic behavior caused by the interleaving of the assignments in different blocks in **MSV**. Without the atomicity rule, the simulator may generate a trace that is not captured by the BV formula.

Although **MSV** excludes some features of Verilog, it includes commonly used ones; hence most synchronous Verilog designs can be converted into **MSV** descriptions. For example, a **case** statement can be converted into an **if / else** conditional statement, and a function describing combinational logic can be converted into a combinational block in **MSV** if the function does not read global variables. In addition, Restriction 5 on **MSV** can be weakened by allowing a mixture of different assignment types that still describes combinational logic. Restriction 6 can be also weakened by allowing the target of a blocking assignment to be used outside of its block if that may not cause a hold time violation; however, the restrictions are kept simple, since most synchronous designs can be described within them.

In practice, more restrictive Verilog coding guidelines [MC99, Cum02] are used for describing hardware designs. The guidelines are useful to avoid many pre- and post-synthesis simulation mismatches caused by nondeterministic behavior in the design. In **MSV**, a nondeterministic **MSV** description can be easily excluded by imposing the restriction that prevents multiple assignments to the same target in sequential blocks. Given the simulation environment for a deterministic **MSV** description, the **strobe** task reports a unique execution trace during the simulation. For the synthesized circuit of the description, the unique execution trace is achieved by assigning the initial values to the state variables by the **initial** blocks of the description; hence there is no pre- and post-synthesis simulation mismatch for a deterministic **MSV** description. Furthermore, the unique execution trace is also captured in the BV formula of the description and vice versa.

The example in Fig. 5.5 shows that without initial block, simulation may produce incorrect results, namely $a = 1000\ldots$. With initialization, however, a mismatch between pre- and post-synthesis models signals a bug in either the synthesis or verification tools.

```
initial $\#0$ $\#0$ $a \Leftarrow w$;
always @(posedge clock) if $(a)$ $a \Leftarrow 1$; else $a \Leftarrow 0$;
```

Figure 5.5: Deterministic Verilog design

5.5 Related Work

In [Cho97], the author defined Synchronous Verilog (**SV**), a subset of the language that describes synchronous circuits. In contrast to our approach, the author aims at checking the behavioral equivalence between an intermediate form of an **SV** program and its synthesized circuit. In particular, a nondeterministic **SV** program is converted into a deterministic Verilog program by adding zero delays to the **SV** program; hence the user of **SV** has no control over nondeterministic behavior. Furthermore, equivalence between the Verilog program and the synthesized circuit is not proved. The author assumes that the initial values of the state variables in the synthesized circuit are all zeros; however, this may cause a mismatch between the RTL and the synthesized circuit.

In [Gor], the author defined **V0**, a subset of Verilog that has both event and trace semantics. The event semantics describes the execution of a **V0** program in terms of propagation of changes to variables, and the trace semantics describes the execution in terms of sequences of states. The state in the trace semantics changes in every simulation cycle as it does in our cycle based transition relation. The ultimate goal of this work is to prove that the restrictions in **V0** prevent nondeterministic behavior in **V0** programs and hence guarantee the consistency of the event and trace semantics; however, the preliminary report does not address semantic restrictions of **V0** that guarantee the consistency.

In [MKMR10], the authors focus on a detailed event semantics of Verilog rather than deriving an efficient verification condition for synchronous circuit. The execution of Verilog is described in rewriting logic that is implemented in the Maude tool. The tool can be used to verify the results of simulators or other formal tools.

5.6 Experimental Results

We have implemented a translator called **Vl2smt** that uses Icarus Verilog [Ica] as front end, accepts an **MSV** design as input, and generates a BV formula for the verification condition of the design. We used **Vl2smt** to perform equivalence checking of **cf-cordic** design from Opencores [Ope]. The original **cf-cordic** Verilog design (1143 lines of code) is composed of 13 modules, 30 **always** @ (**posedge clock**), 313 continuous assignments, and 30 **initial** blocks In the optimized design, we reduce the number of modules and continuous assignments to 5 and 196 manually. Although multiple modules and continuous assignments are not allowed in **MSV**, **Vl2smt** supports these features of Verilog. The original **cf-cordic** code is converted into the **MSV** code by changing the **initial** type to **initial** $\#0\ \#0$. To check the equivalence of the original and optimized designs, we generated the equivalence checking problem in BV formula by **Vl2smt**. The generated BV formula is composed of 30 state variables and the equivalence is proved by the BV solver (Boolector-1.4 [Boo]) in less than a second. Although the hierarchy of the design is flattened in the BV formula, the file size of the BV formula (185kb) is not much larger than the file size of its original code (50kb). To evaluate our tool, we used the Verilog designs from VIS Verilog benchmarks [VVB] and Opencores. For **cf-fir**, **altmult-accum**, and **FPMult** designs, we generated the equivalence checking problems as described above. For others, we generated Bounded Model Checking (BMC) problems with invariants. Table 5.1 shows the number of lines and the file size of each Verilog design, and shows the file size of the BV formula and the number of state variables in the formula. It also shows the unrolling depth, the CPU time, and the result of model checking. The pass result indicates that the invariant holds in the design, and the fail result indicates that the invariant fails at bound k. The unsat result indicates that the invariant holds up to the bound k.

5.7 Conclusions

In this chapter, we presented **MSV** with restrictions and proved behavioral equivalence between the verification condition and the simulation model. The restrictions allow us to generate a concise verification condition to be checked by an SMT solver. With controlled nondeterminism in **MSV**, nondeterministic

design	lines	V(byte)	BV(byte)	$\|V_S\|$	k	time(s)	result
cf-fir	428	12584	43921	16	-	0.1	pass
altmult-accum	166	3770	16835	8	-	0.1	pass
FPMult	236	7186	50728	14	-	0.1	pass
Timeout	196	6111	13220	10	80	181.53	unsat
FIFO	171	5016	31370	37	8	258.8	unsat
Am2910	116	3183	11378	9	100	13.79	unsat
MinMax	60	1537	4502	3	300	595.85	unsat
DAIO	259	8277	19991	14	14	0.7	fail
Blackjack	136	4261	40930	24	13	4.8	fail
Vending	252	6065	16237	10	2	0.1	fail

Table 5.1: Result of Vl2smt on Verilog designs

behavior of an **MSV** model can be easily eliminated and the mismatches between pre- and post synthesis simulations can be avoided.

Chapter 6

Selective SMT Encoding for Hardware Model Checking

6.1 Introduction

In Chapter 5, we have presented **MSV** with restrictions. In this chapter, we study the translation of **MSV** into a verification condition to be checked by SMT solvers. In today's hardware designs, bit-level and word-level operations are often tightly intermingled. On some designs, a bit-level model checker may perform better than a word-level model checker or vice versa. Depending on the characteristics of the design, we selectively choose an encoding method (either bit-level or word-level) to improve the efficiency of hardware model checking. We present a model analysis method for the encoding selection and evaluate the method on a set of hardware verification problems.

This work is motivated by the results shown in Fig. 6.1. We have encoded each pair of Verilog design and property into SMT for bounded model checking (BMC). In particular, we used BV and LIA encodings for each design. The details of these encoding methods will be discussed in Sect. 6.3. The Verilog designs we used are from VIS Verilog benchmarks [VVB], Opencores [Ope] and Altera design examples [Ter]. We compared BV solvers (Boolector-1.4 [Boo], Z3-2.8 [Z3], Beaver [Bea] with Precosat-456r2 [Pre]) and LIA solvers (MathSAT-4.3 [Mat], Yices-1.0.28, Z3-2.8) for the encodings. These solvers are the ones that performed best on our BMC problems. In the experiment, the timeout was set to 1000 seconds. Figure 6.1 shows the comparison of average CPU times of the solvers for the two encodings. Table A.1 in Appendix A shows the detailed results of the comparison.

The points above the diagonal are wins for the BV solvers, and the ones below are wins for the LIA solvers. As the scatterplot shows, some of the designs work well with BV encoding, and others work well

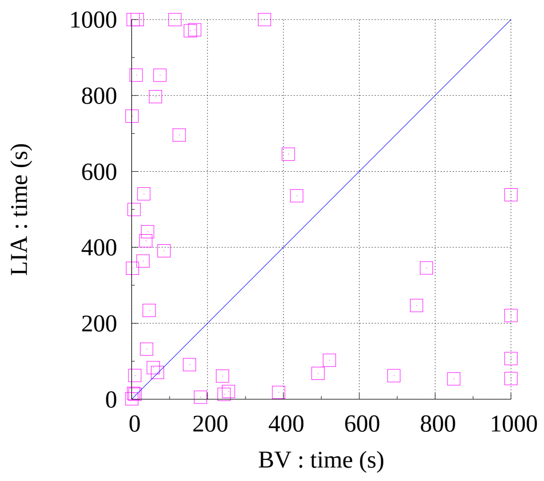


Figure 6.1: BV vs. LIA

with LIA encoding. This indicates that we need different encodings depending on the design.

We introduce a model analysis method that considers each bit-vector operation in the design and selects the encoding based on the analysis. In addition, we present several enhancements to SMT encoding for hardware designs. Our experiments show that our approach selects the right encoding for the hardware design and improves the efficiency of model checking.

The rest of this chapter is organized as follows. Section 6.2 describes the translation to BV logic. Section 6.3 describes the encoding methods. Section 6.4 presents a model analysis method. After a survey of related work in Sect. 6.5, conclusions are offered in Sect. 6.6.

6.2 From Hardware Description to BV

In this section, we outline the conversion from hardware description to BV formula. Hardware is assumed to be described in a subset of the Verilog hardware description language (HDL) [Ver] suitable for the modeling of synchronous hardware. The subset supports the mixture of blocking and non-blocking assignments in the procedural blocks, and allows non-deterministic interleaving of procedural blocks. We impose restrictions to the description to ensure that the evaluation of each procedural block is not affected by the interleaving of the assignments in different procedural blocks. The restrictions are compatible with common design guidelines used in the industry (e.g., blocking assignments for combinational logic and non-blocking assignments for memory elements) and allow us to produce concise verification conditions. Although the subset we consider includes essential features of Verilog, it does not support delays, strengths, and other features that are not needed for RTL verification of synchronous designs.

We represent a hardware description as a Concurrent Control Flow Graph (CCFG) [KGW10] in Static Single Assignment (SSA) form [CFR$^+$89]. With the CCFG, we generate a set of constraints in BV logic for blocking and non-blocking assignments in each procedural block. If there is a conflict among the constraints that is caused by different assignments in multiple procedural blocks, we generate an additional conflict arbitration constraint.

```
initial #0 #0 $z = 0$;                always @(posedge clk) begin
                                          $w = y$;
always @(posedge clk)                     if ($u$) $w = x$;
      if ($v$) $z \Leftarrow 1$;                 $z \Leftarrow w$;
                                      end

initial #0 #0 $z_1 = 0$;              always @(posedge clk) begin
                                          $w_1 = y_0$;
always @(posedge clk) begin               if ($u_0$) $w_2 = x_0$;
      if ($v_0$) $\bar{z}_1 = 1$;                $w_3 = \phi(w_2, w_1)$;
      $\bar{z}_2 = \phi(\bar{z}_1, z_1)$;        $\bar{z}_3 = w_3$;
end                                   end
```

Figure 6.2: Conversion from HDL to SSA form

In Fig. 6.2, the two procedural blocks at the top are converted into the SSA form at the bottom. In each procedural block, we generate the BV formula for each target variable. Suppose $u, v \in V_P$ and $w, x, y, z \in V_B(4)$. For the target z, we generate the BV formula

$$\mathit{ite}(v_0, \bar{z}_1[4] = 1[4] \wedge \bar{z}_2[4] = \bar{z}_1[4], \bar{z}_2[4] = z_1[4])) \tag{6.1}$$

in the first procedural block, and

$$w_1[4] = y_0[4] \wedge \textit{ite}(u_0, w_2[4] = x_0[4] \wedge w_3[4] = w_2[4], \\ w_3[4] = w_1[4]) \wedge \bar{z}_3[4] = w_3[4] \quad (6.2)$$

in the second procedural block. Then, we introduce z' for z and generate a conflict arbitration constraint $z'[4] = \bar{z}_2[4] \vee z'[4] = \bar{z}_3[4]$. This formula conjoined with (6.1) and (6.2) is the transition relation for the description, where z_1 and z' are the current and next state variables for z.

6.3 SMT Encoding for Hardware Designs

In Sect. 6.2, we showed how a hardware description is converted into a BV formula. In this section, we discuss the translation from BV encoding to LIA encoding and to BV $\cup$ LIA. SMT encoding for hardware design (RTL Verilog) was first presented in [Bru08] where both BV and LIA encodings for combinational circuits were introduced. We review those basic encoding methods for LIA, and introduce several enhancements.

6.3.1 LIA Encoding

In LIA encoding, each bit-vector $x[n]$ is encoded into an integer variable X with a bound constraint $0 \le X < 2^n$.

For an equality $z[n] = concat(x[i], y[j])$ with the $concat$ term and bit-vectors $x[i], y[j], z[n]$ where $n = i + j$, we generate

$$Z = 2^j \cdot X + Y \quad . \quad (6.3)$$

For an equality $z = x[i:j]$ with the bit-select term and bit-vectors $x[n], z[i-j+1]$ where $n > i \ge j \ge 0$, three fresh variables X_h, X_m, X_l that correspond to the bit-vectors $x[n-1:i+1], x[i:j], x[j-1:0]$ are introduced to generate

$$(X = 2^{i+1} \cdot X_h + 2^j \cdot X_m + X_l) \wedge (0 \le X_h < 2^{n-i-1}) \wedge \\ (0 \le X_m < 2^{i-j+1}) \wedge (0 \le X_l < 2^j) \wedge (Z = X_m) \quad . \quad (6.4)$$

When converting an arithmetic BV term into an LIA term, we need to deal with overflow. We introduce either a fresh variable or a term-ITE operator. In particular, for an addition operation $z[n] = x[n] + y[n]$ with $x[n], y[n], z[n]$, we may generate either

$$(Z = X + Y - 2^n \cdot \alpha) \wedge (0 \leq \alpha \leq 1) \tag{6.5}$$

with a fresh variable α, or

$$(Z = \textit{tite}(X + Y \geq 2^n, X + Y - 2^n, X + Y)) \tag{6.6}$$

with a term-ITE operator. For an equality $z[n] = k[n] \cdot x[n]$ with multiplication, where $x[n]$, $z[n]$ are bit-vectors, $k[n]$ is a constant, and α is a fresh variable, we generate

$$(Z = k \cdot X - 2^n \cdot \alpha) \wedge (0 \leq \alpha \leq k - 1)\ \ . \tag{6.7}$$

For an equality $z[n] = x[n] \diamond y[n]$ with a bit-wise term and the bit-vectors $x[n], y[n], z[n]$ where $\diamond \in \{\&, |\}$, we introduce fresh variables $X_0, ..., X_{n-1}, Y_0, ..., Y_{n-1}, Z_0, ..., Z_{n-1}$ for the bit-vectors. Suppose $\diamond$ is $\&$. Then, we generate

$$(Z = \sum_{i=0}^{n-1} 2^i \cdot Z_i) \wedge (X = \sum_{i=0}^{n-1} 2^i \cdot X_i) \wedge (Y = \sum_{i=0}^{n-1} 2^i \cdot Y_i) \wedge \\ \bigwedge_{i=0}^{n-1} (Z_i = 1) \leftrightarrow ((X_i = 1) \wedge (Y_i = 1))\ \ .$$

Having reviewed the basic encoding method we present two enhancements: selective value enumeration and term-ITE introduction for BV arithmetic terms.

The basic encoding methods often introduces the product $k \cdot X$ where k is a constant and X is a variable. The coefficient k may be large, and large coefficients often degrade the performance of LIA solvers because they often require many pivots in the simplex-based ILP (Integer Linear Programming) algorithm [DdM06b, NW88]. We tackle the problem with selective enumeration. If the range of X is small enough to express it with few term-ITEs, term-ITEs replace the multiplication. For instance, if $0 \leq X \leq 1$ in (6.3), then the new encoding with a term-ITE is

$$Z = \textit{tite}(X = 1, 2^j + Y, Y)\ \ .$$

For arithmetic terms, we saw two types of encoding in (6.5) and (6.6). The LIA encodings for an equality $z[n] = \sum_{i=1}^{m} x_i[n]$ with a general arithmetic term can be

$$(Z = (\sum_{i=1}^{m} X_i) - 2^n \cdot \alpha) \wedge (0 \leq \alpha \leq m-1) \ , \tag{6.8}$$

and

$$\begin{aligned} t_{m-1} = \textit{tite}(t_m \geq (m-1) \cdot 2^n, t_m - (m-1) \cdot 2^n, t_m) \\ \wedge\, t_{m-2} = \textit{tite}(t_{m-1} \geq (m-2) \cdot 2^n, \\ t_{m-1} - (m-2) \cdot 2^n, t_{m-1}) \wedge \cdots \wedge \\ t_2 = \textit{tite}(t_3 \geq 2 \cdot 2^n, t_3 - 2 \cdot 2^n, t_3) \wedge \\ Z = \textit{tite}(t_2 \geq 2^n, t_2 - 2^n, t_2) \wedge t_m = \sum_{i=1}^{m} X_i \ . \end{aligned} \tag{6.9}$$

We prefer (6.9), which introduces term-ITEs, to (6.8), because (6.8) often introduces a large coefficient for the fresh variable α.

For multiplication, the encoding in (6.7) also introduces a large coefficient for α. As an alternative, we use the encoding

$$\begin{aligned} t_{N_t-1} = \textit{tite}(k \cdot X \geq 2^{N_t-1} \cdot 2^n, k \cdot X - 2^{N_t-1} \cdot 2^n, k \cdot X) \wedge \\ t_{N_t-2} = \textit{tite}(t_{N_t-1} \geq 2^{N_t-2} \cdot 2^n, t_{N_t-1} - 2^{N_t-2} \cdot 2^n, t_{N_t-1}) \\ \wedge \cdots \wedge t_1 = \textit{tite}(t_2 \geq 2 \cdot 2^n, t_2 - 2 \cdot 2^n, t_2) \wedge \\ Z = \textit{tite}(t_1 \geq 2^n, t_1 - 2^n, t_1) \ . \end{aligned} \tag{6.10}$$

The conditions of the term-ITEs in (6.10) enumerate the different overflow cases. If a condition is true, the value of $k \cdot X$ overflows; hence, the true branch of the term-ITE subtracts a power of 2 from the value of $k \cdot X$ to satisfy the condition $0 \leq k \cdot X < 2^n$.

The number of term-ITEs N_t required for encoding a multiplication $k[n] \cdot x[n]$ in LIA is given by

$$N_t = \lceil log_2(k) \rceil \ .$$

For a BV equality $z[n] = k_1[n] \cdot x[n] + k_2[n] \cdot y[n]$,the number of term-ITEs is $N_t = \lceil log_2(k_1) \rceil + \lceil log_2(k_2) \rceil + 1$ with the first method and $N_t = \lceil log_2(k_1 + k_2) \rceil$ with the second method. Since

$$\lceil log_2(k_1 + k_2) \rceil \leq \lceil log_2(k_1) \rceil + \lceil \log_2(k_2) \rceil + 1 \ ,$$

we use the second method. The number of term-ITEs in (6.9) can be reduced from $m - 1$ to $\lceil log_2(m) \rceil$.

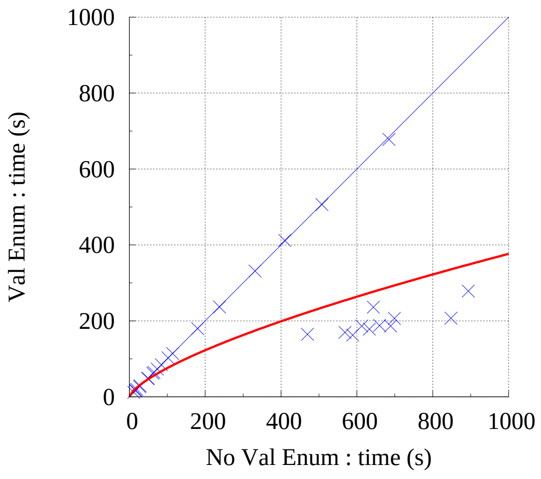


Figure 6.3: VAL ENUM vs. No Val Enum

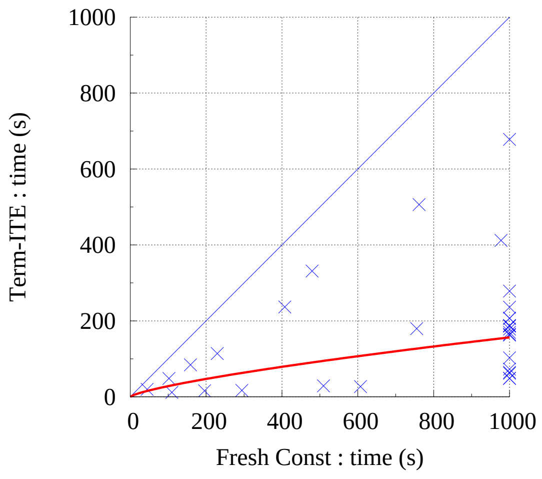


Figure 6.4: TERM-ITE vs. Fresh Variable

The results of Yices (LIA) on the hardware verification problems in Fig. 6.1 with and without the enhanced encodings are shown in Fig. 6.3 and Fig. 6.4. In the experiments, the timeout was set to 1000 seconds. Figure 6.3 compares the encodings with and without value enumeration. Figure 6.4 compares the encodings with and without term-ITE introduction. Points below the diagonal represent wins for the enhanced encoding. Each scatterplot shows two lines: The main diagonal, and $y = \kappa \cdot x^{\eta}$, where κ and η are obtained by least-square fitting. Figure 6.3 shows that the encoding with the value enumeration outperforms the one without. Figure 6.4 shows that the encoding with the term-ITE introduction often outperforms the one without significantly. Tables A.3 and A.4 in Appendix A show the detailed results of the comparisons.

6.3.2 SMT Encoding with Combined Theories (BV $\cup$ LIA)

In this section, we describe the conversion from BV encoding to BV $\cup$ LIA encoding. Since representing bit-wise operations in LIA is rather inefficient, when a BV variable is involved in both arithmetic and bit-wise operations, it may be convenient to split it into a BV variable and an LIA variable constrained so as to always have the same value. For example, given the BV formula $c[n] = a[n]$ & $b[n]$ $\wedge d[n] = c[n] + e[n]$, where the bit-vector $c[n]$ is used both in bit-wise and arithmetic terms, if we convert the formula to BV $\cup$ LIA, the equality with the bit-wise term remains as is, and the equality with the arithmetic term is converted into the LIA formula

$$(0 \leq C < 2^n) \wedge (0 \leq D < 2^n) \wedge (0 \leq E < 2^n) \wedge$$
$$(D = \mathit{tite}((C + E \geq 2^n), (C + E - 2^n), C + E)) \ \ ,$$

where C, D, and E are fresh variables for the bit-vectors $c[n], d[n]$, and $e[n]$. Since the fresh variable C corresponds to the bit-vector $c[n]$, we generate the interface constraint

$$C = \mathit{tite}(c[n-1:n-1] = 1[1], 2^{n-1}, 0) +$$
$$\mathit{tite}(c[n-2:n-2] = 1[1], 2^{n-2}, 0) + \cdots +$$
$$\mathit{tite}(c[0:0] = 1[1], 1, 0)$$

for the relation between $c[n]$ and C, and conjoin it with the LIA formula. Overall, the BV $\cup$ LIA encoding for the BV formula $c[n] = a[n]$ & $b[n] \wedge d[n] = c[n] + e[n]$ is:

$$(c[n] = a[n] \ \& \ b[n]) \wedge$$
$$(0 \leq C < 2^n) \wedge (0 \leq D < 2^n) \wedge (0 \leq E < 2^n) \wedge$$
$$(D = \mathit{tite}((C + E \geq 2^n), (C + E - 2^n), C + E)) \wedge$$
$$C = \mathit{tite}(c[n-1:n-1] = 1[1], 2^{n-1}, 0) +$$
$$\mathit{tite}(c[n-2:n-2] = 1[1], 2^{n-2}, 0) + \cdots +$$
$$\mathit{tite}(c[0:0] = 1[1], 1, 0) \ \ . \tag{6.11}$$

Furthermore, given the BV $\cup$ LIA encoding, the BV formula in the encoding can be converted into an equisatisfiable Boolean formula by bit blasting. In particular, for the bit-vectors $a[n], b[n]$, and $c[n]$ in (6.11), a set of propositional variables $V_P = \{a_{n-1}, a_{n-2}, \ldots, a_0, b_{n-1}, b_{n-2}, \ldots, b_0, c_{n-1}, c_{n-2}, \ldots, c_0\}$

are generated. The BV formula $(c[n] = a[n]\ \&\ b[n])$ is converted into

$$(c_{n-1} \leftrightarrow (a_{n-1} \wedge b_{n-1})) \wedge (c_{n-2} \leftrightarrow (a_{n-2} \wedge b_{n-2})) \wedge \cdots \wedge (c_0 \leftrightarrow (a_0 \wedge b_0)) \ ,$$

and each BV formula $c[k:k] = 1[1]$ where $1 \leq k < n$ is converted into a propositional variable c_k in the interface constraint.

6.4 Model Analysis

Figure 6.1 shows that choosing the proper encoding is important. Given a hardware design, we analyze the model to choose the encoding method between BV and LIA (plus, possibly, bit blasting). If the model contains many bit-wise and bit-select operators, or it uses only a narrow data path, then the BV encoding is more likely to be suitable for the model. On the other hand, if the model contains a large number of arithmetic and relational operators with a wide data path, the LIA encoding may be preferable. In practice, we often encounter designs with a mixture of bit-wise, bit-select, and arithmetic operators. On those problems, it is hard to apply LIA solvers even though they contain a large number of arithmetic operators with wide data paths. On the other hand, there is still a chance to apply LIA solver if certain conditions are met. We discuss these conditions in the following.

6.4.1 Analysis of Bit-Select Operations

The bit-select operators in hardware designs often produce LIA encodings that are hard for SMT solvers. As shown in (6.4), each bit-select operator generates three fresh variables possibly with large coefficients. If there are multiple bit-select operations applied to one bit-vector, there is no benefit in encoding them in LIA. In [Bru08], the author showed degradation of performance in an LIA solver as the number of slices of a bit-vector grows. When a slice includes either the MSB (most significant bit) or the LSB (least significant bit) of a bit-vector, only two fresh variables are needed. However, the LIA encoding may not be efficient depending on the location of the slice. According to our experiments, if the bit-vector is decomposed only into two and the slicing bit is close to the MSB, then LIA encoding can be still effective. In practice, the slice is often applied close to the MSB of the data path.

6.4.2 Analysis of Bit-Wise Operation

Bit-wise operators make LIA encoding much harder compared to the encodings for other BV operators. There is not much choice but to bit-blast the bit-vectors in the bit-wise operations. On the other hand, some designs contain a large number of arithmetic operations with wide data paths and small numbers of bit-wise operations. In those designs, BV $\cup$ LIA encoding can be used to encode the bit-wise operation with BV logic, and still maintain the arithmetic operations with LIA encoding. Unfortunately, SMT solvers for BV $\cup$ LIA encoding do not perform well compared to other solvers (BV or LIA) according to our experiments. Instead of using the BV $\cup$ LIA encoding, we apply bit blasting for the bit-wise operations and use LIA encoding for the arithmetic operations. For each bit-vector subjected to both bit blasting and arithmetic operations, we introduce an interface constraint as discussed in Sect. 6.3.2. The experimental results in Fig. 6.5 compare LIA encoding with and without bit blasting for the Palu design [VVB]. As Fig. 6.5 shows, LIA encoding with bit blasting gives much better performance compared to pure LIA encoding. We also compared LIA encoding with bit blasting and BV $\cup$ LIA encoding. The solver with BV $\cup$ LIA encoding timed out for most of the Palu problems whereas the solver with LIA with bit blasting solved all the problems within the timeout (1000 seconds). Table A.5 in Appendix A shows the detailed results of the comparison.

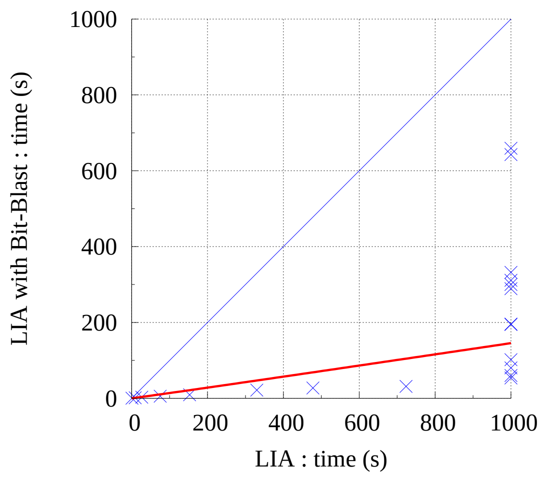


Figure 6.5: LIA WITH BIT-BLAST VS. LIA

6.4.3 Scoring System

The model analysis method decides the encoding method based on a scoring system. Let $\textbf{Score}_B$ be the score for BV encoding and $\textbf{Score}_L$ be the score for LIA encoding. Let w_a, $w_{\textbf{bw}}$, $w_{\textbf{bs}}$, and $w_{\textbf{r}}$ be the weights for the arithmetic, relational, bit-wise, and bit-select operators, with $w_{\textbf{bw}} > w_{\textbf{bs}} > w_a > w_{\textbf{r}}$. We give a larger value to $w_{\textbf{bw}}$ and $w_{\textbf{bs}}$ because the numbers of bit-wise and bit-select operators have a stronger impact on the effectiveness of the LIA encoding than the numbers of arithmetic and relational operators have on the effectiveness of the BV encoding. The score is computed for each relational expression e_r in the transition system based on (6.12) and (6.13), in which $bw(e_r)$ is the number of bit-wise operators, $bs(e_r)$ is the number of bit-select operators, $ar(e_r)$ is the number of arithmetic operators, $re(e_r)$ is the number of relational operators, and $bits(e_r)$ is the number of bits in e_r.

$$\textbf{Score}_B = bw(e_r) \times bits(e_r) \times w_{\textbf{bw}} + bs(e_r) \times bits(e_r) \times w_{\textbf{bs}} \ . \tag{6.12}$$

$$\textbf{Score}_L = ar(e_r) \times bits(e_r) \times w_a + re(e_r) \times bits(e_r) \times w_r \ . \tag{6.13}$$

A bit-select operator that decomposes the data path into only two and whose slicing bit is close to the MSB is considered a weak bit-select and is not counted in $bs(e_r)$.

Given the scores $\textbf{Score}_B$ and $\textbf{Score}_L$ and their thresholds $\textbf{th}_B$ and $\textbf{th}_L$, we compare the score with its threshold and decide the encoding method. If $\textbf{Score}_L > \textbf{th}_L$ and $\textbf{Score}_B < \textbf{th}_B$, then we select LIA encoding, otherwise we select BV encoding. When encoding in LIA, the bit-vectors in the bit-wise operations are bit-blasted, and the bit-vectors only in the relational operators are also bit-blasted. The selective bit blasting in LIA encoding often improves the efficiency of SMT solvers.

6.4.4 Experimental Evaluation

We have implemented a translator called **Vl2smt** that uses Icarus Verilog [Ica] as front end, accepts a Verilog design as input, and generates an SMT formula for the verification condition of the design. The translator chooses the encoding method for a given design between BV and LIA with bit blasting as discussed in Sect. 6.4.3. We used the set of designs of Fig. 6.1 as training set for the predictor. All results are for the solvers listed as in Sect. 6.1 with a timeout of 1000 seconds. Figure 6.6 shows the comparison of average

CPU times of BV and LIA solvers with the designs classified according to the predicted encoding method. The symbol ○ is used for designs with BV encoding prediction, and the symbol × is used for the design with LIA encoding prediction. The scatterplot shows that most designs for which BV encoding was predicted to work better actually end up above the diagonal, while most designs for which LIA encoding was predicted to work better actually end up under the diagonal. This result shows that **Vl2smt** predicts the right encoding for most of the problems in the training set. Table A.2 in Appendix A shows the detailed results of the selection.

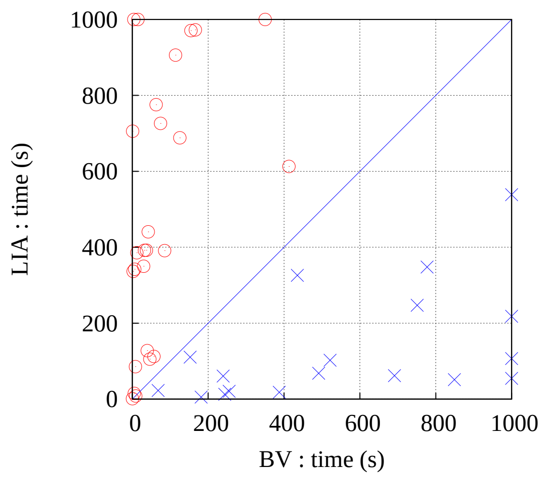


Figure 6.6: BV vs. LIA for Training Set

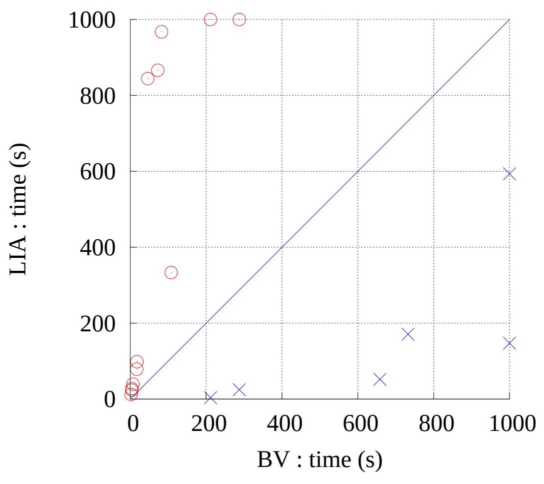


Figure 6.7: BV vs. LIA for Evaluation Set

A set of hardware model checking problems from VIS Verilog benchmarks [VVB], Opencores [Ope] and Altera design examples [Ter] disjoint from the training set was used for evaluation of **Vl2smt**. The result of the evaluation in Fig. 6.7 shows that **Vl2smt** predicts the right encoding method for each of these model checking problem. Table A.6 in Appendix A shows the detailed results of the evaluation.

Table 6.1 shows the average number of bits, the numbers of arithmetic, relational, bit-wise, and bit-select operations, the scores, and the encoding predictions for the models in the training (T-Model) and evaluation (E-Model) sets.

6.5 Related Work

As we discussed in Sect. 6.3, the basic LIA encoding for combinational circuits was presented in [Bru08]. In contrast to our selective approach for hardware verification, they adopted the layered approach inside the solver that deals with EUF, the incomplete BV, and the complete LIA encodings. In [Bje09], the author presented a word-level reduction method for industrial netlist verification. He focused on simplifying the netlist as much as possible by applying word-level reductions to equality and disequality comparators. Then, the simplified netlist was bit-blasted, and solved with either SAT or BDDs. In [KJJP09], the authors applied BV solvers to equivalence checking of a system-level model and an RTL design. In [WSBK07], the authors presented a normalization technique to simplify the word-level description of an arithmetic circuit for SAT-based BMC. In [PICB05], the authors presented a simplification method for RTL-SAT instances with the combination of interval-arithmetic and Boolean reasoning. Earlier references of word-level hardware verification include [BD02], [Dre04], and [ZKC01]. Finally, the authors of [XHHLB08] presented an algorithm selection approach that selects one among the SAT solvers that performed best on a representative set of problem instances.

6.6 Conclusions

The choice of the right encoding style has great effect on the efficiency of model checkers at the word level. In this chapter, we have presented a selective SMT encoding for hardware model checking. The approach is based on a model analysis method that selects the encoding by considering several characteristics of the model. In particular, the effects of bit-vector and bit-select operations have been studied. Experiments show that our approach selects the right encoding for most of the designs. This greatly improves the efficiency of hardware model checking. Enhanced encoding techniques have also been introduced and their effectiveness demonstrated experimentally.

T-Model	bit	ar	re	bw	bs	$Score_L$	$Score_B$	**Enc**
Am2910	11	1	13	0	0	66.4	0	BV
Bakery	5	0	23	0	0	54.8	0	BV
Blackjack	5	4	15	0	0	205.4	0	BV
Cube	4	0	106	0	0	179.2	0	BV
FPMult	11	44	50	20	22	1167.6	8181	BV
Palu	18	12	19	9	4	663.6	9216	BV
RetherRTF	5	0	8	0	0	25	0	BV
Swap	3	0	11	0	0	14	0	BV
Miim	4	19	22	4	2	257.2	516	BV
Timeout	51	1	20	0	0	513	0	LIA
cf_fir	9	106	69	0	12	2461.2	0	LIA
FIFOs	60	0	46	0	0	984	0	LIA
FIR	17	40	4	0	9	1715.2	0	LIA
DSP_Adder	23	94	34	0	0	2378.8	0	LIA
MinMax	48	2	21	0	0	540.4	0	LIA
E-Model	bit	ar	re	bw	bs	$Score_L$	$Score_B$	**Enc**
cf_cordic	16	8712	2606	0	314	205754	90112	BV
Daio	2	16	11	0	1	116	0	BV
Dekker	2	0	4	0	0	3.2	0	BV
Unidec	4	0	55	0	28	352	3584	BV
soc_ram	46	0	10	0	0	657.6	0	LIA
AltMult	8	58	48	0	0	1247.2	0	LIA

Table 6.1: Comparison of using selective, SAT, BV, and LIA encodings on evaluation set

Chapter 7

Application of Formal Word-Level Analysis to Constrained Random Simulation

7.1 Introduction

During our study of decision procedures for SMT, we have found that they are also applicable to constrained random simulation. Constrained random simulation is in increasing demand with hardware designers and verification engineers. As the name indicates, it is the simulation of a design under specified constraints. The user is required to capture the behavior of the environment of the design as constraints and the simulation tools simulate the design under these constraints with the aid of constraint solvers embedded in them. Commercial tools, such as Specman, have been popular for providing this capability. To address the need for constrained random simulation, modern hardware description languages (HDL), such as System Verilog, have incorporated constraint specification as part of their syntax.

The overwhelming benefit of constrained random simulation over the traditional writing of test-benches is the automation. Once the constraints are specified, the constraint solver in the simulator enumerates the valid scenarios instead of a human. Further, by specifying weights on the search space, the user can indicate whether the constrained space should be sampled uniformly or specific areas should be focused on.

Given that constraint solving comprises the bulk of constrained random simulation time, the efficiency and performance of constraint solvers is critical. Traditional constraint solving techniques, such as integer linear programming and constraint programming, far lag the performance of simulators. Boolean engines, e.g., BDDs, have been applied quite successfully to this problem[YSP$^+$99] by taking advantage of the finite state nature of HDL constraints. More recently, Kitchen and Keuhlmann[KK07] have provided a word-level technique based on Markov-chain Monte Carlo methods. The scalability of this technique to industrial strength designs is yet to be proven.

In our constraint solver, **ValueGen**, we have incorporated both BDD and SAT-based Boolean engines. BDDs provide the advantage of fast generation of uniformly distributed solutions. However, some constraints have very large BDDs that cause memory explosion during simulation. SAT solvers are less vulnerable to size explosion. On the other hand, each solution generation could be exponentially slower than BDDs.

In this chapter, we present a word-level pre-processor, **DomRed**, that **ValueGen** applies to the constraints to reduce the size of their representation in the Boolean engines. The pre-processing is a static analysis technique that uses an SMT-like framework. **DomRed** combines a SAT solver and a linear arithmetic solver that handles primarily integer difference logic, with a minor extension to positive and negative coefficient inequalities. The input to the tool is a Boolean combination of linear arithmetic constraints and bit-vector constraints. The output is a set of variables and their reduced domains. The constraints with reduced-domain variables are then passed on to the Boolean engines, resulting in smaller Boolean representations for constraint solving. We present experimental results of applying **DomRed** within **ValueGen** on our simulation testcases.

7.2 Constraint Solving in Simulation

Constraints are Boolean combinations of linear arithmetic and bit-vector expressions on design variables. The expressiveness of the specified constraints is limited by the HDL being used. For example, a System Verilog constraint is

```
constraint c1 {src_addr >= 0 && src_addr < 65536 &&
               payload_len >= 0 && payload_len < 4096 &&
               dest_addr - src_addr >= 4096 && dest_addr < 65536}
```

Constraint solving is the task of generating values for the design variables that satisfy the constraints. In the above example, `src_addr` $= 512$, `payload_len` $= 1024$, `dest_addr` $= 4608$ is a set of legal values. Our constraint solver, **ValueGen**, is invoked dynamically during simulation i.e., every time the simulator encounters a user call to generate new values for variables appearing in constraints, the simulator calls the constraint solver. Tight integration is required between the two to maintain efficiency.

Constraints are typically written on the inputs of the design and may depend on some internal design signals (**state** variables). During constraint solving, the solver is required to generate values that satisfy both the constraints as well as the states values.

Each set of related HDL constraints, when encountered, is parsed by the simulator, and sent to **ValueGen** through a word-level API along with the state values. Internally, **ValueGen** maintains a applies several optimizations at the word-level, including partitioning based on non-overlapping variable support and constant propagation. Finally, it bit-blasts the word-level constraints and calls the Boolean engines (BDD or SAT) on the Boolean representation.

The optimizations in **ValueGen** result from syntactic and very minor semantic analysis of the constraints. They do not include the ability to deduce that the tightest ranges of `dest_addr` and `src_addr` in the above example. **DomRed** addresses exactly this deficiency. It extracts a subset of invariants from semantic analysis of the constraints. If an invariant yields variable bound reductions, then the reduced number of bits are applied to encode the respective variables, the default number of bits are used otherwise.

7.3 DomRed: Technical Details

ValueGen provides **DomRed** with a quantifier-free first order logic formula with linear arithmetic constraints. An **LA** constraint is of the form $a_1x_1 + \ldots + a_nx_n \bowtie c$, where $\bowtie \in \{=, \leq, <, >, \geq, \neq\}$. A **difference** constraint is a special case of an LA constraint whose form is $x_i - x_j \bowtie c$. A positive-(negative-)inequality is another special case of an LA constraint where $\forall i.a_i \geq 0, x_i \geq 0, c \geq 0$ ($\forall i.a_i \leq 0, x_i \leq 0, c \leq 0$). We are working on the extension to bit-vector constriants.

As in the SMT-framework, the first order logic formula is abstracted conservatively into a propositional formula and given to the SAT solver. The SAT solver extracts a set of level-zero assignments, which corresponds to a set of LA constraints. From this set, we gather **difference** constraints, analyze them with the Bellman-Ford algorithm described in [KS06] and derive reduced bounds for the variable domains if possible. Among the LA constraints left over, positive- and negative-coeffient inequalites may also yield

Table 7.1: Comparison Table of without and with Bound Reduction

Design	Sim. cycles	# of bits			CPU (sec)			MEM (Mbytes)		
		w.o.	with	%	w.o.	with	%	w.o.	with	%
design1	5000000	112	101	10	683.0	549.4	**20**	40.8	34.2	**16**
design2	1000000	335	321	4	325.5	319.2	**2**	70.6	53.9	**24**
design3	50000	491	301	39	412.3	333.4	**19**	103.1	93.5	**9**
design4	1000000	54	40	26	180.9	174.1	**4**	37.2	37.8	-2
design5	1000000	64	60	6	86.1	44.0	**49**	33.2	33.6	-1
design6	1000000	64	60	6	75.9	48.1	**37**	33.2	33.7	-1
design7	1000000	16	14	12	340.2	344.6	-1	37.0	33.8	**9**
design8	44000	7	5	29	967.2	966.7	0	115.0	116.4	-1
design9	400000	8484	8428	1	607.1	559.6	**8**	62.3	62.0	0
design10	40	160	97	39	648.5	603.3	**7**	809.1	756.2	**7**
design11	2500	374	335	10	234.6	186.3	**21**	370.7	282.1	**24**

reduced upper (lower) bounds of x_i equal to c/a_i. The remaining LA constraints are conservatively marked as not yielding any domain reduction.

Example: Users commonly declare design inputs as `int`, meaning a 32-bit finite integer, causing the Boolean representation of the example in Section 7.2 to contain 96 bits. In applying **DomRed**, the equality constraint is translated into two inequalities in the usual manner. Inequalities are encoded with one bit each in the SAT solver. All these bits appear in the set of level-zero assignments. Since they all correspond to difference constraints, the Bellman-Ford algorithm yields the intervals $[0, 61439]$ for `src_addr`, $[0, 4095]$ for `payload_len` and $[4096, 65535]$ for `dest_addr`. The Boolean encoding will then require 16, 12 and 16 bits respectively, totalling 44 bits in the resulting Boolean expression (more than 2X reduction).

DomRed may also indicate to **ValueGen** that the constraints are infeasible (over-constrained situation) if the SAT solver or the LA solver detects it. This is of great value to **ValueGen** since it can avoid building the Boolean representations altogether.

7.4 Experimental Results

We integrated our tool **DomRed** into **ValueGen**, which, in turn, is integrated with our simulator. Our benchmark set includes both System-C and System Verilog examples. The System-C examples are smaller in size; 40 out of 68 showed improvements, the rest showed no degradation. The detailed table of results is not presented here for lack of space. The System Verilog examples consist of industrial-strength customer benchmarks. Of the 34 System Verilog examples that we experimented with, 11 showed improvement and are presented in Table7.1, the remaining 23 showed no degradation.

We use three parameters to measure the performance impact of applying **DomRed**—number of bits, CPU times and memory used. **ValueGen** switches between the BDD and SAT solver based on the Boolean representation size to maximize the size constraints that can be solved and optimize the speed of constraint solving (better with BDDs). Our experimental results show the improvement over the default optimized algorithm. However, this makes comparing the Boolean representation sizes harder since different solvers may be used when **DomRed** is applied. We are working on addressing this problem to obtain a tighter comparison.

Column 1 of Table7.1 specifies the design, Column 2 shows the number of simulation cycles, Columns 3–5 show the reduction of the number of bits in the constraints. Note that the number of bits is measured for the constraints only and the design may have several thousand more bits. Columns 6–8 show the CPU times and Columns 9–11 the memory reduction. The time taken by **DomRed** is negligibly small and hence, not presented here. The CPU time includes simulation time only in 2/11 cases, hence the CPU time improvement for most examples is for constraint solving alone.

The table shows that the reduction in the number of bits is sometimes substantial, upto 39%. Smaller constraints yield better CPU times and memory reductions. Given that **DomRed** takes negligible time, 11/34 examples show improvement on applying **DomRed** and the remaining 23 examples are no worse off, we conclude that **DomRed** is a cheap preprocessing technique and that it is always beneficial to apply it. These results are encouraging and as part of future work, we hope to apply more powerful static analysis to reduce the size of the Boolean representation even further.

Chapter 8

Conclusions

8.1 Thesis Conclusions

In this thesis, we have presented several efficient SMT solving techniques that can be applied to hardware model checking and constrained random simulation. To improve the efficiency of SMT solvers, we have presented a hybrid method that combines lazy and eager approaches. In addition, we have presented an SMT preprocessing technique that simplifies the original formula in word level. The presented SMT solving techniques are applied to hardware model checking and constrained random simulation, and the experimental results show the effectiveness of these approaches.

In Chapter 3, we have presented a finite instantiation approach combined with the Bellman-Ford algorithm to solve integer difference logic. The approach is particularly effective when the constraints are rich in disequalities. We have presented a bound computation algorithm for the integer variables in the constraints including the disequalities by restricting consideration to a small sufficient set of solutions. Experiments show that the approach is more effective compared to the one that splits the disequality in the disjunction of inequalities.

In Chapter 4, we have presented an algorithm for the term-ITE conversion in SMT preprocessing. The approach is based on the computation of cofactors and theory simplification. The simplification is done by detecting special cases in the formula or using theory propagation on the atomic predicates. Experiments show that the approach is very effective in most QF_LIA benchmarks in SMT-LIB and often speeds up SMT solvers.

In Chapter 5, we have presented **MSV** with restrictions and proved behavioral equivalence between the verification condition and the simulation model. The restrictions allow us to generate a concise verification condition to be checked by an SMT solver. With controlled nondeterminism in **MSV**, nondeterministic behavior of an **MSV** model can be easily eliminated and the mismatches between pre- and post synthesis simulations can be avoided.

In Chapter 6, we have presented a selective SMT encoding for hardware model checking. The approach is based on a model analysis method that selects the encoding by considering several characteristics of the model. Experiments show that our approach selects the right encoding for most of the designs and hence improves the efficiency of hardware model checking. Enhanced encoding techniques have also been introduced and their effectiveness demonstrated experimentally.

In Chapter 7, we have presented a new application of using SMT to constrained random simulation. To avoid size explosion problem in the bit-level solver of the constrained random simulation, we applied the word-level analysis with SMT solver on the model. We use the Bellman-Ford algorithm and simple coefficient checking to reduce the bounds of the variables used in bit encoding. Experiments show that our simple and fast algorithm can give huge amount of reduction to the variables in the real problem.

8.2 Future Work

Although **MSV** is a small subset of Verilog, our tool **Vl2smt** supports more features such as multiple modules, continuous assignments, and case statements. The tool can be further extended to handle more Verilog features such as **fork-join**, **repeat**, **function**, **task**, **assign**, **deassign**, **wait**, etc. On the other hand, we need further study on the behavior of each Verilog feature to describe the behavior correctly into a verification condition. More restrictions for the additional Verilog feature will be required to generate a concise verification condition with the correct behavioral description. For instance, a function describing combinational logic should not contain a global variable since only the inputs of the function are considered as a member of sensitivity list. (A procedural block in System Verilog such as **always_comb** also addresses some of the problems describing the correct behavior.) For handling delays and event controls, **MSV** will be required to have more fine-grained semantics.

We have shown that the verification condition for a hardware design encoded in BV, LIA or $\mathsf{BV} \cup \mathsf{LIA}$

logic is more concise than the SAT encoding; however, the verification condition may still get large if the design contains a memory whose depth is large and the memory elements are accessed or updated frequently in the design. Whenever the memory element is accessed or updated with an index variable, the values of the index variable needs to be enumerated for the encoding; the enumeration generates a large number of constraints. To preserve the conciseness of the SMT encoding, the logics of arrays and EUF can be used for encoding memories in hardware designs.

Bibliography

[AMP06] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. In Thirteenth International SPIN Workshop on Model Checking of Software (SPIN'06), Vienna, Austria, March 2006.

[BB09] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In TACAS '09: Proceedings of the Fifteenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 174–177, Berlin, Heidelberg, 2009. Springer-Verlag.

[BBC$^+$05a] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In Seventeenth Conference on Computer Aided Verification (CAV'05), pages 335–349. Springer-Verlag, Berlin, July 2005. LNCS 3576.

[BBC$^+$05b] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. van Rossum, S. Schulz, and R. Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'05), pages 317–333, Edinburgh, UK, April 2005. LNCS 3440.

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

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99), pages 193–207, Amsterdam, The Netherlands, March 1999. LNCS 1579.

[BCF$^+$06] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. Delayed theory combination vs. Nelson-Oppen for satisfiability modulo theories: A comparative analysis. In LPAR, pages 527–541, 2006.

[BD02] R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear programming. In ASP-DAC '02: Proceedings of the 2002 Asia and South Pacific Design Automation Conference, pages 741–746, 2002.

[BDS02] C. W. 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, Fourteenth Conference on Computer Aided Verification (CAV'02), pages 236–249. Springer-Verlag, Berlin, July 2002. LNCS 2404.

[Bea] URL: http://www.eecs.berkeley.edu/ jha/beaver.html.

[Bje09] P. Bjesse. Word level bitwidth reduction for unbounded hardware model checking. Form. Methods Syst. Des., 35(1):56–72, 2009.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In PLDI 01: Programming Language Design and Implementation, Snowbird, UT, June 2001.

[Boo] URL: http://fmv.jku.at/boolector.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In Proceedings of the 27th Design Automation Conference, pages 40–45, Orlando, FL, June 1990.

[Bru08] R. Bruttomesso. RTL Verification: From SAT to SMT(BV). PhD thesis, University of Trento, 2008. Available at ”URL: http://www.inf.unisi.ch/postdoc/bruttomesso”.

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

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Proceedings Workshop on Logics of Programs, pages 52–71, Berlin, 1981. Springer-Verlag. LNCS 131.

[CFR$^+$89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In POPL’89: Proceedings of the Sixteenth ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 25–35, New York, NY, USA, 1989. ACM.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. Model Checking. MIT Press, Cambridge, MA, 1999.

[Cho97] C.-T. Chou. Synchronous Verilog: A proposal, 1997. Available at http://home.pacbell.net/ctchou/sv0.ps.gz.

[CKZ96] E. Clarke, M. Khaira, and X. Zhao. Word level model checking–avoiding the pentium FDIV error. In 33rd Design Automation Conference (DAC’96), DAC’96, pages 645–648, New York, NY, USA, 1996.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. An Introduction to Algorithms. McGraw-Hill, New York, 1990.

[Cum02] C. Cummings. The fundamentals of efficient synthesizable finite. state machine design using NC-Verilog and Build Gates. In ICU ’02: Proceedings of the 2002 International Cadence Users Group Conference, San Jose, CA, USA, 2002.

[DdM06a] B. Duterte and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In Eighteenth International Conference on Computer-Aided Verification (CAV’06), pages 81–94, Seattle, WA, August 2006. LNCS 4144.

[DdM06b] B. Dutertre and L. de Moura. Integrating Simplex with DPLL(T). Technical Report SRI-CSL-06-01, SRI International, 2006.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. Communications of the ACM, 5:394–397, 1962.

[dMR02] L. de Moura and H. Reuß. Lemmas on demand for satisfiability solvers. In Fifth International Symposium on the Theory and Application of Satisfiability Testing (SAT'02), Cincinnati, OH, May 2002.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. Journal of the Association for Computing Machinery, 7(3):201–215, July 1960.

[DPL] URL: http://www.lsi.upc.edu/ oliveras/bclt-main.html.

[Dre04] R. Drechsler. Using word-level information in formal hardware verification. Autom. Remote Control, 65(6):963–977, 2004.

[EB05] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT 2005), pages 61–75, St. Andrews, UK, June 2005. Springer-Verlag. LNCS 3569.

[FJOS03] C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proof explication. In W. A. Hunt, Jr. and F. Somenzi, editors, Fifteenth Conference on Computer Aided Verification (CAV'03), pages 355–367. Springer-Verlag, Berlin, July 2003. LNCS 2725.

[GBT07] Y. Ge, C. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In CADE, pages 167–182, 2007.

[GdM09] Y. Ge and L. de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In 21st International Conference on Computer-Aided Verification (CAV'09), pages 306–320, Grenoble, France, 2009.

[GG08] M. Ganai and A. Gupta. Completeness in SMT-based BMC for software programs. In Design, Automation and Test in Europe (DATE'08), Munich, Germany, March 2008.

[GHN$^+$04] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, Sixteenth Conference on Computer Aided Verification (CAV'04), pages 175–188. Springer-Verlag, Berlin, July 2004. LNCS 3114.

[GMP] URL: http://gmplib.org.

[GN02] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In Proceedings of the Conference on Design, Automation and Test in Europe, pages 142–149, Paris, France, March 2002.

[Gor] M. J. C. Gordon. Synthesizable Verilog: syntax and semantics. Available at http://www.cl.cam.ac.uk/users/mjcg/Verilog/V/V.ps.gz.

[GTG06] M. K. Ganay, M. Talupur, and A. Gupta. SDSAT: Tight integration of small domain encoding and lazy abstraction in a separation logic solver. In International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'06), pages 135–150, Vienna, Austria, March 2006. LNCS 3920.

[Ica] URL: http://www.icarus.com/eda/verilog.

[IEE06] IEEE Standard for Verilog Hardware Description Language, IEEE STD 1364-2005, 2006. Available at http://ieeexplore.ieee.org.

[IPC03] M. K. Iyer, G. Parthasarathy, and K.-T. Cheng. SATORI – a fast sequential SAT engine for circuits. In Proceedings of the International Conference on Computer-Aided Design, pages 320–325, San Jose, CA, November 2003.

[IYG$^+$05] F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-Soft: Software verification platform. In Seventeenth International Conference on Computer-Aided Verification(CAV'05), pages 301–306, 2005.

[JDB95] R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In Proceedings of the International Conference on Computer-Aided Design, pages 2–6, San Jose, CA, November 1995.

[JHS05] H. Jin, H. Han, and F. Somenzi. Efficient conflict analysis for finding all satisfying assignments of a Boolean circuit. In International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'05), pages 287–300, April 2005. LNCS 3440.

[Joh01] P. Johannsen. Speeding Up Hardware Verification by Automated Data Path Scaling. PhD thesis, Christian-Albrechts-University, Kiel, 2001.

[JS04] H. Jin and F. Somenzi. CirCUs: A hybrid satisfiability solver. In International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), Vancouver, Canada, May 2004.

[JS05] H. Jin and F. Somenzi. Prime clauses for fast enumeration of satisfying assignments to Boolean circuits. In Proceedings of the Design Automation Conference, pages 750–753, Anaheim, CA, June 2005.

[Kar88] K. Karplus. Representing Boolean functions with if-then-else DAGs. In Technical Report UCSC-CRL-88-28, Board of Studies in Computer Engineering, University of California at Santa Cruz, Santa Cruz, CA 95064, December 1988.

[KGW10] S. Kundu, M. K. Ganai, and C. Wang. Contessa: Concurrency testing augmented with symbolic analysis. In 22nd International Conference on Computer-Aided Verification (CAV'10), pages 127–131, Edinburgh, UK, 2010.

[KJJP09] A. Kölbl, R. Jacoby, H. Jain, and C. Pixley. Solver technology for system-level to RTL equivalence checking. In DATE, pages 196–201, 2009.

[KJR$^+$08] H. Kim, H. Jin, K. Ravi, P. Spacek, J. Pierce, R. P. Kurshan, and F. Somenzi. Application of formal word-level analysis to constrained random simulation. In Twentieth Conference on Computer Aided Verification (CAV'08), pages 487–490, Princeton, NJ, July 2008. LNCS 5123.

[KJS07a] H. Kim, H. Jin, and F. Somenzi. Disequality management in integer difference logic via finite instantiations. Journal on Satisfiability, Boolean Modeling and Computation, 3:47–66, 2007.

[KJS07b] H. Kim, H. Jin, and F. Somenzi. Disequality management in integer difference logic via finite instantiations. Journal on Satisfiability, Boolean Modeling and Computation, 3:47–66, 2007.

[KK07] N. Kitchen and A. Kuelhmann. Stimulus generation for constrained random simulation. In IEEE/ACM Int. Conference on Computer Aided Design (ICCAD), November 2007.

[KOSS04] D. Kroening, J. Ouaknine, S. Seshia, and O. Strichman. Abstraction-based satisfiability solving of Presburger arithmetic. In R. Alur and D. Peled, editors, Sixteenth Conference on Computer Aided Verification (CAV'04), pages 308–320. Springer-Verlag, Berlin, July 2004. LNCS 3114.

[KS06] H. Kim and F. Somenzi. Finite instantiations for integer difference logic. In Formal Methods in Computer Aided Design (FMCAD'06), pages 31–38, San Jose, CA, November 2006.

[KSJ09] H. Kim, F. Somenzi, and H. Jin. Efficient term-ITE conversion for satisfiability modulo theories. In Twelfth International Conference on Theory and Applications of Satisfiability Testing (SAT 2009), pages 195–208, Swansea, UK, June 2009. Springer-Verlag. LNCS 5584.

[LM05] S. Lahiri and M. Musuvathi. An efficient Nelson-Oppen decision procedure for difference constraints over rationals. In Third International Workshop on Pragmatical Aspects of Decision Procedures in Automated Reasoning (PDPAR'05), pages 2–9, Edinburgh, UK, July 2005. To appear in ENTCS.

[LNO06] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT techniques for fast predicate abstraction. In Eighteenth International Conference on Computer Aided Verification, CAV'06, volume 4144 of Lecture Notes in Computer Science, pages 413–426. Springer, 2006.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages, pages 97–107, New Orleans, January 1985.

[LS04] S. K. Lahiri and S. A. Seshia. The UCLID decision procedure. In R. Alur and D. Peled, editors, Sixteenth Conference on Computer Aided Verification (CAV'04), pages 475–478. Springer-Verlag, Berlin, July 2004. LNCS 3114.

[Mat] URL: http://www.smtcomp.org/2009.

[MC99] D. Mills and C. Cummings. RTL coding styles that yield simulation and synthesis mismatches. In SNUG '99: Proceedings of the 1999 Synopsys Users Group Conference, San Jose, CA, USA, 1999.

[McM94] K. L. McMillan. Symbolic Model Checking. Kluwer Academic Publishers, Boston, MA, 1994.

[MKMR10] Patrick O'Neil Meredith, Michael Katelman, José Meseguer, and Grigore Roşu. A formal executable semantics of Verilog. In Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'10), pages 179–188. IEEE, 2010.

[MMZ$^+$01] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In Proceedings of the Design Automation Conference, pages 530–535, Las Vegas, NV, June 2001.

[MS96] J. P. Marques-Silva and K. A. Sakallah. Grasp—a new search algorithm for satisfiability. In Proceedings of the International Conference on Computer-Aided Design, pages 220–227, San Jose, CA, November 1996.

[NO05] R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In Seventeenth Conference on Computer Aided Verification (CAV'05), pages 321–334. Springer-Verlag, Berlin, July 2005. LNCS 3576.

[NO08] G. Nelson and D. Oppen. Simplification by cooperating decision procedures. In ACM Transactions on Programming Languages and Systems, 1(2):245-257, October 2008.

[NORCR07] Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Challenges in Satisfiability Modulo Theories. In Franz Baader, editor, Eighteenth International Conference on Rewriting Techniques and Applications, RTA'07, volume 4533 of Lecture Notes in Computer Science, pages 2–18. Springer, 2007.

[NW88] G. L. Nemhauser and L. A. Wolsey. Integer and combinatorial optimization. Wiley-Interscience, New York, NY, USA, 1988.

[Ope] URL: http://opencores.org.

[PICB05] G. Parthasarathy, M. K. Iyer, K.-T Cheng, and F. Brewer. RTL SAT simplification by Boolean and interval arithmetic reasoning. In ICCAD '05: Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design, pages 297–302, 2005.

[Pre] URL: http://fmv.jku.at/precosat.

[PRSS02] A. Pnueli, Y. Rodeh, O. Strichman, and M. Siegel. The small model property: How small can it be? Journal of Information and Computation, 178(1):279–293, October 2002.

[Roe06] K. Roe. The heuristic theorem prover: Yet another SMT modulo theorem prover. In Eighteenth International Conference on Computer-Aided Verification (CAV'06), pages 467–470, Seattle, WA, aug 2006.

[RS04] K. Ravi and F. Somenzi. Minimal assignments for bounded model checking. In International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'04), pages 31–45, Barcelona, Spain, March-April 2004. LNCS 2988.

[SMTa] URL: http://smtcomp.org/.

[SMTb] URL: http://www.csl.sri.com/users/demoura/smt-comp/results-qf_idl.shtml.

[Ter] URL: http://www.altera.com/support/examples/verilog/verilog.html.

[TSSP04] M. Talupur, N. Sinha, O. Strichman, and A. Pnueli. Range allociation for separation logic. In R. Alur and D. Peled, editors, Sixteenth Conference on Computer Aided Verification (CAV'04), pages 148–161. Springer-Verlag, Berlin, July 2004. LNCS 3114.

[Ver] URL: http://www.verilog.com.

[VIS] URL: http://vlsi.colorado.edu/∼vis.

[VVB] Vis verification benchmarks. http://vlsi.colorado.edu/∼vis.

[WIGG05] C. Wang, F. Ivancic, M. Ganai, and A. Gupta. Deciding separation logic formulae by SAT and incremental negative cycle elimination. In Logic for Programming Artificial Intelligence and Reasoning (LPAR'2005), Montego Bay, Jamaica, December 2005.

[WSBK07] M. Wedler, D. Stoffel, R. Brinkmann, and W. Kunz. A normalization method for arithmetic data-path verification. In IEEE Trans. on CAD of Integrated Circuits and Systems, volume 26, pages 1909–1922, 2007.

[WVS83] P. Wolper, M. Y. Vardi, and A. P. Sistla. Reasoning about infinite computation paths. In Proceedings of the 24th IEEE Symposium on Foundations of Computer Science, pages 185–194, 1983.

[XHHLB08] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. Journal of Artificial Intelligence Research, 32:565–606, 2008.

[Yic] URL: http://yices.csl.sri.com.

[YM06] Y Yu and S. Malik. Lemma learning in SMT on linear constraints. In A. Biere and C. P. Gomes, editors, Proceedings of Theory and Applications of Satisfiability Testing – SAT 2006, pages 142–155, August 2006.

[YSP$^+$99] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz. Modeling design constraints and biasing in simulation using BDDs. In IEEE/ACM Int. Conference on Computer Aided Design (ICCAD), pages 584–590, 1999.

[Z3] URL: http://research.microsoft.com/en-us/um/redmond/projects/z3.

[ZKC01] Z. Zeng, P. Kalla, and M. Ciesielski. LPSAT: A unified approach to RTL satisfiability. In In Proc. DATE, pages 398–402, 2001.

Appendix A

Tables for Comparison

In this chapter, we list the tables comparing the different encodings for hardware designs used in Chapter 5. Table A.1 shows the detailed results of Fig. 6.1. The table shows the model names with different unrolling depths and the CPU times of BV and LIA solvers. The BV solvers used are Z3-2.8 (**Z3-B**), Boolector-1.4 (**BL-B**) and Beaver (**BE-B**). The LIA solvers used are Yices-1.0.28 (**YI-L**), Z3-2.8 (**Z3-L**) and MathSAT-4.3 (**MA-L**). The fifth column (**AVG-B**) shows the average CPU times of the BV solvers, and the ninth column (**AVG-L**) shows the average CPU times of the LIA solvers. The last row shows the number of timeouts for each solver. The timeout was set to 1000 seconds. Table A.2 shows the detailed results of Fig. 6.6. The table shows the selected encodings for the designs and the CPU times of the BV and LIA solvers. Table A.2 shows the detailed results of Fig. 6.3. Table A.4 shows the detailed results of Fig. 6.4. Table A.5 shows the detailed results of Fig. 6.5. Table A.6 shows the detailed results of Fig. 6.7.

Model	Z3-B	BL-B	BE-B	AVG-B	YI-L	Z3-L	MA-L	AVG-L
am2910-tr50	16.48	4.48	3.79	8.25	156.47	22.69	8.13	62.43
am2910-tr100	86.76	13.79	11	37.18	1000	219.51	32.35	417.29
am2910-tr150	327.62	27.5	20.45	125.19	1000	1000	87.98	695.99
bakery-tr5	8.16	3.83	2.64	4.88	11.03	9.87	24.61	15.17
bakery-tr10	234.83	13.62	7.37	85.27	389.46	501.75	281.5	390.9
bakery-tr15	1000	34.61	15.68	350.1	1000	1000	1000	1000
blackjack-tr30	44.01	34.49	17.26	31.92	505.61	117.33	1000	540.98
blackjack-tr40	108.52	77.76	37.04	74.44	1000	560.18	1000	853.39
blackjack-tr50	154.72	133.41	53.99	114.04	1000	1000	1000	1000
cube-tr10	24	1.52	1.46	8.99	5.04	4.68	29.14	12.95
cube-tr15	78.71	5.34	5.32	29.79	34.46	57.54	1000	364
cube-tr20	699.04	254.92	285.03	413	243.23	693.54	1000	645.59
FPMult-tr5	0.54	0.85	0.84	0.74	1000	237.42	1000	745.81
FPMult-tr10	4.02	3.82	3.93	3.92	1000	1000	1000	1000
FPMult-tr15	19.44	10.85	14.33	14.87	1000	1000	1000	1000
palu-tr10	2.59	1.81	1.9	2.1	17.97	16.53	1000	344.83
palu-tr20	8.31	4.57	5.75	6.21	375.95	123.24	1000	499.73
palu-tr30	18.54	8.56	8.66	11.92	1000	560.6	1000	853.53
retherRTF-tr70	26.89	42.8	48.28	39.32	60.4	241.73	93.18	131.77
retherRTF-tr80	34.32	48.53	55.46	46.1	37.55	415.85	248.03	233.81
retherRTF-tr90	50.82	55.67	64.27	56.92	37.28	111.95	99.67	82.97
swap-tr5	0.48	0.06	0.07	0.2	0.68	0.8	1.47	0.98
swap-tr10	179.01	110.7	173.95	154.55	1000	1000	911.76	970.59
swap-tr15	1000	1000	1000	1000	1000	1000	1000	1000
vMiim-tr100	102.75	10.35	12.75	41.95	1000	158.29	165.2	441.16
vMiim-tr150	148.92	15.12	24.25	62.76	1000	920.71	469.75	796.82
vMiim-tr200	428.16	24.4	46.1	166.22	1000	1000	919.44	973.15
cf-fir-tr10	230.05	191.18	122.82	181.35	0.99	9.28	5.26	5.18
cf-fir-tr20	453.72	212.53	52.03	239.43	6.84	154.42	21.52	60.93
cf-fir-tr30	627.57	764.11	82.02	491.23	14.2	129.14	60.54	67.96
FIFOs-tr8	10.27	258.8	461.73	243.6	13.3	16.28	8.99	12.86
FIFOs-tr10	73.49	1000	1000	691.16	64.96	79.92	41.27	62.05
FIFOs-tr12	331.54	1000	1000	777.18	335.74	526.53	174.58	345.62
fir-tr5	54.52	403.43	307.9	255.28	19.73	17.13	24.16	20.34
fir-tr10	1000	1000	1000	1000	48.47	33.51	80.93	54.3
fir-tr15	1000	1000	1000	1000	86.11	53.13	181.69	106.98
minMax-tr100	1000	76.73	85.3	387.34	12.68	15.38	25.73	17.93
minMax-tr200	1000	287.66	276.4	521.35	66.27	72.95	168.23	102.48
minMax-tr300	1000	595.85	656.93	750.93	167.33	153.58	419.96	246.96
adder-chain-tr10	1000	1000	547.35	849.12	17.02	26.01	117.4	53.48
adder-chain-tr15	1000	1000	1000	1000	43.59	89.11	528.92	220.54
adder-chain-tr20	1000	1000	1000	1000	127.71	488.07	1000	538.59
timeout-tr40	123.9	44.46	35.48	67.95	48.62	16.21	146.01	70.28
timeout-tr60	295.95	87.15	74.54	152.55	106.33	64.34	102.63	91.1
timeout-tr80	1000	181.53	123.49	435.01	443.05	164.24	1000	535.76
Timeout	11	8	7	5	14	8	15	5

Table A.1: Comparison of using BV solvers and LIA solvers on Verilog design

Model	Sel	Z3-B	BL-B	BE-B	AVG-B	YI-L	Z3-L	MA-L	AVG-L
am2910-tr50	BV	16.48	4.48	3.79	8.25	216.06	31.49	8.53	85.36
am2910-tr100		86.76	13.79	11	37.18	1000	143.39	33.38	392.26
am2910-tr150		327.62	27.5	20.45	125.19	1000	1000	64.79	688.26
bakery-tr5	BV	8.16	3.83	2.64	4.88	11.09	9.75	24.43	15.09
bakery-tr10		234.83	13.62	7.37	85.27	391.94	500.94	280.62	391.17
bakery-tr15		1000	34.61	15.68	350.1	1000	1000	1000	1000
blackjack-tr30	BV	44.01	34.49	17.26	31.92	448.84	119.9	606.51	391.75
blackjack-tr40		108.52	77.76	37.04	74.44	871.67	306.77	1000	726.15
blackjack-tr50		154.72	133.41	53.99	114.04	1000	718.58	1000	906.19
cube-tr10	BV	24	1.52	1.46	8.99	5.02	4.23	15.18	8.14
cube-tr15		78.71	5.34	5.32	29.79	30.61	18.87	1000	349.83
cube-tr20		699.04	254.92	285.03	413	457.29	380.94	1000	612.74
FPMult-tr5	BV	0.54	0.85	0.84	0.74	1000	117.23	1000	705.74
FPMult-tr10		4.02	3.82	3.93	3.92	1000	1000	1000	1000
FPMult-tr15		19.44	10.85	14.33	14.87	1000	1000	1000	1000
palu-tr10	BV	2.59	1.81	1.9	2.1	2.87	6.28	1000	336.38
palu-tr20		8.31	4.57	5.75	6.21	12.89	12.84	1000	341.91
palu-tr30		18.54	8.56	8.66	11.92	20.56	136.29	1000	385.62
retherRTF-tr70	BV	26.89	42.8	48.28	39.32	26.86	180.79	175.62	127.76
retherRTF-tr80		34.32	48.53	55.46	46.1	17.26	167.96	130.51	105.24
retherRTF-tr90		50.82	55.67	64.27	56.92	26.84	132.37	177.71	112.31
swap-tr5	BV	0.48	0.06	0.07	0.2	0.67	0.89	1.49	1.02
swap-tr10		179.01	110.7	173.95	154.55	1000	1000	912.29	970.76
swap-tr15		1000	1000	1000	1000	1000	1000	1000	1000
vMiim-tr100	BV	102.75	10.35	12.75	41.95	1000	158.05	163.88	440.64
vMiim-tr150		148.92	15.12	24.25	62.76	1000	859.99	466.77	775.59
vMiim-tr200		428.16	24.4	46.1	166.22	1000	1000	917.49	972.5
cf-fir-tr10	LIA	230.05	191.18	122.82	181.35	1.03	9.42	5.09	5.18
cf-fir-tr20		453.72	212.53	52.03	239.43	6.68	154.15	21.22	60.68
cf-fir-tr30		627.57	764.11	82.02	491.23	14.27	129.38	60.62	68.09
FIFOs-tr8	LIA	10.27	258.8	461.73	243.6	13.18	16.21	8.98	12.79
FIFOs-tr10		73.49	1000	1000	691.16	64.67	79.69	40.76	61.71
FIFOs-tr12		331.54	1000	1000	777.18	334.65	534.38	174.35	347.79
fir-tr5	LIA	54.52	403.43	307.9	255.28	19.66	17.43	24.22	20.44
fir-tr10		1000	1000	1000	1000	48.71	33.62	81.2	54.51
fir-tr15		1000	1000	1000	1000	85.06	52.87	182.5	106.81
minMax-tr100	LIA	1000	76.73	85.3	387.34	12.71	15.36	25.85	17.97
minMax-tr200		1000	287.66	276.4	521.35	65.35	73.36	168.13	102.28
minMax-tr300		1000	595.85	656.93	750.93	167.86	153.7	419.8	247.12
adder-chain-tr10	LIA	1000	1000	547.35	849.12	17.02	26.01	110.48	51.17
adder-chain-tr15		1000	1000	1000	1000	43.59	89.11	521.46	218.05
adder-chain-tr20		1000	1000	1000	1000	127.71	488.07	1000	538.59
timeout-tr40	LIA	123.9	44.46	35.48	67.95	16.06	30.82	21.36	22.75
timeout-tr60		295.95	87.15	74.54	152.55	210.91	65.51	55.3	110.57
timeout-tr80		1000	181.53	123.49	435.01	365.8	49.36	563.09	326.08
Timeout		11	8	7	5	12	7	13	4

Table A.2: Comparison of using BV solvers and LIA solvers (Bit-Blast) on training set of Verilog designs

Model	Val Enum	No Val Enum
cf-fir-tr100	162.4	588.82
cf-fir-tr101	164.63	469.67
cf-fir-tr102	169.6	568.52
cf-fir-tr103	178.28	633
cf-fir-tr104	186.45	688.95
cf-fir-tr105	186.48	612.76
cf-fir-tr106	187.76	659.92
cf-fir-tr107	236.13	643.59
cf-fir-tr108	278.38	893.66
cf-fir-tr109	207.1	848
cf-fir-tr110	206.58	698.94
fir-tr10	48.26	48.78
fir-tr15	84.22	84.53
fir-tr20	114.03	114.27
fir-tr25	179.66	180.09
fir-tr30	236.46	237.75
fir-tr35	331.2	331.38
fir-tr40	411.84	409.91
fir-tr45	506.41	507.9
fir-tr50	678.02	684.64
fir-tr5	19.45	19.49
adder-chain-tr10	11.72	11.82
adder-chain-tr11	15.9	16.34
adder-chain-tr12	16.89	17.28
adder-chain-tr13	28.8	28.57
adder-chain-tr14	26.93	27.11
adder-chain-tr15	48.78	49.12
adder-chain-tr16	48.33	49.63
adder-chain-tr17	64.25	64.83
adder-chain-tr18	62.01	62.03
adder-chain-tr19	102.63	101.96
adder-chain-tr20	72.82	73.85

Table A.3: Comparison of LIA encodings with and without value enumeration

Model	**Term-ITE**	**Fresh var**
cf-fir-tr100	162.4	1000
cf-fir-tr101	164.63	1000
cf-fir-tr102	169.6	1000
cf-fir-tr103	178.28	1000
cf-fir-tr104	186.45	1000
cf-fir-tr105	186.48	1000
cf-fir-tr106	187.76	1000
cf-fir-tr107	236.13	1000
cf-fir-tr108	278.38	1000
cf-fir-tr109	207.1	1000
cf-fir-tr110	206.58	1000
fir-tr10	48.26	102.01
fir-tr15	84.22	158.76
fir-tr20	114.03	229.35
fir-tr25	179.66	755.39
fir-tr30	236.46	407.44
fir-tr35	331.2	479.38
fir-tr40	411.84	977.49
fir-tr45	506.41	761.67
fir-tr50	678.02	1000
fir-tr5	19.45	44.54
adder-chain-tr10	11.72	109.83
adder-chain-tr11	15.9	196.06
adder-chain-tr12	16.89	294.2
adder-chain-tr13	28.8	509.37
adder-chain-tr14	26.93	607
adder-chain-tr15	48.78	1000
adder-chain-tr16	48.33	1000
adder-chain-tr17	64.25	1000
adder-chain-tr18	62.01	1000
adder-chain-tr19	102.63	1000
adder-chain-tr20	72.82	1000

Table A.4: Comparison of LIA encoding with Term-ITEs and LIA encoding with fresh variables

Model	**LIA with Bit-Blast**	**LIA**	**BV ∪ LIA**
palu-tr5	0.21	0.75	43.78
palu-tr10	1.25	9.3	1000
palu-tr15	2.97	26.7	1000
palu-tr20	5.45	75.29	1000
palu-tr25	9.32	152.82	1000
palu-tr30	21.61	330.01	1000
palu-tr35	27.14	477.78	1000
palu-tr40	31.44	723.35	1000
palu-tr45	53.48	1000	1000
palu-tr50	80.23	1000	1000
palu-tr55	60.24	1000	1000
palu-tr60	101.81	1000	1000
palu-tr65	194.64	1000	1000
palu-tr70	195.73	1000	1000
palu-tr75	332.02	1000	1000
palu-tr80	311.31	1000	1000
palu-tr85	289.03	1000	1000
palu-tr90	300.2	1000	1000
palu-tr95	642.66	1000	1000
palu-tr100	659.18	1000	1000

Table A.5: Comparison of LIA with Bit-Blast, pure LIA and BV ∪ LIA encoding

Model	**Sel**	**Z3-B**	**BL-B**	**BE-B**	**AVG-B**	**YI-L**	**Z3-L**	**MA-L**	**AVG-L**
cordic-tr8		71.63	101.62	150.06	57.75	1000	0.15	0	333.38
cordic-tr10	BV	463.07	147.19	251.78	203.42	1000	1000	1000	666.67
cordic-tr12		312.37	124.56	197.71	145.64	1000	1000	1000	666.67
daio-receiver-tr50		2.36	2.62	1.24	1.66	11.89	12.81	12.39	8.23
daio-receiver-tr60	BV	7.57	3.15	1.62	3.57	34.6	17.28	19.07	17.29
daio-receiver-tr70		6.46	3.69	1.8	3.38	28.05	26.4	28.56	18.15
dekker-tr50		13.59	4.61	3.23	6.07	53.25	27.12	37.28	26.79
dekker-tr60	BV	41.58	5.6	4.23	15.73	120.06	41.48	75.08	53.85
dekker-tr70		40.59	7.25	5.08	15.95	95.32	72.17	128.33	55.83
Unidec-tr50		125.84	5.86	6.92	43.9	1000	1000	533.34	666.67
Unidec-tr60	BV	202.41	7.5	7.79	69.97	1000	1000	598.78	666.67
Unidec-tr70		229.63	9.05	8.81	79.56	1000	902.36	1000	634.12
soc-ram-tr4		29.84	227.29	604.97	85.71	31.31	34.71	8.7	22.01
soc-ram-tr5	LIA	167.2	807.33	1000	324.84	71.54	67.27	17.23	46.27
soc-ram-tr6		196.8	1000	1000	398.93	158.39	324.62	29.31	161
altmult-accum-tr5		232.71	394.1	8.81	208.94	1.8	5.44	3.9	2.41
altmult-accum-tr7	LIA	1000	1000	1000	1000	283.76	115.63	42.91	133.13
altmult-accum-tr9		1000	1000	1000	1000	592.33	1000	187.89	530.78
Timeout		2	3	4	2	6	5	3	0

Table A.6: Comparison of using BV solvers and LIA solvers (Bit-Blast) on evaluation set of Verilog designs