TECHNIQUES FOR EFFICIENT FORMAL VERIFICATION USING BINARY DECISION DIAGRAMS

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

By
Alan John Hu
December 1995

© Copyright 1995 by Alan John Hu
All Rights Reserved

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

David L. Dill
(Principal Advisor)

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

Robert K. Brayton

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

Vaughan R. Pratt

Approved for the University Committee on Graduate Studies:

To my parents
For always standing behind me
But never pushing...

Abstract

The appeal of automatic formal verification is that it's automatic — minimal human labor and expertise should be needed to get useful results and counterexamples. BDD(binary decision diagram)-based approaches have promised to allow automatic verification of complex, real systems. For large classes of problems, however, (including many distributed protocols, multiprocessor systems, and network architectures) this promise has yet to be fulfilled. Indeed, the few successes have required extensive time and effort from sophisticated researchers in the field. Clearly, techniques are needed that are more sophisticated than the obvious direct implementation of theoretical results.

This thesis addresses that need, emphasizing an application domain that has been particularly difficult for BDD-based methods — high-level models of systems or distributed protocols — rather than gate-level descriptions of circuits. Additionally, the emphasis is on providing useful debugging information for the designer rather than on certifying correctness. Accordingly, I only consider a simple verification paradigm (that all reachable states of a system satisfy a propositional logic formula) and seek to make it applicable to complex, real systems, rather than devising theoretically more complicated verification paradigms.

I identify several common obstacles to BDD-based automatic formal verification and propose techniques to overcome them. Specifically, (1) I consider the difficulty of specifying realistic high-level designs using BDDs and give a set of high-level language constructs and the accompanying algorithms for automatic translation that are expressive enough to specify real designs, yet are still efficiently handled by BDDs; (2) I describe how to compute images (pre- and post-conditions) of sets of states efficiently

through the high-level description of the system being verified without ever building the BDD for the transition relation of the system, thereby avoiding a major BDD-size blowup; and (3) I highlight why BDD-size blowup is so common when performing high-level verification and propose techniques based on functional dependencies and on implicitly conjoined lists of BDDs to avoid some of these blowups. I have implemented these techniques in the Ever verification system and applied them to several examples, illustrating the effectiveness of the new techniques in expanding the range of problems that can routinely be verified automatically.

Acknowledgments

Dissertation tradition apparently dictates that I rattle off a lengthy list of my nearest and dearest friends, sprinkle in an inside joke or two, and disclaim that just because I forgot to list someone doesn't mean that I'm not eternally grateful to him or her. I choose to break with this tradition not out of lack of gratitude, but rather from an overabundance of it. Any delineation of people to thank excludes, unavoidably and arbitrarily, other people who also deserve my thanks. On what basis do I thank person A who accrued x UGUs (universal gratitude units), yet exclude person B who has only accrued $x - \epsilon$? And doesn't person C, who has accrued $10x$ UGUs, deserve special mention? Worse, the UGU doesn't even exist, for the intellectual and personal debts I have incurred are multidimensional, incomparable quantities. How can I rank a net.friend whose quick wit and insightful intellect brightened many a day versus a dance partner whose grace, poise, and beauty (not to mention quick wit and insightful intellect) brightened many an evening? How do I valuate my intellectual indebtedness to the teachers and mentors who taught me theoretical computer science versus those who taught me about electronic design automation? How do I compare my eternal gratitude to the friend who has countless times kept my computer running versus the friend who held my head in her lap while I wept with self-doubt? In short, how can I list a set of names without committing a disservice to those I list as well as to those I do not? Allow me instead to paint with broad strokes.

That being said, I do need to single out my advisor for conning me into a wonderfully interesting research area and then putting up with me for the better part of a decade with patience, humor, good advice (research and otherwise), friendship, and money.

A consequence of the research area is the excellent set of researchers I have met, at Stanford, across the Bay, across the country, and around the world. They have unfailingly provided intellectual stimulation and comraderie.

I consider myself very fortunate to have spent a decade at Stanford. The faculty and my fellow students are truly outstanding and inspirational. Even with all the downsides of student life, if I had the option to spend my career working with these people, I would leap at the chance.

And despite my generally rude and antisocial personality, I've somehow been blessed with a wonderful set of friends and family members. (Another reason not to list names is to deny this information to long-distance carrier telemarketers.) From across the vast asynchrony in time and space of soc.culture.asian.american to the intimate closeness of a tango, whether (very) patiently explaining to me the intricacies (basics?) of ω-automata or of T'ai Chi Ch'uan, from going up and down mountains to hitting things over nets, whether sharing a concert, a meal, or an impromptu chat in the hallway, they have made my life so much fuller, so much richer, so much happier.

To all of you who have touched my life in your myriad ways, thank you. May your friends be as kind to you as you have been to me.

Contents

1 Introduction **1**

1.1 Motivation and Philosophy 1

1.2 Background 3

1.2.1 Proof-Based Methods 3

1.2.2 State-Exploration Methods and Model Checking 6

1.2.3 Binary Decision Diagrams 9

1.2.4 Symbolic Model Checking 13

1.3 What This Thesis Is About 17

2 Translation into BDDs **21**

2.1 High-Level Language Constructs 21

2.2 Translation 24

2.2.1 Constants and Variables 25

2.2.2 Arithmetic and Logical Expressions 28

2.2.3 Assignment 30

2.2.4 Control Flow 32

3 Efficient Image Computation **35**

3.1 Background 35

3.2 Image Computation Procedure 38

4 BDD Blow-Up Representing Sets of States **41**

4.1 A Historical Note 42

4.2 BDD Size Intuition 43

4.3 High-Level Verification Is Hard for BDDs 47

5 Functionally Dependent Variables **51**

5.1 Introduction 51
5.2 Theoretical Basis 53
5.3 Implementation 57
5.4 A Benchmark Example 61

6 Implicitly Conjoined BDDs **73**

6.1 Introduction 73
6.2 Theoretical Basis 75
6.3 Heuristics 78
6.3.1 A Simple Approach 78
6.3.2 Greedy Evaluation 79
6.3.3 Exact Termination Testing 81
6.4 Experimental Results 87
6.4.1 A Typed FIFO Buffer 87
6.4.2 A Simple Network 89
6.4.3 A Moving-Average Filter 91
6.4.4 A Simple Pipelined Processor 95
6.5 Comments 98

7 Contributions and Future Work **101**

7.1 Contributions 101
7.2 Future Work 102

A Ever Verifier Reference Manual **107**

A.1 Introduction 107
A.2 A Tutorial Example 109
A.3 Ever Language Details 116
A.3.1 Lexical Units 116
A.3.2 Type System 118
A.3.3 Variable Versions 120

A.3.4 Expressions 121
A.3.5 Declarations 128
A.3.6 Commands 131

Bibliography **135**

List of Tables

5.1 Functionally Dependent Variables Results 71

6.1 Implicitly Conjoined BDDs Results: Typed FIFO Buffer 90
6.2 Implicitly Conjoined BDDs Results: Messages in Network 92
6.3 Implicitly Conjoined BDDs Results: Moving-Average Filter 96
6.4 Implicitly Conjoined BDDs Results: Simple Pipelined Processor . . . 99

List of Figures

1.1 CTL Model Checking Example 8
1.2 A Binary Decision Diagram (BDD) 10
1.3 BDD Variable Order Matters 12
1.4 Symbolic Model Checking Example 14

4.1 Fooling Set Intuition 45
4.2 Effect of Separating Related Variables 46
4.3 Variable Relationships in a Typical System-Level Model 48

5.1 Network Example 62
5.2 Proof of Theorem 5.3: BDD Nodes in a Square 65
5.3 Proof of Theorem 5.3: BDD Nodes on a Chessboard 67
5.4 Functionally Dependent Variables Results: Largest Intermediate BDD 69
5.5 Functionally Dependent Variables Results: Reachable State BDD . . 70

6.1 Backward Traversal with Implicitly Conjoined BDDs 76
6.2 Greedy Algorithm to Select Conjunctions in a List to Evaluate 82
6.3 Moving-Average Filter Example 95
6.4 Pipelined Processor Example 98

A.1 Link-Level Protocol Example 110

Chapter 1

Introduction

Chapter Overview

This chapter motivates my emphasis on automatic verification as a practical debugging tool, situates my approach as a state-exploration or model-checking approach rather than a theorem-proving one, and provides basic background on CTL model-checking, binary decision diagrams (BDDs), and how to use BDDs for my verification problem.

1.1 Motivation and Philosophy

Bugs cost money. Whenever an error creeps into a design, time and money must be spent to locate the problem and correct it, and the longer a bug evades detection, the harder and more expensive it is to fix. Indeed, the cost of making an engineering change rises by an order of magnitude at each successive stage of the design cycle [78]. Beyond the high cost of fixing bugs is the even higher cost of being late to market — a McKinsey and Company study indicates that being only six months late on a product with a five year life cycle results in one-third less total profit [25]. Should a bug slip through into the completed product, the results can be disastrous. For example, the Pentium FDIV public relations fiasco cost Intel Corporation a half *billion* dollars [40]. Detecting and eliminating bugs as early in the design cycle as possible is clearly an

economic imperative.

Working against this imperative, however, is the engineering reality that systems are becoming ever larger and more complicated. A person can no longer gain a reasonable assurance of correctness by simply staring at and thinking about a paper design. Worse, as design complexity increases, simulation times become prohibitive and coverage becomes poor, allowing numerous bugs to slip through to later stages of the design cycle. What is needed, therefore, is an aid to developing conceptually correct designs to start with and a supplement to simulation and testing in debugging the implementation of the conceptual design.

My interest in formal verification is to meet this need. Note that my motivation is explicitly economic. I have made no appeals to safety-critical systems, saving lives, or the moral superiority of proving systems correct. I do not discount the importance of such work. Indeed, I believe that safety-critical systems demand the most rigorous assurances of correctness available, especially when my own safety is at stake. But I also believe that for an enormous range of practical, real problems, pure financial profit is a compelling motivation to use formal verification *as a debugging tool* to speed a product to market, and my research focuses on this domain. Condensed to a slogan, my research aims to save money, not lives.[1]

Adopting a purely economic justification for formal verification has consequences. Fundamentally, the time and labor invested in verification must repay itself by helping the designer find and correct bugs quickly. Therefore, the emphasis must be on methods that are easy-to-use and highly automatic to minimize the time and labor invested and that provide counterexamples to help the designer find and correct the bugs. Furthermore, verification should be applied as early in the design cycle as possible, when bugs are cheaper and easier to fix. The goal is rapid, light-weight debugging support, rather than slow, laborious certification of correctness.

[1]For readers interested in safety-critical systems, Leveson [60] provides a comprehensive introduction.

1.2 Background

The goals of designing correct systems and finding and correcting bugs have been around for as long as people were designing systems. The idea of using formal methods to help these goals have been around almost as long.

1.2.1 Proof-Based Methods[2]

"In the beginning, there was Floyd."[3] Floyd introduced the first rigorous treatment of correctness of sequential programs [31]. The basic idea is to annotate each point in a program with an assertion. The program computes the correct result if (1) it terminates, (2) the state of the system before the program executes satisfies the assertion at the start of the program, (3) for each statement in the program, we can prove that the truth of the assertion before the statement implies the truth of the assertion following the statement, and (4) the assertion at the end of the program implies the result is correct. Of particular relevance to my work is Floyd's notion of a strongest verifiable consequent: Given an assertion that holds before a statement, the strongest verifiable consequent is the strongest assertion that must hold after executing that statement. Furthermore, Floyd gives rules to compute the strongest verifiable consequent automatically for several programming-language constructs and notes that these rules allow fully automatic verification in the absence of loops. We will see these ideas again later in this thesis.

An extensive array of formal treatments of program correctness have sprung from this start. Hoare reworked Floyd's ideas into a logical framework, introducing the notation $P\{Q\}R$ to denote the statement: If assertion P (the precondition) holds

[2]The terminology is somewhat problematic, since all formal methods are "proof-based" in that, by definition, they produce some sort of mathematical proof. I am stealing this terminology from Long, who contrasts "proof-based methods" — which emphasize the construction of a proof of correctness via axioms and inference rules for the specification language used — to "state-exploration methods" — which search the state space of an implementation to check that it satisfies its specification [62]. Emerson draws a similar distinction, labeling them "proof-theoretic" and "model-theoretic" [28].

[3]Quotation is from Lamport [58], referring to Floyd's seminal work [31].

before program Q executes, then assertion R (the postcondition) holds after Q completes [39]. An idea relevant to my thesis is that Hoare defined his axiom for assignment backwards from Floyd's — whereas Floyd starts from the precondition and computes the strongest verifiable consequent, Hoare starts from the postcondition and gives a means to compute the precondition. Dijkstra expanded in this direction, introducing the notion of the weakest precondition $wp(S, R)$, which is the weakest assertion about the state prior to the execution of program (or program fragment) S that guarantees that assertion R will hold afterwards [26]. Furthermore, Dijkstra provided rules to compute the weakest precondition of a program constructed from assignment statements and guarded command sets (which provide if-then-else, non-deterministic choice, and looping) by recursively propagating weakest preconditions backward through the structure of the program. (In the absence of loops, this approach also gives fully automatic verification.) Pratt's introduction of dynamic logic [79] unified earlier work in a clean conceptual framework that has served as the foundation for considerable further research, which lies beyond the scope of this thesis. Kozen and Tiuryn [56] and Cousot [24] provide recent surveys of this area.

The above methods were all designed to deal with programs that compute a result and then terminate, sometimes called transformative programs. In contrast, many important applications, generally referred to as reactive systems, are considered not to terminate: for example, operating systems, controllers, communication protocols, and hardware systems. Instead, correct behavior means the system is maintaining a continuing dialog with its environment. Thus, a different logical formalism is better suited to these applications.

Temporal logic has emerged as a main formalism for reactive systems. A temporal logic is essentially an ordinary predicate or propositional logic with the addition of modal operators for describing how the interpretation of symbols changes over time. Typical temporal operators include the next-time operator (generally written $\bigcirc$ or X), the eventuality operator (generally written $\Diamond$ or F), and the henceforth or always operator (generally written $\Box$ or G). For example, the sentence p asserts that p holds in the current state, the sentence Xp is true in the current state if p holds in the next state, and the sentence $\Box\Diamond p$ (always eventually p) is true if at all points in

the future, it is always true that eventually p will hold (in other words, p will hold infinitely often). Pnueli introduced the use of temporal logic for the formal verification of reactive systems [76], and Manna and Pnueli codified a general means to create a temporal logic proof system for any programming language [64]. As with program logics, the field of temporal logic has spawned considerable research, most of which lies beyond the scope of this thesis. Pnueli [77] surveys this area espousing a proof-based methodology; Emerson [28] provides a more recent survey and also covers the model-checking approach to temporal logic (discussed below).

Regardless of logical formalism, constructing proofs can be tedious and difficult, so some have turned to automatic theorem provers, achieving limited success. The field of automatic deduction and theorem proving is enormous, spanning both logic and AI, and extending far beyond the field of formal verification, not to mention this thesis. Wos *et al.* [87] provide a starting point to survey this area. Recent high-profile successes in the use of automatic theorem provers for formal verification include Cohn's verification of the Viper microprocessor [20] using the HOL theorem proving system [37, 36], Hunt's verification of the FM8502 microprocessor [46] using the Boyer-Moore theorem prover nqthm [4] (both of these microprocessors are very simple and were specifically designed for formal verification), and, very recently, Srivas and Miller's verification of a half-million transistor commercial microprocessor, the AAMP5 [84], using the PVS theorem prover [71, 72].

Despite the periodic success stories, proof-based methods remain notoriously time-consuming [60, p. 496], and "most of the 'automated' theorem-provers available today are semi-automated at best." [38, p. 220] Much progress has been made in automating proofs of correctness, but these methods have yet to have widespread industrial impact. The user must cleverly guide the proof process, and the generation of appropriate invariants for loops and inductions is particularly difficult. Furthermore, if the user is unable to prove correctness, no indication differentiates inadequate cleverness on the part of the user from the existence of a real bug in the design. Indeed, even proponents of automatic theorem provers have taken a keen interest in integrating state-exploration methods into their provers [52, 80], touting the increased automation and efficiency of state-exploration methods over proof-based methods. Over a

quarter century ago, Hoare wrote:

> The practice of supplying proofs for nontrivial programs will not become widespread until considerably more powerful proof techniques become available, and even then will not be easy.... At present, the method which a programmer uses to convince himself of the correctness of his program is to try it out in particular cases and to modify it if the results produced do not correspond to his intentions. After he has found a reasonably wide variety of example cases on which the program seems to work, he believes that it will always work. [39, p. 579]

A quarter century later, the situation has only started to change.

1.2.2 State-Exploration Methods and Model Checking

Of course, the world of simulation and testing hasn't been stationary. Real people have been designing ever larger and more complex real systems and have therefore developed more sophisticated means to simulate and test them. (For just two examples, Holzmann [41] overviews a wide variety of validation techniques for protocols, and Cheng and Agrawal [14] survey simulation and test generation techniques for VLSI.) In particular, if we can somehow exhaustively simulate every possible execution of a system, then we have actually proven (by exhaustive case analysis) the system correct. This idea provides a natural lead-in to the concept of model checking.

Model checking was invented by Clarke and Emerson [18] and represents an approach to verification fundamentally different from the proof-based methods described above. With proof-based methods, the user describes the system being verified using assertions in some logical formalism and attempts to deduce a proof, using inference rules for the logic being used, that the assertions imply the correctness property being checked. In contrast, model checking treats the entire reachability graph of the system being verified as a Kripke structure, and verification consists of checking whether or not the structure is a model of the correctness property being checked. For many logics, if the state space is finite, checking whether the (finite) Kripke structure is a

model of a formula is very efficient, using graph algorithms to traverse the Kripke structure.

Before presenting an example of model checking, let me introduce the logical notation used throughout this thesis as well as the temporal logic CTL used in the following example. Throughout the thesis, I will mix Boolean algebra notation and propositional logic notation in an effort to maximize clarity and brevity. Accordingly, the unary operator $\neg$ denotes logical negation, the binary operators $\cdot$ and $\wedge$ as well as juxtaposition denote conjunction, the binary operator $\oplus$ denotes exclusive-OR, the binary operators $+$ and $\vee$ denote disjunction, and the binary operator $\Rightarrow$ denotes logical implication. Operator precedence is in the order listed, with $\neg$ being highest. CTL (Computation Tree Logic) is a propositional, branching-time temporal logic frequently associated with model checking [18, 17]. In addition to atomic propositions and Boolean operators, CTL contains the usual forward time temporal operators: X (next time), F (eventually), G (always), and U (until). However, each use of any temporal operator must always be paired with a path quantifier: A (for all paths), or E (there exists a path). For example, the formula AGp is true in a state if for all paths starting from that state, p is always true; the formula EFp is true in a state if there exists a path starting in that state that eventually leads to a state where p is true; and the formula AFGp is not a valid CTL formula because the **G** is not controlled by a path quantifier, and in fact, this formula cannot be expressed in CTL.[4]

Let's consider the simple example in Figure 1.1(a). In this example, the Kripke structure has four states and ostensibly describes the transition structure of some system we wish to verify. Suppose we wish to check the CTL formula AG($r \Rightarrow$ AFs) that says that throughout the system, whenever r is true, s must inevitably be true now or in the future. Model checking proceeds by repeatedly traversing the graph, labeling all states that satisfy each subformula. Note that in CTL, the truth or falsehood of a subformula at a state does not depend on the formula in which it is embedded, so we can make a single pass through the CTL formula starting from

[4]This result follows from Emerson and Halpern [29, Theorem 4.4], since AFGp is equivalent to $\neg$EGF$\neg p$, which is equivalent to $\neg \mathrm{E} \overset{\infty}{\mathrm{F}} \neg p$ ("There does not exist a path where $\neg p$ holds infinitely often."), which is shown not to be expressible in CTL.

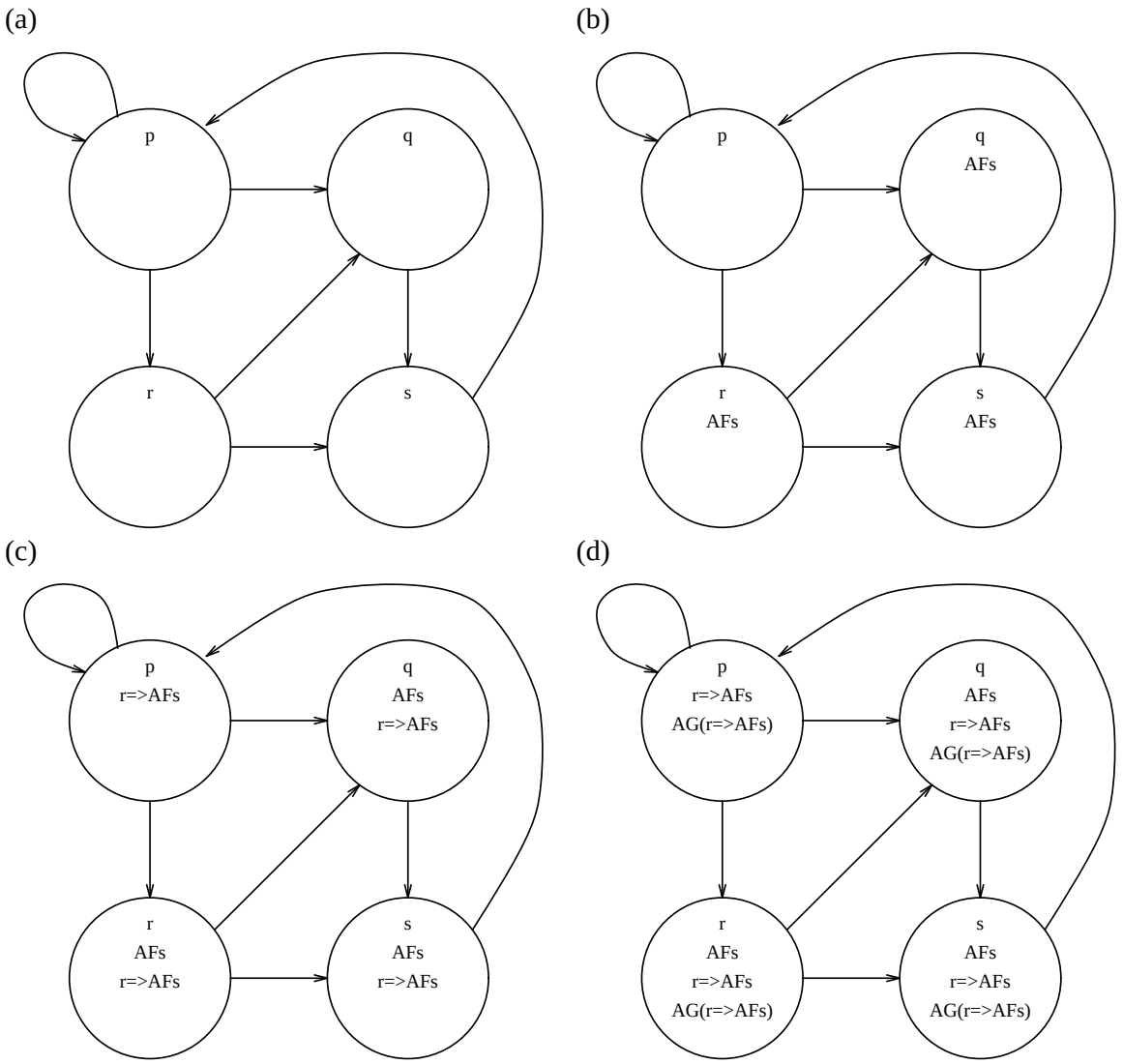


Figure 1.1: CTL Model Checking Example: Model checking proceeds by repeatedly traversing the graph for each subformula, labeling those states that satisfy the subformula. Here, we are checking the formula AG($r \Rightarrow$ AFs). Part (a) shows the original Kripke structure labeled with atomic propositions, part (b) labels those states that satisfy AFs, part (c) labels those states that satisfy $r \Rightarrow$ AFs, and part (d) labels those states that satisfy the entire formula. As we can see, all states satisfy the formula.

the smallest subformulas, giving time complexity linear in the size of the formula. Accordingly, we label those states that satisfy AFs (Figure 1.1(b)) by using a depth-first search of the graph: if a state is labeled s, then label it AFs, mark it as visited, and return; if a state is on our depth-first search stack, then there exists a cycle where s is never true, so don't label it AFs, do mark it as visited, and return; otherwise, AFs holds iff it holds at all of the successors of the state. Next, we can traverse the graph again, labeling all states that satisfy $r \Rightarrow$ AFs (Figure 1.1(c)). Finally, we perform another depth-first search, similar to the one for AF, to label all states that satisfy AG($r \Rightarrow$ AFs) (Figure 1.1(d)). Details about CTL model checking as well as model checking research for other logics can be found in Emerson's recent survey [28, Sec. 6.4 and 7.4].

In general, the complexity of model checking is a function of the size of the Kripke structure. CTL model checking, for example, requires time proportional to the product of the size of the structure and the length of the formula being checked. Since the size of the model is typically exponential in the number of variables in the system being verified (the "state explosion problem"), unless we find efficient means to represent the structure and sets of states, we can use model checking only on fairly small problems. Symbolic model checking with binary decision diagrams promises to be such an efficient means and has generated considerable excitement recently. Before we proceed, therefore, we need to know what a binary decision diagram is; then, we can examine how to use binary decision diagrams for symbolic model checking.

1.2.3 Binary Decision Diagrams

The binary decision diagram (BDD) is a data structure for representing Boolean functions. Bryant [6] introduced the BDD in its current popular incarnation, although the general ideas have been floating around for quite some time (e.g., as branching programs in the theoretical computer science literature). Conceptually, we can construct the BDD for a Boolean function as follows. First, build a decision tree for the desired function, obeying the restrictions that along any path from root to leaf, no variable may appear more than once, and that along every path from root to leaf, the variables always appear in the same order (Figure 1.2(a)). Next, repeatedly apply

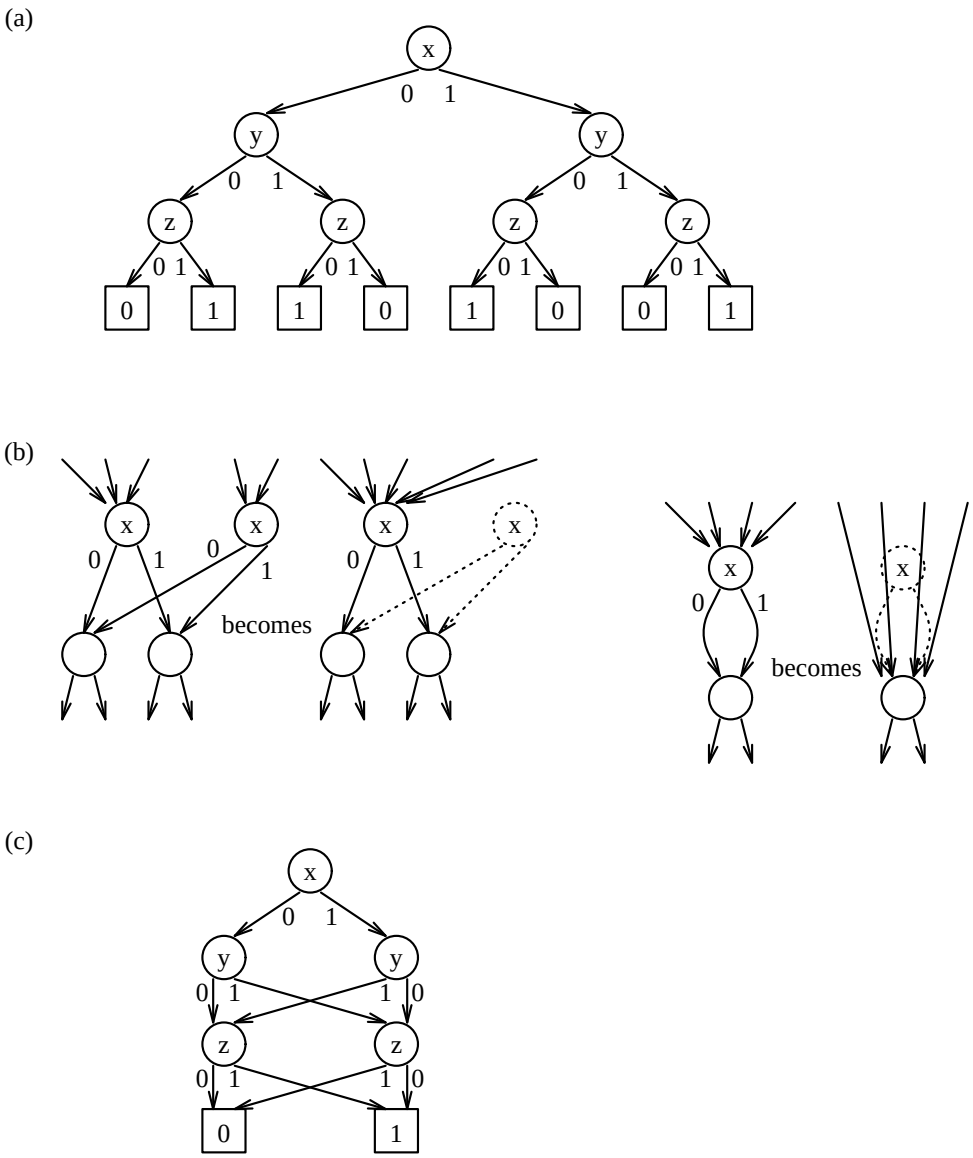


Figure 1.2: A Binary Decision Diagram (BDD): This example shows the BDD for the exclusive OR of x, y, and z. Conceptually, we can construct the BDD for the function by starting from a decision tree, shown in part (a). The decision tree is restricted so that (1) along any path from root to leaf, no variable appears more than once, and (2) along every path from root to leaf, the variables always appear in the same order. Next, we merge identical nodes and delete redundant nodes, as shown in part (b). The result is the BDD for the function, shown in part (c). In practice, BDDs are always maintained in the fully reduced form.

the following two reduction rules as much as possible: (1) Merge any duplicate (same label and same children) nodes, and (2) if both child pointers of a node point to the same child, delete the node because it is redundant (with the parents of the node now pointing directly to the child of the node) (Figure 1.2(b)). The resulting DAG is the BDD for the function (Figure 1.2(c)).[5] In practice, BDDs are generated and manipulated in the fully reduced form, without ever building the decision tree.

BDDs have several useful properties. First, although a simple counting argument shows that "most" Boolean functions require large BDD representations,[6] many common function have small BDDs. For example, generalizing the pattern in Figure 1.2(c), we can see that the BDD for the parity of n variables requires $2n - 1$ nodes, whereas parity requires an exponential-size representation using disjunctive normal form. In addition, BDDs are easy to manipulate. We can compute any binary Boolean operation on two functions represented as BDDs in time proportional to the product of the sizes of the BDDs. We can evaluate a function in time linear in the number of variables. We can existentially or universally quantify (Boolean) variables in a function in time quadratic in the size of the BDD. Finally, once we fix the order in which the variables appear, the BDD is a canonical representation for the Boolean function. Thus, function comparison, including special cases tautology and satisfiability, become trivially easy (constant time for efficient implementations [5]).

Choosing a good variable order is important. For example, suppose we wish to build a BDD for the function $(x_1 \oplus y_1) + (x_2 \oplus y_2) + (x_3 \oplus y_3)$. If we order the variables x_1, x_2, x_3, y_1, y_2, y_3, we get the large BDD shown in Figure 1.3(a). If instead we order the variables x_1, y_1, x_2, y_2, x_3, y_3, we get the smaller BDD shown in Figure 1.3(b).

[5] In general usage, "BDD" refers to the data structure described here, possibly with some modifications for efficiency [5] that don't concern us at the moment. This data structure has achieved widespread acceptance in VLSI, formal verification, and other fields. Subsequently, several researchers have generated a menagerie of BDD variants that relax, modify, or extend the rules governing the construction of BDDs, e.g., if-then-else DAG [54], MDD [83, 53], XBDD [50], snDD [49], EVBDD [57], FDD [55], SQBDD [70, 69], OPDD [74], ADD [1], etc., to name only a few. Accordingly, when there is risk of confusion in the literature, the BDD described here is dubbed an ROBDD, for **r**educed (because the reduction rules were applied) **o**rdered (because the variables are always in the same order from root to leaf) BDD. Since this thesis only concerns itself with the standard BDD, I will use the simpler and common term "BDD" rather than "ROBDD".

[6] There are 2^{2^n} Boolean functions of n variables. A representation scheme with size bounded by some polynomial $P(n)$ can represent at most $O(2^{P(n)})$ functions.

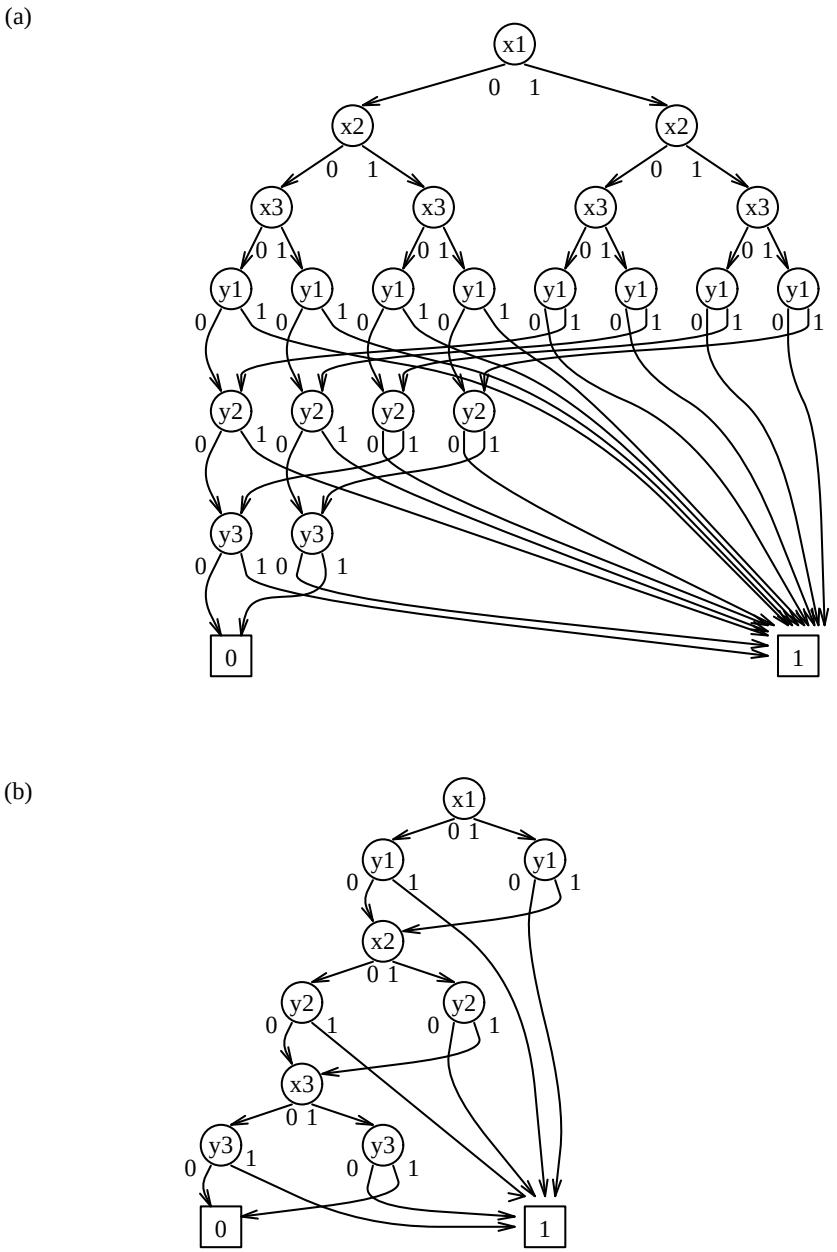


Figure 1.3: BDD Variable Order Matters: Here we consider the BDD for $(x_1 \oplus y_1) + (x_2 \oplus y_2) + (x_3 \oplus y_3)$ using two different variable orders. In general, the choice of variable order can make a difference between linear and exponential size.

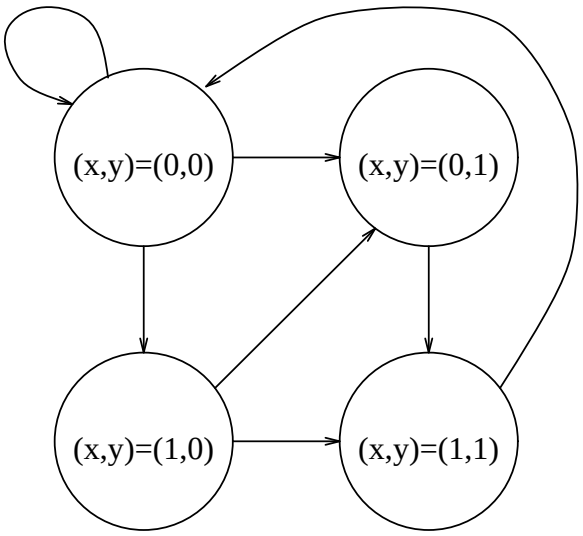
In general, the choice of variable order can make the difference between a linear size BDD and an exponential one. The problem of finding the best variable order for a given function is co-NP-complete [6]. Friedman and Supowit give an $O(n^2 3^n)$ algorithm for finding the optimum order [32]. Several heuristics for variable ordering exist for various applications such as combinational circuits [34, 63, 33] and sequential circuits [51]. The variable order can even be changed efficiently after the BDD has already been built [81].

In sum, BDDs are a practically efficient representation of Boolean functions. This is sufficient for us to examine how to use BDDs for model checking. Bryant [8] provides a detailed exposition on BDDs and surveys some applications and variations.

1.2.4 Symbolic Model Checking

The basic idea behind symbolic model checking is to use a more efficient "symbolic" representation for the Kripke structure being checked and for sets of states of the Kripke structure. Since the sizes of these representations is typically the limiting factor in applying model checking, an efficient representation can potentially allow much larger structures to be checked. The idea of using BDDs as this representation came upon several researchers roughly simultaneously [22, 21, 3, 10, 75, 85]. Among these early works, Burch *et al.*'s [10] is the most comprehensive treatment of symbolic model checking with BDDs and can be consulted for details. Here, I will briefly sketch symbolic model checking for CTL.

The first step is to label each state of the Kripke structure with a unique binary vector of length n, choosing n so that 2^n is at least the number of states in the structure. Associate each position in the binary vectors with a Boolean variable. For example, Figure 1.4 shows the same structure from the model checking example we saw previously (Figure 1.1), but now each state has a unique two bit label. We will associate the first bit with the Boolean variable x and the second bit with the Boolean variable y. In practice, we omit this step and simply use the state variables of the system we are verifying as the unique binary vector identifying each state. (For hardware, use the logic levels of the nets or latches; for software, use the program state.)



p	$\neg x \neg y$
q	$\neg x y$
r	$x \neg y$
s	xy
$\mathrm{AF}s$	$x + y$
$r \Rightarrow \mathrm{AF}s$	1
$\mathrm{AG}(r \Rightarrow \mathrm{AF}s)$	1

Figure 1.4: Symbolic Model Checking Example: This is the same structure as in Figure 1.1. Each state now has a unique two bit label (x, y), so we can describe any set of states by a Boolean expression as shown. Symbolic model checking works by computing directly with the Boolean expressions (represented as BDDs), rather than with an explicit representation of the Kripke structure.

With this labeling, any expression on the Boolean variables represents a set of states of the structure. For example, the expression $\neg x \neg y$ represents the single state in the upper left of the structure, the expression $x + y$ represents the set of all states except the upper left corner, and the expression 1 represents the set of all states. To perform model checking, instead of labeling states with subformulas as we did before, we associate each subformula with a BDD that represents the Boolean expression for the set of states at which the subformula holds. Thus, for example, we associate the atomic proposition s with the BDD for the Boolean expression xy.

We also need to represent the Kripke structure itself symbolically. The easiest way is to build the transition relation $\delta(q, q')$ that is true iff there is a transition from state q to state q'. Note that in the preceding paragraph, we represent a set of states by a Boolean function on a single state that indicates whether or not the state is in the set. In contrast, here we need to represent a transition relation, which is a Boolean function on a **pair** of states that indicates whether or not the specified transition exists. To use BDDs to represent transition relations, therefore, we must introduce

for each BDD variable x, another BDD variable x' that denotes the value of x in the next state. Returning to the example in Figure 1.4, we see that $(0,0)$ can transition to itself or state $(0,1)$ or state $(1,0)$, yielding $\neg x \neg y \Rightarrow \neg x' + \neg y'$. Similarly, state $(0,1)$ must transition to state $(1,1)$, yielding $\neg x y \Rightarrow x'y'$; state $(1,0)$ can transition to either states $(0,1)$ or $(1,1)$, yielding $x \neg y \Rightarrow y'$; and state $(1,1)$ must transition to state $(0,0)$, yielding $xy \Rightarrow \neg x' \neg y'$. Putting this all together gives us the transition relation

$$(\neg x \neg y \Rightarrow \neg x' + \neg y') \cdot (\neg x y \Rightarrow x'y') \cdot (x \neg y \Rightarrow y') \cdot (xy \Rightarrow \neg x' \neg y'),$$

which simplifies to $\neg x \neg y \neg x' + \neg x \neg y \neg y' + \neg x y x' y' + x y \neg x' \neg y' + x \neg y y'$.

The key to symbolic model checking is to perform all calculations directly using these Boolean expressions, rather than using the Kripke structure explicitly. For example, if we want to complement or intersect sets of states, we can just negate or AND the corresponding BDDs. More interesting are the modal operators. For example, given BDDs for the transition relation δ and the set of states that satisfy some formula p, we can compute the BDD for the set of states that satisfy AXp as $\lambda q \forall q' [\delta(q, q') \Rightarrow p(q')]$.[7] Using AX, we can express AFp as the least fixed point of a predicate transformer:

$$\mathrm{AF}p = \mathrm{lfp} Z.p \vee \mathrm{AX}Z,$$

which can be computed iteratively by starting Z at False. Similarly, AGp is the greatest fixed point of a predicate transformer:

$$\mathrm{AG}p = \mathrm{gfp} Z.p \wedge \mathrm{AX}Z,$$

which can be computed iteratively by starting Z at True. Convergence of both

[7] Please forgive the abuse of notation. Recall that δ and p are BDDs and are therefore propositional. The pseudo-first-order notation here simply indicates whether the BDD contains the primed or unprimed variables. Accordingly, the precise sequence of BDD operations to implement this expression is (1) change the BDD for p from unprimed to primed variables, (2) from the BDDs for δ and (primed) p, compute the BDD for the implication, and (3) universally quantify the primed variables from the resulting BDD. Changing the variables in a BDD between primed and unprimed is straightforward as long as the variable order keeps each primed-unprimed variable pair together.

iterations is guaranteed because the transformation is monotonic on a finite lattice.

Returning again to Figure 1.4, we can now work the example using symbolic model checking. The atomic proposition s has associated Boolean formula xy. To compute the Boolean formula for the CTL subformula AFs, we use the fixed point formula given above:

$$
\begin{aligned}
Z_0 &= 0 \\
Z_1 &= xy + \mathrm{AX}Z_0 \\
&= xy + \forall x'y'[(\neg x\neg y\neg x' + \neg x\neg y\neg y' + \neg xyx'y' + xy\neg x'\neg y' + x\neg yy') \Rightarrow 0] \\
&= xy + \\
&\quad (\neg x\neg y + xy \Rightarrow 0) \cdot \\
&\quad (\neg x\neg y + x\neg y \Rightarrow 0) \cdot \\
&\quad (\neg x\neg y \Rightarrow 0) \cdot \\
&\quad (\neg xy + x\neg y \Rightarrow 0) \\
&= xy + (x + y)(\neg x + \neg y)(\neg x + y)(x + \neg y) \\
&= xy \\
Z_2 &= xy + \mathrm{AX}Z_1 \\
&= xy + \forall x'y'[(\neg x\neg y\neg x' + \neg x\neg y\neg y' + \neg xyx'y' + xy\neg x'\neg y' + x\neg yy') \Rightarrow x'y'] \\
&= xy + \\
&\quad (\neg x\neg y + xy \Rightarrow 0) \cdot \\
&\quad (\neg x\neg y + x\neg y \Rightarrow 0) \cdot \\
&\quad (\neg x\neg y \Rightarrow 0) \cdot \\
&\quad (\neg xy + x\neg y \Rightarrow 1) \\
&= xy + (x + y)(\neg x + \neg y)(\neg x + y) \\
&= y \\
Z_3 &= xy + \mathrm{AX}Z_2 \\
&= xy + \forall x'y'[(\neg x\neg y\neg x' + \neg x\neg y\neg y' + \neg xyx'y' + xy\neg x'\neg y' + x\neg yy') \Rightarrow y'] \\
&= xy +
\end{aligned}
$$

$$
\begin{aligned}
& (\neg x \neg y + xy \Rightarrow 0) \cdot \\
& (\neg x \neg y + x \neg y \Rightarrow 1) \cdot \\
& (\neg x \neg y \Rightarrow 0) \cdot \\
& (\neg xy + x \neg y \Rightarrow 1) \\
= \;& xy + (x + y)(\neg x + \neg y) \\
= \;& x + y \\
Z_4 = \;& xy + \mathrm{AX} Z_3 \\
= \;& xy + \forall x' y' [(\neg x \neg y \neg x' + \neg x \neg y \neg y' + \neg xy x' y' + xy \neg x' \neg y' + x \neg y y') \Rightarrow x' + y'] \\
= \;& xy + \\
& (\neg x \neg y + xy \Rightarrow 0) \cdot \\
& (\neg x \neg y + x \neg y \Rightarrow 1) \cdot \\
& (\neg x \neg y \Rightarrow 1) \cdot \\
& (\neg xy + x \neg y \Rightarrow 1) \\
= \;& xy + (x + y)(\neg x + \neg y) \\
= \;& x + y,
\end{aligned}
$$

yielding $x + y$ as the Boolean formula for AFs. Atomic proposition r has Boolean formula $x \neg y$, so CTL subformula $r \Rightarrow$ AFs has Boolean formula $x \neg y \Rightarrow x + y$, which simplifies to 1. Computing the Boolean formula for the entire CTL formula AG($r \Rightarrow$ AFs) requires another fixed point computation, which converges immediately to 1. Therefore, we know that all states in the structure satisfy the formula.

Note that all calculations described in this section can be done efficiently, provided the BDDs are reasonably small. Whether the BDDs are indeed small in practice remains to be seen.

1.3 What This Thesis Is About

The research in this thesis falls under the general umbrella of symbolic model checking using BDDs. The emphasis of the work, however, is very specific. While most

current verification work with BDDs has focused on gate and switch-level designs, my emphasis is on the very highest levels of design, for example, checking communication and consistency-maintenance protocols in a very large system. As stated earlier, the goal is to provide a powerful debugging tool to catch conceptual errors early in the design cycle, when they are easier and cheaper to correct. Furthermore, my emphasis is explicitly practical and problem-driven, continually attempting interesting real problems in order to find the simplest tools that are still useful for a real user. Accordingly, I have restricted myself to a simple, very limited verification paradigm that has still proven useful for debugging a wide range of real, high-level designs [27] and instead concentrated on providing the techniques needed to bridge the gap between what is theoretically possible and what is practically feasible. In short, I have chosen to take something theoretically simple and try to make it useful on complex real problems, rather than taking something theoretically complex and apply it to simple problems.

More concretely, I consider verifying only that every reachable state of a system satisfies a user-specified propositional logic formula (equivalent to model-checking only CTL formulas of the form AGp where p contains no modal operators). Formally, I model the system being verified as a single non-deterministic finite-state machine. Non-determinism is important for high-level verification both to model non-determinism in the environment and also to abstract away implementation details, allowing us to postpone making low-level decisions until we've finished high-level verification. Let the machine have state space Q, transition relation $\delta : Q \times Q \rightarrow \{0, 1\}$, and a set of start states $S \subseteq Q$. The verification task is, given the set of "good" states $G \subseteq Q$ that satisfies the property being verified, to determine if there exists a path starting from a state in S and leading to a state not in G, and, if such a path exists, to output it as a counterexample to the property being verified.

Before proceeding, we need a bit of notation:

Definition 1.1 (Image Operators) *Given a set $Z \subseteq Q$, define the following operators:*

$$Image(\delta, Z) \quad = \quad \{v | \exists u[u \in Z \wedge \delta(u, v)]\}$$

$$\begin{aligned} \mathit{PreImage}(\delta, Z) &= \{u | \exists v[v \in Z \wedge \delta(u, v)]\} \\ \mathit{BackImage}(\delta, Z) &= \{u | \forall v[\delta(u, v) \Rightarrow v \in Z]\}. \end{aligned}$$

Intuitively, Image gives the set of states that can be reached in one transition from a state in Z, PreImage gives the set of states that in one transition *can* reach a state in Z, and BackImage gives the set of states that in one transition *must* end up in Z. These operators are by no means new: Image is essentially Floyd's strongest verifiable consequent [31], and BackImage is essentially Dijkstra's weakest precondition [26]. In CTL (with δ implicit in the Kripke structure), PreImage(δ, Z) corresponds to EX$[Z]$, and BackImage(δ, Z) corresponds to AX$[Z]$. Using the unified notation of dynamic logic [79], the expression Image(δ, Z) is precisely $\langle \delta^- \rangle Z$, the expression PreImage(δ, Z) is precisely $\langle \delta \rangle Z$, and the expression BackImage(δ, Z) is precisely $[\delta]Z$.

Using these image operators, it's easy to explain the standard approaches to our verification problem. One standard algorithm I call "forward traversal." The intuition is that we iteratively compute the set R_i of states that can be reached in i or fewer transitions from the start states. Mathematically, we initialize $R_0 = S$, and compute $R_{i+1} = R_0 \vee \text{Image}(\delta, R_i)$.[8] If R_i ever goes outside the set of good states ($R_i \not\subseteq G$), then we have a violation, and it's easy to produce a counterexample trace. Otherwise, the sequence will eventually converge to the set of reachable states, meaning that the verification succeeds. (The predicate transformation is monotonic on a finite Boolean lattice, so convergence is guaranteed.) Details of this approach are available elsewhere (e.g., [22, 11, 16, 85, 9]). The other standard algorithm I call "backward traversal." The intuition here is that we iteratively compute the set G_i of states such that all paths of length i or less starting in G_i must remain within the set of good states G. Mathematically, we initialize $G_0 = G$, and compute $G_{i+1} = G_0 \wedge \text{BackImage}(\delta, G_i)$.

[8] An alternative formulation that is perhaps more intuitive is $R_{i+1} = R_i \vee \text{Image}(\delta, R_i)$. Both formulas compute the same R_i for all i. An advantage of the formula given in the text is that R_0 is often a simple property with smaller BDD than R_i. An advantage of the alternative formulation is that we can actually compute the image on any set S such that $R_i - R_{i-1} \subseteq S \subseteq R_i$, giving $R_{i+1} = R_i \vee \text{Image}(\delta, S)$. This formula still generates the same R_i for all i, but the image computation can be faster if we can find at each iteration an S with BDD representation smaller than R_i [22, 21]. Note that combining these tricks into $R_{i+1} = R_0 \vee \text{Image}(\delta, S)$ is **not** guaranteed to produce the same sequence, and indeed may produce a sequence that is non-monotonic.

If we reach a point where G_i does not contain all of the start states ($S \not\subseteq G_i$), then there exists a sequence of i transitions from a start state to a violating state. Otherwise, the sequence will converge, meaning the verification succeeds. (Again, convergence follows immediately from monotonicity and finiteness.) Details for this approach are also available from several sources (e.g., [10, 75, 30]). Incidentally, these two approaches are actually duals of each other, based on the dynamic logic duality between forward and backward execution of a program [79].

At this point, I appear to be done: I've stated a simple verification problem and have given two standard algorithms to solve it. We just build BDDs for δ and the R_is or G_is, compute the image operators using Definition 1.1, and crank through the iterations described in the preceding paragraph. However, although indulging my Inner Theoretician by declaring victory and moving on to more mathematically beautiful verification problems is tempting and certainly valuable, I've explicitly set out to provide a practical solution to practical problems, so I need to indulge my Inner Practitioner first. As often happens, the difference between theory and practice emerges in the practice. Despite the extreme simplicity of our verification problem, several obstacles prevent the practical usefulness of direct implementations of the standard approaches on an enormous range of real problems. In particular, describing a complex system directly with BDDs is difficult and error-prone, the BDD for δ is frequently too large to build, and the BDDs for the R_is and G_is are frequently too large to build as well. This thesis addresses these problems.

Chapter 2

Translation into BDDs

Chapter Overview[1]

An enormous semantic gap lies between the low-level Boolean logic supported by BDDs and the high-level descriptions of systems we wish to verify. The first part of this chapter motivates the need to support high-level language constructs for these high-level descriptions and identifies an empirically useful set of constructs: data structures consisting of scalars, arrays, and records; expressions built from boolean and arithmetic operators; and control flow based on conventional assignment statements, sequence, if-then-else, and non-deterministic choice. The remainder of the chapter describes how to translate these language constructs automatically into BDDs.

2.1 High-Level Language Constructs

The first step in BDD-based verification is to use Boolean formulas to describe the system being verified. For high-level verification, writing and maintaining such low-level descriptions can be a major obstacle. Allow me to illustrate with a personal

[1]This chapter is based on material first published in the *Fourth International Workshop on Computer-Aided Verification*, 1992 [44].

anecdote. The seed of the Stanford Murφ group began in the spring of 1990. Inspired by recent stories of success with BDD-based model checking [10, 67], we set out to use BDD-based verification to help debug the directory-based cache coherence protocol of the Stanford DASH multiprocessor [59], then well under development. The results were humbling. The task of modeling the high-level description of the system in Boolean logic proved to be enormously time-consuming and error-prone. Furthermore, subtle modeling errors were hard to detect, as a hard-to-spot typographical error in the Boolean description can silently delete transitions from the transition relation. Worse, upon finding an error in our model, the task of revising a description written in Boolean logic was difficult and even more error-prone. The killing blow was that if we wanted to scale the description to have more or less memory or processors, we essentially needed to rewrite the description from scratch. The end result was that we were unable to produce useful verification results in a timely manner. In retrospect, McMillan and Schwalbe's success story that had inspired us — verifying a hierarchical snooping cache coherence protocol [67] — should also have forewarned us, as they too complained that the low-level Boolean descriptions were very cumbersome for high-level verification. Clearly, high-level verification with BDDs required better tool support.

We learned two main language-design lessons from this experience. First, descriptions must be easy to revise and modify. This requirement is true for programming in general but is particularly compelling for high-level prototyping, debugging, and verifying, as we need and expect to iterate the modify-check-debug cycle very quickly and very often. A particularly important revision for state-exploration verification is scaling. Often, we cannot verify the system in its full complexity, so being able to quickly scale the sizes of various parts of the description (e.g., number of processors, number of addresses, size of buffers, etc.) is crucial to be able to find bugs [27]. The other language-design lesson is that the description language must be semantically close to the way the user normally writes descriptions. Thus, the verification system must support the data structures, the imperative semantics, and the control flow typical of current programming languages. A particularly unpleasant mismatch between imperative semantics and logic is an instance

of the Frame Problem in AI [66]: in imperative semantics, when modifying a variable, any other variable that isn't mentioned doesn't change; in logic, any variable that isn't mentioned is completely unconstrained. While it's straightforward but tedious to translate an imperative assignment like "`x:=1`" into the next-state relation[2] $(x' = 1) \wedge (y' = y) \wedge (z' = z) \wedge \cdots$(a clause for every variable)$\cdots$, it's arduous, error-prone, and still tedious to translate something (taken from an industrial cache coherence protocol) like:

```
If (i < Homes[h].Dir[a].Shared_Count) &
    (Homes[h].Dir[a].Entries[i] = n)
Then
  -- overwrite this entry with last entry.
  Homes[h].Dir[a].Entries[i] :=
    Homes[h].Dir[a].Entries[Homes[h].Dir[a].Shared_Count-1];
  -- clear last entry
  Homes[h].Dir[a].Entries[Homes[h].Dir[a].Shared_Count-1] := 0;
  Homes[h].Dir[a].Shared_Count := Homes[h].Dir[a].Shared_Count-1;
Endif;
```

The user should not be forced to perform such a translation by hand.

The objective, therefore, is to support a high-level description language, counterbalanced by the need to keep the description finite-state and comfortably within the bounds of our verification paradigm. For example, to allow easy scalability and high-level descriptions, the language must support scalar-valued variables (integer and enumerated types) as well as complex data structures built from records and arrays. Pointers, however, would allow infinite-state descriptions and must therefore be disallowed. Once the language permits scalar-valued variables, it obviously must provide adequate means to manipulate them, such as built-in operators for addition, subtraction, comparison, etc. Providing a built-in imperative assignment frees the user from the onerous and error-prone translation described in the preceding paragraph. For control flow, the imperative assignment goes hand-in-hand with sequential control

[2] Recall from Chapter 1 that a primed variable denotes the value of the variable in the next state.

flow and if-then-else, which are basic control structures that the user expects to have. While-loops, however, would require a nested fixed point computation, which is too expensive to support. A non-deterministic choice operator provides a convenient high-level means to introduce non-determinism and implement guarded command sets [26]. Finally, some sort of subroutine mechanism is needed to provide modularity, but we cannot permit recursion, which could violate the finite-state restriction.

We built the non-BDD-based Murφ verifier around these language features [27], embedded into an iterated, guarded command framework inspired by the UNITY language [13]. That these language features are sufficiently high-level and easy-to-use is clear from the consistent praise Murφ has drawn on ease of use and description language convenience (e.g., [88, 86]). That these language features can be translated into BDDs is the subject of the following section.

2.2 Translation

As noted in Chapter 1, symbolic model checking requires that the system be modeled by a transition relation on the current and next state values of the state variables. The preceding section listed the set of high-level language features we wish to support. This section shows how to translate these features automatically into transition relations in Boolean logic, suitable for implementation with BDDs. I have implemented the automatic translation in the Ever verification system, described in more detail in Appendix A.

I assume the preceding section's desirable language features are combined in the typical manner into an Algol-style language. The translation process is syntax-directed, traversing the high-level description recursively, building the Boolean formulas for the subparts of a given part of the description and then combining the formulas for the subparts into the Boolean formula(s) for the part. I will describe the routines that perform this translation starting from the bottom-level translation routines and proceeding to the top, which converts the entire high-level description into a transition relation in Boolean logic.

The process of translation requires manipulating different representations of the

same conceptual objects, making explaining the process potentially confusing; introducing some concepts and notation eases the exposition. The translation routines generate three main types of results: Boolean conditions, scalar quantities, and transition relations that correspond to subparts of the high-level description. Both Boolean conditions and transition relations are represented by single Boolean functions, although keeping these two concepts distinct is helpful. In particular, the Boolean formula for a condition refers only to variables for the current state or only to variables for the next state, but not to both, whereas the Boolean formula for a transition relation typically refers to variables for both the current and the next state of the system. Scalar quantities are represented by vectors of Boolean formulas, which give the binary encoding of the scalar value. For a given part P of the high-level description, let BFUN(P), BVEC(P), or TR(P), whichever is appropriate, denote the result of translating P into a Boolean formula, a vector of Boolean formulas, or a transition relation, respectively. Unless otherwise specified, Boolean operations performed on vectors of Boolean formulas should be interpreted as applying component-wise. Subscripts specify particular formulas within a vector of Boolean formulas, with 0 referring to the low-order bit. For example, $\text{BVEC}(13)_1$ is False, and $\text{BVEC}(x)_0$ refers to the low-order bit of x. By default, Boolean formulas and vectors of Boolean formulas should be assumed to refer to the current state variables; a tick mark applied to any expression indicates that the expression refers to the next state variables instead. For example, BFUN($x < 42$) is a Boolean formula that specifies that the current value of x is less than 42, whereas $\text{BFUN}(x < 42)'$ is a Boolean formula that specifies that the next value of x is less than 42. Note that the expression BFUN($x' < 42$) is meaningful only if "$x' < 42$" is meaningful in the high-level description language; if the description language uses "x'" to denote the next value of x, then $\text{BFUN}(x' < 42) = \text{BFUN}(x < 42)'$.

2.2.1 Constants and Variables

For an integer constant c, the translation BVEC(c) is simply the binary representation of c. Enumerated types can be handled by associating each enumeration constant with an integer, exactly as a normal compiler does. Note that other encodings of

finite domains into Boolean variables are possible [53], but the binary encoding is advantageous in supporting a high-level description language because it simplifies automatic translation of arithmetic and logical expressions, as we shall see shortly.

Variable declarations are handled exactly as in a normal compiler, except that rather than allocating bits of memory for the high-level variables, the translator allocates entries in an array of Boolean variables. Call this array V, with the notation V_i indicating the ith Boolean variable and the notation $V_i[s]$ indicating the s Boolean variables starting with the ith one. I will assume a Pascal or C-like type system, in which the base types are integers and enumerations, and additional types can be constructed as records whose fields are of previously defined types and as arrays whose elements are of a previously defined type. For any type, a straightforward recursive computation tells how many Boolean variables are needed to represent it: the size of a scalar is the number of bits used to represent it, the size of a record is the sum of the sizes of its fields, and the size of an array is the product of the number of elements in the array and the size of an array element. A related quantity is the offset from the start of a record or an array at which a specific record field or array element starts. The offset is easily computed as the sum of the sizes of the record fields or array elements that precede the specified one. I will use the notation SIZEOF(x) to denote the size of a variable or type x and the notation OFFSET(x) to denote the offset to the start of the field named x or the xth element in an array (using context to disambiguate which I mean). When a new variable x is declared, its size and offset (We can treat all variables as fields in a global record that starts at V_0.) are stored in the symbol table, and the Boolean variables $V_{\mathrm{OFFSET}(x)}[\mathrm{SIZEOF}(x)]$ are allocated for variable x.

Recall from Section 1.2.3 the importance for BDDs of choosing a good variable ordering. Ideally, such a detail should not belong in the high-level description; the translator should choose a good order automatically. Unfortunately, current variable ordering heuristics for high-level descriptions are not as good as what a human can do, so the user needs some means to provide variable ordering hints to the translator. For example, for an array, in some instances ordering all the BDD variables for one array element before all of the BDD variables of the next array element is the best

choice; in other instances, interleaving the corresponding bits (most significant bits first, followed by the next most significant bits, etc.) is better [83, 53]. The user can use intuition or a result like Jeong *et al.*'s Non-Interleaving Lemma [51][3] to decide which order to use and must be able to tell the translator to do so.

When an expression refers to a variable (possibly with record field accesses and array indexing), the translation process is again very similar to what a normal compiler does, except that rather than generate instructions that compute the correct address offset, the translation into Boolean logic generates a vector of case expressions that select the correct Boolean variables. For a simple variable reference, the translator looks up in the symbol table the offset to the correct BDD variables much as a compiler would generate the offset to the start of the correct block of memory. The Boolean variables starting at that offset form the correct vector of Boolean formulas. For a record field access, start with the base variable as before, add the field offset, and proceed as in the case of the simple variable reference. Array indexing is the most complex. If the index were a constant, we could proceed as for records. The index, however, can be an arbitrary scalar-valued expression. For example, an expression like $a[x+4]$ should be translated into a vector of Boolean formulas that, when restricted to having $x = 0$, are equivalent to the Boolean formulas for $a[4]$; when restricted to having $x = 1$, are equivalent to $a[5]$; and so forth. Therefore, the formulas for an array access must perform a case analysis for each possible value of the array index.

More formally, the translation of a variable reference can be expressed by a recursive computation. Define a *modifier* to be either a field name or an array-indexing expression. Since we can consider a variable to be just a field in a global record structure, any variable reference is just a list of modifiers. Given an arbitrary variable reference ρ, the translation $\mathrm{BVEC}(\rho)$ is given by the function SELECTBITS:

$$\mathrm{BVEC}(\rho) \stackrel{\mathrm{def}}{=} \mathrm{SELECTBITS}(\rho, 0, \mathrm{SIZEOF}(\rho)).$$

[3] In an unfortunate clash of terminology, what I and Srinivasan *et al.* [83] call an interleaved variable order, because the bits comprising the scalars are interleaved, creates what Jeong *et al.* call a non-interleaved order, because the Boolean relations for the individual bit-slices have disjoint supports and are grouped together.

In defining SELECTBITS, recall that operations on vectors of Boolean formulas should be assumed to be applied component-wise. Two exceptions are $\stackrel{*}{=}$, which denotes checking equality of two vectors of Boolean formulas and returns a single Boolean formula, and $\stackrel{*}{\Rightarrow}$, which denotes a single Boolean formula implying a vector of Boolean formulas and returns the vector in which the implication was applied to each component. For example, BVEC(12) = BVEC(7) yields the result (False, True, False, False), whereas BVEC(12) $\stackrel{*}{=}$ BVEC(7) yields the result False. With this notation, we define SELECTBITS recursively as follow:

$$\text{SELECTBITS}(\epsilon, \text{offset}, \text{size}) \stackrel{\text{def}}{=} V_{\text{offset}}[\text{size}]$$

$$\text{SELECTBITS}(\textit{field_name} :: \sigma, \text{offset}, \text{size}) \stackrel{\text{def}}{=}$$
$$\text{SELECTBITS}(\sigma, \text{offset} + \text{OFFSET}(\textit{field_name}), \text{size})$$

$$\text{SELECTBITS}(\textit{index_expr} :: \sigma, \text{offset}, \text{size}) \stackrel{\text{def}}{=}$$
$$\bigwedge_{i=l}^{u} \left[\begin{array}{c} \text{BVEC}(\textit{index_expr}) \stackrel{*}{=} \text{BVEC}(i) \\ \stackrel{*}{\Rightarrow} \\ \text{SELECTBITS}(\sigma, \text{offset} + \text{OFFSET}(i), \text{size}) \end{array} \right],$$

where σ is a (possibly empty) list of modifiers, ϵ is the empty list, :: is the list `cons` operator, and l and u are the lower and upper bounds of the array. The actual implementation also requires type information for the variables to be propagated through the recursive calls in the obvious way; I have omitted this detail for brevity. We will see how to compute BVEC(*index_expr*) in a moment. The generated formulas are essentially banks of multiplexors whose select lines are driven by the array-indexing expressions and whose inputs are the BDD variables. The formula for array indexing can be considered a scalar-valued generalization of work by Beatty *et al.* [2].

2.2.2 Arithmetic and Logical Expressions

Compared to the translation of variable references, the translation of arithmetic and logical expressions is easy. I assume that the translator can parse expressions in the

high-level language and perform type checking using standard compiler techniques. Thus, the task of translating arbitrary expressions can be broken down into defining the translation of each supported operator, assuming its operands have been properly translated already. The translator applies these rules recursively on the parse tree, first translating the subexpressions that are the arguments of an operator, and then using the specific translation rules for that operator. The base cases of the recursion are the translations for variable references and constants.

Keep in mind that the translation process does not compute the result of the operation; it generates Boolean functions that compute the result of the operation for all input values. For example, if the translation BVEC(*expr*) is the vector of Boolean formulas (f_3, f_2, f_1, f_0), the translation of $expr - 4$ would be the vector of formulas $(f_3 = f_2, \neg f_2, f_1, f_0)$.

The use of the standard unsigned binary representation to encode integers makes defining the translation of common arithmetic operators easy.[4] Let's consider some representative examples: addition, less-than, and a conditional expression.

Suppose the high-level description contains the expression "*expr1* + *expr2*", where *expr1* and *expr2* are arbitrary scalar-valued expressions. As stated above, we first apply the translation recursively on the subexpressions. Let x be the translation BVEC(*expr1*), and let y be the translation BVEC(*expr2*). Compute the vector of Boolean formulas z by the following rules:

$$\begin{aligned} c_{-1} &\stackrel{\text{def}}{=} \text{False} \\ z_i &\stackrel{\text{def}}{=} x_i \oplus y_i \oplus c_{i-1} \\ c_i &\stackrel{\text{def}}{=} (x_i \wedge y_i) \vee (c_{i-1} \wedge (x_i \vee y_i)), \end{aligned}$$

with $\oplus$ denoting exclusive-OR. These rules implement addition by a ripple-carry adder. Note that since BDDs are canonical, using this simple adder structure gives

[4]Not surprisingly, we are actually encoding integers modulo 2^k, where k is the number of bits we've allocated for the result. Two's complement permits handling negative numbers with minimal hassle. If the high-level language makes extensive use of negative numbers, the translation should be modified slightly to handle signed comparison (versus the unsigned comparison given in the text) and sign extension issues, just as a normal compiler does.

the exact same final result z as using a more complicated adder. The translation BVEC(*expr1* + *expr2*) is just the vector z.

As a second example, suppose the high-level description contains the expression "*expr1* < *expr2*", where again *expr1* and *expr2* are arbitrary scalar-valued expressions. Note that in this example, the result should be a single Boolean formula, rather than a vector of Boolean formulas, so the translation is denoted by BFUN rather than BVEC. This type inference is a function of the high-level language, not the translation process, and can be handled by standard compiler techniques. As in the preceding example, let x be the translation BVEC(*expr1*), and let y be the translation BVEC(*expr2*). Compute a sequence of Boolean formulas:

$$\begin{aligned} z_0 &\stackrel{\text{def}}{=} \text{False} \\ z_i &\stackrel{\text{def}}{=} (\neg x_i \wedge y_i) \vee ((x_i = y_i) \wedge z_{i-1}). \end{aligned}$$

The translation BFUN(*expr1* < *expr2*) is just the Boolean formula z_i for i equal to the larger of SIZEOF(x) and SIZEOF(y). (If one expression is smaller than the other, pad the smaller with leading 0s for the computation of z. Small adjustments like this are needed for all operations.)

As a final example, consider a conditional expression: "*cond*?*expr1*:*expr2*" in C syntax. Let x be BVEC(*expr1*) and y be BVEC(*expr2*) as in the previous examples. Let c be BFUN(*cond*). Compute the vector of Boolean formulas z:

$$z_i \stackrel{\text{def}}{=} (c \wedge x_i) \vee (\neg c \wedge y_i),$$

padding the smaller of x and y as before. The translation BVEC(*cond*?*expr1*:*expr2*) is just z.

Translation rules for many other operators are similar.

2.2.3 Assignment

So far, we have only considered translating the bottom-level parts of a description: constants, variables, and expressions. Now, we have developed enough machinery

to translate our first construct that generates a transition relation: an assignment statement. The assignment statement, in turn, will be the basis for inductively constructing high-level descriptions.

To translate an assignment statement, we use a computation similar to the variable reference computation described above. Now, however, we must build a transition relation that specifies that every Boolean variable that isn't modified in the assignment keeps its current value. Doing so resolves the mismatch mentioned earlier between imperative and logical semantics. For records, generating this relation is straightforward. For each field not being accessed, we AND into the Boolean relation being generated the further requirement that the field not change. For the field that we do access, we equate a variable reference expression like those we generated previously with the right-hand side of the assignment. For arrays, we must again perform a case analysis.

Formally, given assignment statement "$var := expr$", let ρ be the list of modifiers that corresponds to the variable reference *var*. As with the computation for a variable reference, we define the translation $\mathrm{TR}(var := expr)$ by a recursive helper function:

$$\mathrm{TR}(var := expr) \stackrel{\text{def}}{=} \mathrm{ASSIGN}(\rho, 0, \mathrm{SIZEOF}(\rho), expr).$$

The recursive definition of ASSIGN is similar to that of SELECTBITS:

$$\mathrm{ASSIGN}(\epsilon, \mathrm{offset}, \mathrm{size}, \mathrm{expr}) \stackrel{\text{def}}{=} (V_{\mathrm{offset}}[\mathrm{size}]' \stackrel{*}{=} \mathrm{BVEC}(\mathrm{expr}))$$

$$\begin{aligned}\mathrm{ASSIGN}(&\mathit{field_name} :: \sigma, \mathrm{offset}, \mathrm{size}, \mathrm{expr}) \stackrel{\text{def}}{=} \\ &\mathrm{ASSIGN}(\sigma, \mathrm{offset} + \mathrm{OFFSET}(\mathit{field_name}), \mathrm{size}, \mathrm{expr}) \quad \wedge \\ &\bigwedge_{\substack{f \in \mathit{record_fields} \\ f \neq \mathit{field_name}}} \begin{pmatrix} V_{\mathrm{offset}+\mathrm{OFFSET}(f)}[\mathrm{SIZEOF}(f)]' \stackrel{*}{=} \\ V_{\mathrm{offset}+\mathrm{OFFSET}(f)}[\mathrm{SIZEOF}(f)] \end{pmatrix}\end{aligned}$$

$$\mathrm{ASSIGN}(\mathit{index_expr} :: \sigma, \mathrm{offset}, \mathrm{size}, \mathrm{expr}) \stackrel{\text{def}}{=}$$

$$\bigwedge_{i=l}^{u} \left[\begin{array}{l} \text{if} \quad \text{BVEC}(\mathit{index_expr}) \stackrel{*}{=} \text{BVEC}(i) \\ \text{then} \\ \quad \text{ASSIGN}(\sigma, \text{offset} + \text{OFFSET}(i), \text{size}, \text{expr}) \\ \text{else} \\ \quad \left(\begin{array}{c} V_{\text{offset}+\text{OFFSET}(i)}[\text{SIZEOF}(\mathit{element})]' \stackrel{*}{=} \\ V_{\text{offset}+\text{OFFSET}(i)}[\text{SIZEOF}(\mathit{element})] \end{array} \right) \end{array} \right],$$

where σ is a list of modifiers, ϵ is the empty list, :: is the list `cons` operator, and l and u are the lower and upper bounds of the array.

A non-deterministic assignment statement, in which a variable is assigned any value that satisfies some user-specified condition, can be implemented by essentially the same computation, except the base case of the recursion is the condition on the variable rather than the expression ($V_{\text{offset}}[\text{size}]' \stackrel{*}{=} \text{BVEC(expr)}$) above. For example, the translation of an assignment statement that says, "Let x have any value less than 42." would proceed exactly as above for a normal deterministic assignment to x, except the base case of the recursion would be $\text{BFUN}(x < 42)'$.

2.2.4 Control Flow

With the assignment statement as the basis, we can define compound statements for the high-level language features we wish to support (sequence, if-then-else, and non-deterministic choice) inductively in the usual manner. If $s_1, \ldots, s_n$ are statements, then the sequence of statements "$s_1; \ldots; s_n$" (executed sequentially in order) is also a statement. If s_1 and s_2 are statements and c is a conditional expression, then "if c then s_1 else s_2 endif" is also a statement. If $s_1, \ldots, s_n$ are statements, then the non-deterministic choice of one of them, denoted here by the expression "$s_1 | \ldots | s_n$", is also a statement. The description of the entire transition relation, therefore, is just a single statement, albeit a complex one.

Given the inductive definition of statements, the translation of a statement into a transition relation is naturally recursive. The base case is the assignment statement. The other rules are as follows:

Sequence: The result is just the composition of the transition relations for the statements in the list:

$$\mathrm{TR}(s_1; \ldots; s_n) \stackrel{\mathrm{def}}{=} \mathrm{TR}(s_1) \circ \cdots \circ \mathrm{TR}(s_n),$$

where $\circ$ denotes relational composition: $\delta_1 \circ \delta_2 \stackrel{\mathrm{def}}{=} \lambda q, q'.\exists q''[\delta_1(q, q'') \wedge \delta_2(q'', q')]$.[5] Clarke *et al.* [19], as part of a larger work on building approximate abstract models of programs, have proposed essentially this rule combined with a special case (no complex data structures) of the deterministic assignment rule from Section 2.2.3.

If-Then-Else: The resulting transition relation just uses the condition to select which branch applies:

$$\begin{aligned}\mathrm{TR}(\text{if } c \text{ then } s_1 \text{ else } s_2 \text{ endif}) \stackrel{\mathrm{def}}{=} \; & (\mathrm{BFUN}(c) \Rightarrow \mathrm{TR}(s_1)) \wedge \\ & (\neg\mathrm{BFUN}(c) \Rightarrow \mathrm{TR}(s_2)).\end{aligned}$$

Non-Deterministic Choice: The result is just the OR of the transition relations for the statements in the list:

$$\mathrm{TR}(s_1 | \ldots | s_n) \stackrel{\mathrm{def}}{=} \mathrm{TR}(s_1) \vee \cdots \vee \mathrm{TR}(s_n).$$

These rules complete the description of the translation process. Subroutines can be handled by macro expansion since recursion is not allowed. We can now translate an empirically useful set of high-level language features automatically into BDDs, greatly simplifying this step for BDD-based verification.

[5] Recall from Chapter 1 that this pseudo-first-order notation simply indicates which variables the BDDs refer to. A straightforward implementation of relational composition introduces a double-primed copy of each BDD variable. Accordingly, the precise sequence of BDD operations is (1) change the primed variables to double-primed in the BDD for δ_1, (2) change the unprimed variables to double-primed in the BDD for δ_2, (3) AND the two BDDs, and (4) existentially quantify the double-primed variables from the resulting BDD.

Chapter 3

Efficient Image Computation

Chapter Overview[1]

Although automatic translation from a high-level language to a transition relation in Boolean logic as given in the preceding chapter is convenient, the transition relation frequently turns out to be too large to represent as a BDD. Fortunately, the verification algorithms do not actually need the transition relation as a BDD; rather, they need to compute the images of sets of states through the transition relation. This chapter shows how this computation can be done without building the BDD for the transition relation.

3.1 Background

The techniques from Chapter 2 are sufficient to convert a high-level description into a Boolean next-state relation represented as a BDD. If we return to the verification algorithms from Chapter 1, we find that the BDD for the transition relation makes it easy to compute the image operators, which in turn make it easy to solve my verification problem.

Unfortunately, the BDD for the transition relation frequently turns out to be too

[1]This chapter is based on material first published in the *Fourth International Workshop on Computer-Aided Verification*, 1992 [44].

large to build. Note that this is not a new problem resulting from inefficiencies in the translation routines I've given; transition relation BDD blow-up was noted in some of the earliest papers on verification with BDDs [21, 85, 9].

Fortunately, a closer examination of the BDD-based verification algorithms suggests that building the BDD for the transition relation might not be necessary. The advantages of the BDD representation are ease of manipulation, canonicity (which makes comparing functions easy), and frequently compact size. In this case, the BDD for the transition relation is blowing up, so the compact size advantage does not apply. The verification algorithms manipulate the transition relation only as part of the image computation, and transition relations are never checked for equality, obviating the need for canonicity. What we really need is to be able to compute the image operators applied to sets of states represented as BDDs; building the BDD for the transition relation is just a means unto the end of computing these image operations.[2]

The first technique to capitalize on this observation that we need not build a BDD for the entire transition relation is the use of Boolean functional vectors [21, 51, 30]. This approach represents the transition structure of the system as a function from the current state to the next state, using a separate BDD for each state bit to compute the value of that bit in the next state. In the domain of gate-level sequential digital circuits (for which this technique was developed), the BDDs correspond naturally to the combinational fan-in cones of the latches. If these BDDs are small, using Boolean functional vectors can be very efficient. Image computation with Boolean functional vectors is unintuitive, but empirically efficient; see the references above for details.

Another approach to avoid building the BDD for the entire transition relation is the idea of partitioned transition relations [9], which come in conjunctive and disjunctive varieties. For a conjunctive partitioned transition relation, the transition relation is represented, not by a single BDD, but by a set of BDDs whose conjunction forms the transition relation. Conjunctive partitionings arise naturally when modeling a system as the synchronous composition of subsystems: the transition relation

[2] Unfortunately, the fact that the BDD for the transition relation is too large to build precludes a number of theoretical improvements to the verification algorithm, such as performing don't-care simplification on the BDD for the transition relation [12] or computing transitive closures via iterative squaring [10] or recursive block-matrix decompositions [65].

for each subsystem is a separate BDD, and the transition relation of the whole system is just the conjunction of the transition relations of the subsystems. Image computation with conjunctive partitioned transition relations can be expensive, degenerating in the worst case to the non-partitioned case. Note that the Boolean functional vector representation is easily converted into a conjunctive partitioned transition relation [85]. For a disjunctive partitioned transition relation, the disjunction of a set of BDDs forms the transition relation. Disjunctive partitionings arise naturally when modeling a system with non-deterministic choice or asynchronous interleaving. Image computation with disjunctive partitioned transition relations is easy, since conjunction and existential quantification distribute over disjunction. The efficient image computation presented in the next section can be viewed as a generalization of the disjunctive partitioned transition relation.

A completely different view motivating the image computation presented in the next section is as a combination of the temporal logic framework for verifying reactive systems with the program logic framework for verifying transformational programs. The verification problem in this thesis falls squarely in the camp of temporal logic and reactive systems. The temporal logic paradigm, however, views the transition relation as a black box. Symbolic model checking proceeds by computing images of sets of states through the transition relation. Now, recall from Definition 1.1 (p. 18) that the Image operator is essentially the strongest consequent and the BackImage operator is essentially the weakest precondition. Since the transition relation is defined in a high-level language, computing the image operators is the same as computing a precondition or postcondition of a transformational program — the program that defines the transition relation. In other words, verification proceeds in the temporal logic model checking framework of iterating to fixed points, but on each iteration, the image computation consists of performing an automatic program-logic verification of a loop-free transformational program. The efficient image computation presented in the next section can be viewed as the application of dynamic logic rules of inference to break down the verification of a program into the verification of its subparts [79]. This observation suggests that the image computation will be a recursive traversal of the high-level description of the transition relation.

3.2 Image Computation Procedure

For simplicity, I will only describe the computation for the Image operator. The computations for PreImage and BackImage are similar.

The basic idea is to define Image(δ, Z) to assume the set of states Z is represented by a Boolean formula, but to assume the transition relation is represented in the high-level description language rather than as a Boolean formula. The definition of Image can therefore recursively traverse the structure of the program defining the transition relation. Since the high-level language statements are defined recursively, Image has a recursive structure identical to that of TR:

Sequence: The image is propagated sequentially through the statements in the sequence:

$$\text{Image}(s_1; \ldots; s_n, Z) \stackrel{\text{def}}{=} \text{Image}(s_n, \ldots \text{Image}(s_1, Z) \ldots).$$

If-Then-Else: The image computation proceeds down both branches separately:

$$\begin{aligned} \text{Image}(\text{if } c \text{ then } s_1 \text{ else } s_2 \text{ endif}, Z) \stackrel{\text{def}}{=} & \ \text{Image}(s_1, \text{BFUN}(c) \wedge Z) \vee \\ & \ \text{Image}(s_2, \neg\text{BFUN}(c) \wedge Z). \end{aligned}$$

Non-Deterministic Choice: The result is just the OR of the images under each option:

$$\text{Image}(s_1 | \ldots | s_n, Z) \stackrel{\text{def}}{=} \text{Image}(s_1, Z) \vee \cdots \vee \text{Image}(s_n, Z).$$

This case is Burch *et al.*'s disjunctive partitioned transition relation [9].

The base case of the recursion is anything (like an assignment statement) that doesn't fit any case above. In the base case, we build the BDD for the (sub-)transition relation using TR,[3] and compute the image using the original definition (p. 18). In principle, we could even resort to using Boolean functional vectors or conjunctive partitioned

[3]Chapter 2 wasn't a total waste. Indeed, only the part about control flow is displaced by this chapter's image computation.

transition relations to reduce the base case further, but in practice, building the BDD for the transition relation in the base case has been adequate. Proving the equivalence of the recursively computed Image presented here to the original definition of Image is straightforward.

The recursive traversal can be cut short at any point. Thus, if some portion of the high-level description has an efficient BDD representation, we can build the BDD for that part of the system and treat it as a base case.

The computation for PreImage is almost identical to the computation above, except that the direction of the transition relations is reversed, since (in dynamic logic) Image is $\langle \delta^- \rangle$ and PreImage is $\langle \delta \rangle$. BackImage can be computed via a similar computation, or by noting that BackImage$(\delta, Z) = \neg$PreImage$(\delta, \neg Z)$, since negation is constant time for efficient BDD implementations [5].

Note that although this image computation is a simple trick, it successfully avoids what otherwise is a show-stopping obstacle preventing BDD-based verification of many systems. For example, consider the task of computing the transition relation for a number of systems.[4] On a small model of a radix-4 SRT divider, the conventional method could not complete in 20 minutes, using 74MB at that point. In contrast, this chapter's recursive computation completed in 2 seconds and just over 1MB. Similarly, on an industrial distributed linked-list protocol, the conventional method required 3 minutes 20 seconds and 56MB of memory, whereas the recursive computation completed in 17 seconds and 5MB. On an industrial directory-based cache coherence protocol, both the conventional method and the disjunctive partitioned transition relation exceeded 120MB without completing, but the recursive computation completed in 5 minutes 33 seconds and 36MB.

[4]The actual task was to verify the property True via a backward traversal. Using this task makes a fair comparison, as it forces this chapter's recursive image computation to traverse the entire description. The examples were run using the Ever verifier (See Appendix A.) on a Sun SPARC 20.

Chapter 4

BDD Blow-Up Representing Sets of States

Chapter Overview

The remaining stumbling block is that the BDDs representing sets of states become too large during the verification process. This chapter argues that for high-level or system-level verification with conventional BDD-based algorithms, this BDD-size blow-up is typical and should be expected, rather than being a rare surprise or the result of an unusually poor choice of variable ordering. The reason is that the high-level view of the system exposes the myriad relationships between various parts of the system, and these relationships produce myriad relationships between the values of the BDD variables, resulting in BDD-size blow-up regardless of variable ordering. This intuitive explanation of why BDD-size blow-up is so common for this type of verification naturally suggests two approaches to avoid the problem — functionally dependent variables and implicitly conjoined BDDs — that are the subject of the following two chapters.

4.1 A Historical Note

Allow me to start this chapter by continuing the personal anecdote with which I started Chapter 2. Recall our frustration trying to perform high-level verification with low-level tools. As a result, we created the Murφ project to build a state-of-the-art high-level BDD-based verifier. We had learned from painful experience, however, the importance of attacking real problems in order to determine the practical research issues, so we quickly created a non-BDD-based, explicit state enumeration verifier to jumpstart the process of writing real verification examples. We intended to have several complex real examples completed and ready to verify by the time we finished the BDD-based version of the verifier, which we were sure would be vastly superior. Meanwhile, others were publishing a steady stream of impressive BDD-based verification results: 10^{20} states [10]! 10^{120} states [9]! 10^{1300} states [19]! And beyond![1]

To our surprise, the non-BDD-based verifier turned out to be very useful. The combination of a high-level, user-friendly description language, a fast, easy-to-use verifier, a big main memory, and clever verification strategy produced real value in helping debug real system-level designs [27]. To our surprise and dismay, the BDD-based verifier was unimpressive. Although the verifier worked on small examples, and although large verifiable examples with impressive state counts were easy to contrive, on the complex real examples we had written, the BDD-based verifier performed comparably to and sometimes considerably worse than the non-BDD-based verifier. Even when verifying fairly small designs, the BDDs representing sets of states would often be quite large, sometimes unmanageably so.

Since this experience flew in the face of the then prevailing wisdom that BDD-based symbolic model checking was indubitably superior to explicit state exploration, we sought to figure out what we were doing wrong. Extensive variable-order twiddling, discussions with other researchers, and literature searches failed to produce any breakthroughs that solved our problems. Apparently, the problems we were verifying

[1]For a while, this became an inside joke among verification researchers. At one conference, a speaker whose work dealt with infinite-state systems jokingly subtitled his talk "10^{∞} States and Beyond."

were somehow different from those that other people were verifying, and this difference somehow made the problems unamenable to the BDD-based techniques available at the time.

Eventually, we discovered an explanation. The number of states in a system, the number of latches or variables, the number of gates, the number of lines of code, and other standard measures of the complexity of a system are all poor measures of how difficult the system is to verify with BDDs. The real issue is BDD size, and BDD size doesn't correlate well with any of these measures. Instead, BDD size captures some notion of the communication complexity in the design. For example, a system with a thousand latches, all of which can take any value at any time, is trivially easy to verify with BDDs, yet has an enormous number of states. On the other hand, even a small 16-bit multiplier produces huge BDDs. The high-level verification problems we were attacking tend to produce large BDDs. To understand why, we need some intuition on what makes a BDD big.

4.2 BDD Size Intuition

As mentioned in Chapter 1, a simple counting argument shows that almost all Boolean functions require large BDD representations. A similar argument, however, also shows that almost all Boolean functions require large representations in any reasonable representation scheme, including digital circuits. Rather than proving that BDDs and digital circuits are useless, the argument really shows that most Boolean functions are not practically interesting. As we have seen, the BDDs for many common functions are small.

A more informative approach is to characterize which functions have small BDD representations or to develop techniques to prove lower bounds on the size of the BDD for a given function. A few results have appeared along these lines. Ishiura and Yajima show that the class of functions that have polynomial (in the number of variables) size BDDs is exactly the class of functions computable by a one-way off-line logspace Turing machine [47]. Meinel compares the expressive power of polynomial-size BDDs to several other varieties of branching programs [68]. A particularly useful

result is the fooling set method developed by Bryant [7] based on an earlier VLSI complexity result. The method is currently the only general technique known for proving lower bounds on the size of the BDD for a specific function.[2] The basic idea is to allow an adversary to partition inputs of the function arbitrarily into two sets L and R of roughly equal size.[3] A truth assignment to the input variables in set R is said to distinguish between two truth assignments to the input variables in set L if the function produces different values depending on which of the two truth assignments to the inputs in L is used. A fooling set is a set of truth assignments to the inputs in L such that for any two truth assignments in the set, there exists a truth assignment for the inputs in R that distinguishes between them. Bryant's main result is that if for all partitions we can construct a fooling set of size c^n, then the size of the BDD for the function is lower-bounded by $\Omega(c^n)$.

The intuition behind the fooling set method provides general insight into causes of BDD-size blowup. The basic idea is that information only flows downward in a BDD. If we cut the BDD at any point in the variable order (so that all the nodes above the cut come earlier in the variable order than all the nodes below the cut), the BDD edges crossing this cut must terminate at as many different BDD nodes as are required to encode all the information needed from the variables above the cut. Thus, for a given variable ordering and cut, with the variables above the cut forming the set L and the variables below the cut forming the set R, if we can construct a fooling set of size k, then the BDD edges crossing the cut must terminate in at least k different BDD nodes. If there weren't at least k BDD nodes, then at least two truth assignments in the fooling set would end up in the same BDD node after crossing the cut, whereupon the truth assignment to the set R that should have distinguished between these two fooling set truth assignments can't possibly do so. Thus, the BDD must have size at least k. See Figure 4.1.

Looking at this intuition a different way, consider two variables x and y whose values are related. The BDD must somehow remember the truth assignment given

[2] More precisely, for a family of Boolean functions parameterized by n, e.g., $n \times n$-bit multipliers, the method produces a lower bound as a function of n.

[3] Actually, we may designate a subset of the inputs and a constant fraction between 0 and 1 and require that that fraction of the designated subset of inputs appears in the set L.

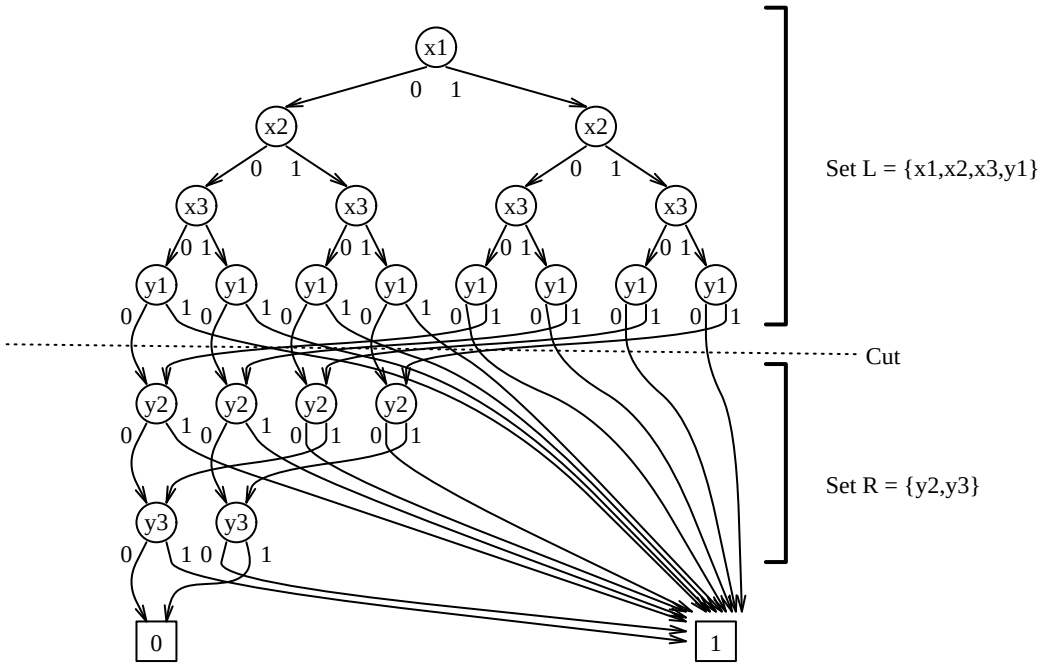


Figure 4.1: Fooling Set Intuition: The BDD shown represents the function $(x_1 \oplus y_1) + (x_2 \oplus y_2) + (x_3 \oplus y_3)$ and is the same as in Figure 1.3(a). In this figure, the cut partitions the variables into the set $L = \{x_1, x_2, x_3, y_1\}$ and the set $R = \{y_2, y_3\}$. A possible fooling set (with the bit-vectors indicating the truth assignment to x_1, x_2, x_3, and y_1 in that order) is $\{0001, 0000, 0010, 0100, 0110\}$. For any two truth assignments in the fooling set, for example 0001 and 0000, there exists a truth assignment to the variables in set R, such as 00, that distinguishes between them, as 000100 yields $(0 \oplus 1) + (0 \oplus 0) + (0 \oplus 0)$ which evaluates to 1, but 000000 yields $(0 \oplus 0) + (0 \oplus 0) + (0 \oplus 0)$ which evaluates to 0. No fooling set has size larger than 5, so the BDD edges that cross the cut terminate in 5 different nodes.

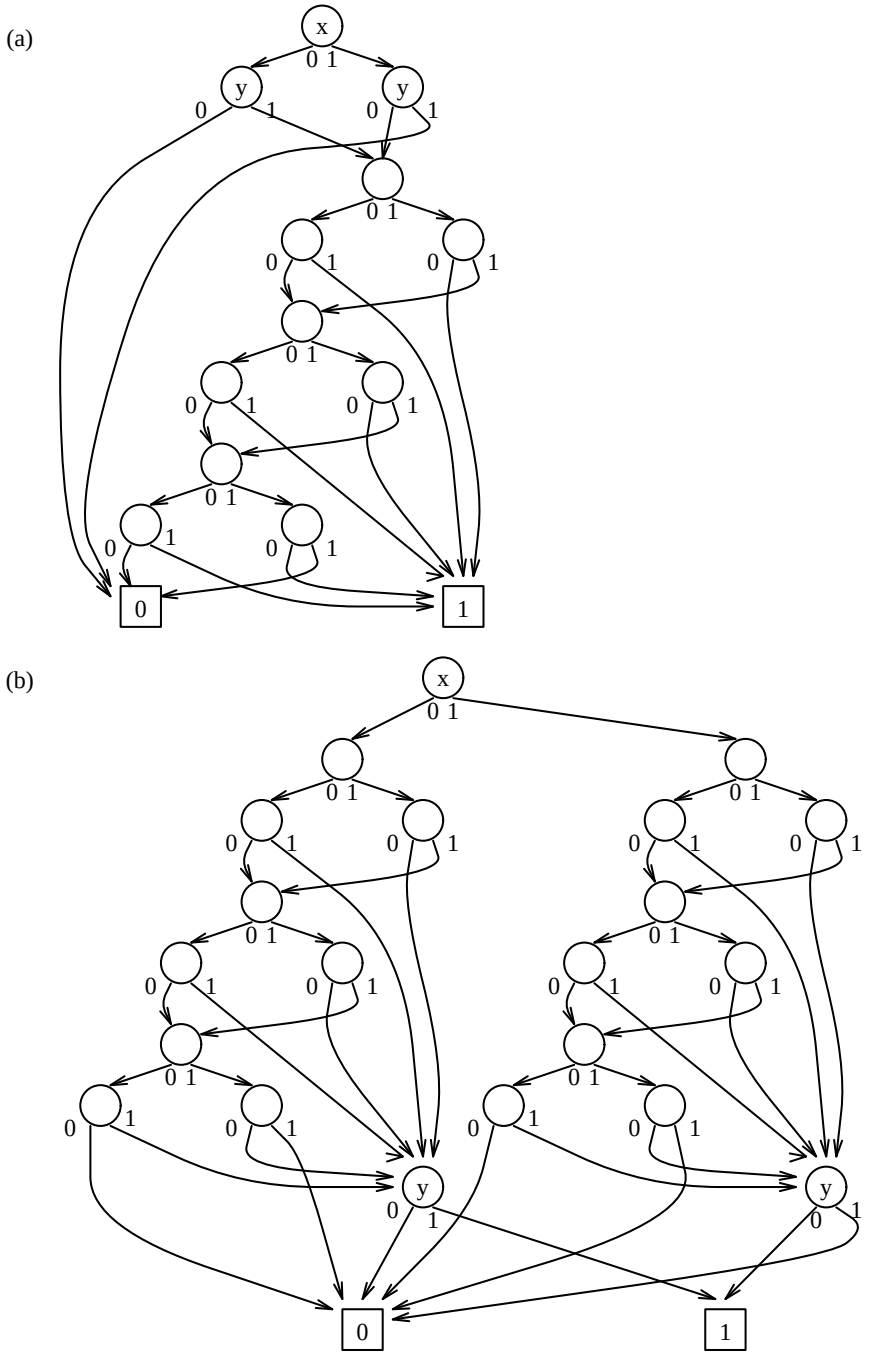


Figure 4.2: Effect of Separating Related Variables: This figure shows BDDs for a function $(x \oplus y) \wedge f$ in which the variables x and y have related values. In part (a), x and y are near each other in the variable order. In part (b), they are separated, forcing the BDD to duplicate the nodes between x and y.

to x until it reaches y. Therefore, at any cut of the BDD between x and y, we can create a larger fooling set because we need to distinguish those truth assignments in which x is true from those in which x is false. In other words, the number of BDD nodes are roughly doubled between x and y. Hence, to keep the BDD small, intuition tells us that we should keep related variables close together. See Figure 4.2. What happens if the function has myriad such relationships between variables? In this case, no matter which related variables we put close together in the variable order, other related variables will be forced farther apart. Thus, no matter what variable order we choose, intuition tells us to expect a large BDD.

4.3 High-Level Verification Is Hard for BDDs

Unfortunately, a high-level model of a system tends to have numerous relationships between the state variables for various parts of the system. To understand why this situation is so common, consider the diagram in Figure 4.3. This figure represents an enormous range of systems viewed at a high level. The processes could be processors in a multiprocessor, blocks of logic on a chip, components in a system, etc. They communicate via some sort of interconnection network, which could be a bus, shared memory, a switched network, assorted control lines, etc. Typically, each part of the system has some local state variables that encode its own state and its knowledge of what's happening in the rest of the system. As the system runs, relationships between the state variables in various parts of the system are established, maintained, and occasionally destroyed. For example, in a cache-coherent multiprocessor, each processor has a local cache that sometimes equals what is in certain memory locations and a local cache controller whose state provides some indication of what transactions are occurring throughout the system. While the system runs, numerous invariants will hold such as: If Processor A is waiting for data from Processor B, then Processor B must have the data, the directory must indicate that Processor B has the data, the network must contain a message dealing with this transaction, and so forth. Modeled at a high level, similar properties typically hold between all parts of the system.

Suppose we try to use the standard BDD algorithms to verify a system at this

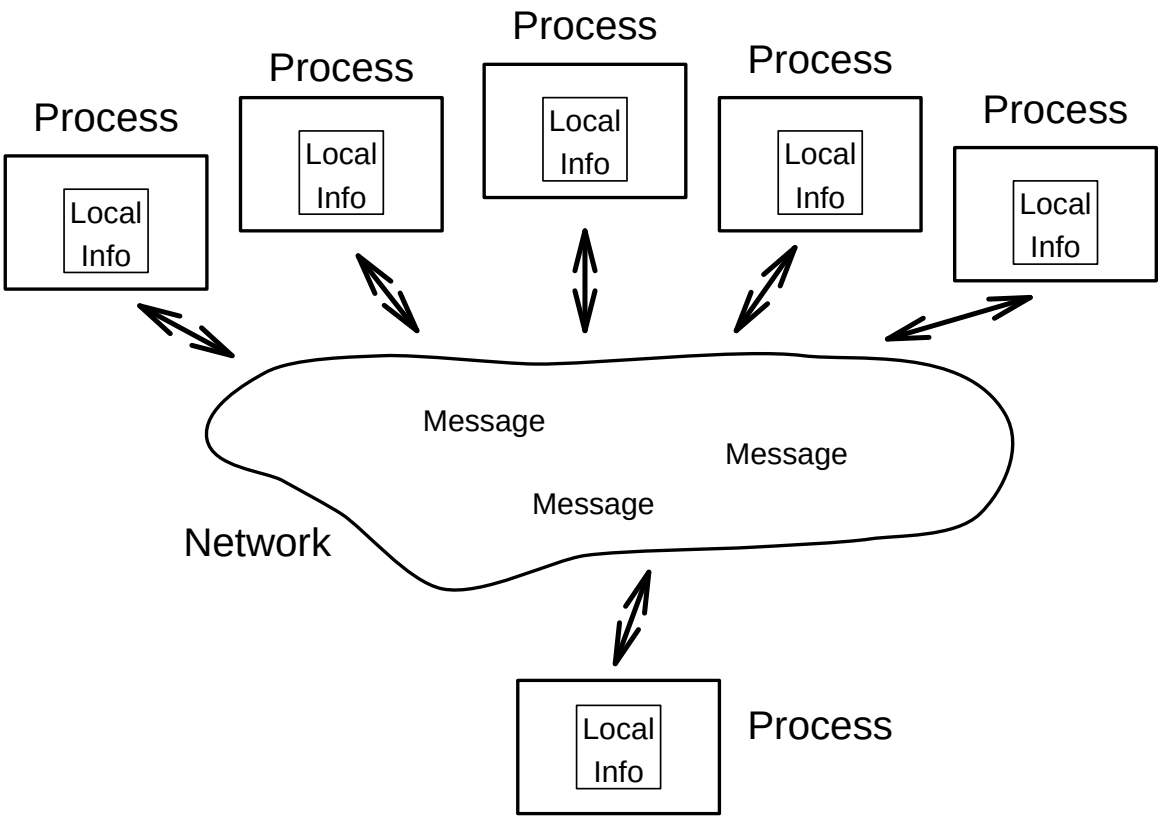


Figure 4.3: A Typical System-Level Model Has Many Relationships Among Variables: This figure captures an enormous range of real designs at a system level. At this very high-level, most systems consist of a number of components, such as processors in a multiprocessor or logic blocks on a chip, communicating via an interconnection network of some sort, such as a bus or point-to-point channels. Each part of the system will have some local state. This local state is typically strongly related to the state in other parts of the system because the local state reflects the status of on-going interactions with other parts of the system. For example, consider the hand-shaking required in a communication protocol. Thus, regardless of variable order, the BDD representing the set of all reachable states will be large.

high level. For a forward traversal, the set of all reachable states will only include those states that satisfy all of the relationships between the state variables. For a backward traversal, if the property being checked is non-trivial, convergence will occur only after the BDD represents only states that satisfy all of the relationships between the state variables. In either case, the verifier must generate a BDD for a function in which most of the variables are related. Thus, no matter what variable order we choose, some closely related variables will be forced to be far apart, so the BDD will be large.

Armed with this intuition, we should expect the BDDs in high-level verification to be large, since the BDDs must encode the myriad relationships between the state variables of the system. Fortunately, this intuition also gives us a clue to avoiding the BDD-size blowup: try to eliminate relationships among the variables in a BDD. This clue is the heart of the two methods I present in the next two chapters to avoid BDD blowup representing sets of states during high-level verification. Chapter 5 presents a method that exploits user-specified functional relationships between state variables. Chapter 6 presents a means to use a list of small BDDs instead of a single large BDD to represent the set of states satisfying a property, thereby separating the relationships among the state variables into separate BDDs.

Chapter 5

Functionally Dependent Variables

Chapter Overview[1]

The preceding chapter explained why the numerous relationships among the the variables of a system produce BDD-size blowup. Frequently, some of these relationships take the form of functional dependency — one variable is always a function of other (independent) variables in the system. This chapter presents a novel means to use user-specified functional dependencies among the BDD variables to greatly reduce BDD sizes, while also checking that the user-specified functional relationship is true. On an example, the new algorithm reduces exponentially-sized BDDs to provably $O(n \log n)$-sized ones.

5.1 Introduction

The preceding chapter tells us that interrelationships between variables in a system produce BDD-size blowup, so we should try to minimize or avoid them. This chapter identifies a common form that some of these interrelationships take — one variable is a function of others in the system — and finds a method to avoid the blowup caused by this relationship.

[1]This chapter is based on material first published in the *30th Design Automation Conference*, 1993 [43].

A variable in a system is a functionally dependent variable if there exists a function that always gives the value of the variable in terms of other variables in the system, assuming the system is operating correctly. Concurrent systems typically have many functionally dependent variables: individual processes in a concurrent system must maintain a local image of certain aspects of the global state of the system. For example, in a directory-based cache-coherence protocol, the directory records which processors have a given memory line cached. The state of the directory is therefore a function of the states of the processor caches.

For another example, suppose a process sends a message requesting a service from another process, and then enters a state waiting for a reply message. Here, the process is in the "wait for reply" state exactly when the original message is in some transmission queue, a remote server process is in an appropriate state processing the request, or the reply message is in the server's transmission queue. In this case, the state variable which stores the "wait for reply" state locally is functionally dependent on the state variables encoding the queues and the state of the server process.

Functionally dependent variables have a high degree of correlation with the variables on which they depend. In a concurrent system with many dependent variables, it becomes difficult or impossible to order the variables so that the dependent variables are close to the variables upon which they depend. As we saw in Chapter 4, the result is BDD-size blowup.

The few case studies of successful protocol verification with BDDs also illustrate the importance of functionally dependent variables, but by their relative absence. In verifying the cache consistency protocol of the Encore Gigamax, for example, McMillan and Schwalbe's careful choice of abstraction as well as the bus-snooping nature of the protocol combine to create a description of the system with few dependent variables [67]. Similarly, the Controller Area Network protocol verified by Chiodo *et al.* has almost no dependent variables because it relies on a broadcast bus with automatic collision detection, and the individual nodes have no information about the rest of the system [15]. Academic examples like dining philosophers also tend not to have many dependent variables because each process has very little state and behaves largely independently of the rest of the system. If BDD-based verification is to apply

to a broader range of problems, dealing with functionally dependent variables is a must.

This chapter presents a new idea for reducing the explosive effects of functionally dependent variables. The user declares the dependent variables as a user-specified function of the independent variables. During verification, the BDD representing the set of reachable states contains only the *independent* variables. (The values of dependent variables are implicit from the user-supplied function.) The fundamental difficulty of this approach is that, typically, the dependent variables are only really dependent when the protocol is error-free. If the protocol malfunctions due to a description or design error, the nature of the problem is often a discrepancy between the dependent and independent variables (e.g., the local image of the global state is not a true image). Hence, the method must check at each step of verification whether the functional dependency continues to hold. In other words, the method *assumes* that the functional dependency holds, thereby realizing a substantial BDD-size savings, while *simultaneously verifying that the assumption of functional dependency was correct.*

5.2 Theoretical Basis

We first need to formalize our notion of a functionally dependent variable.

Definition 5.1 (Functionally Dependent Variable) *A BDD variable y is functionally dependent on a set of BDD variables $x_1, \ldots, x_n$ iff there exists a function f such that $y = f(x_1, \ldots, x_n)$. For notational convenience, we extend this definition in the obvious way to a vector of BDD variables $\vec{y}$: $\vec{y}$ is functionally dependent on $\vec{x}$ iff there exists a vector-valued function f such that $\vec{y} = f(\vec{x})$.*

Let us now review our original verification problem in light of the concept of functionally dependent variables. For concreteness, divide all of the state variables in the system being verified into two vectors of BDD variables: $\vec{x}$ and $\vec{y}$. As before, we specify the set of start states $S(\vec{x}, \vec{y})$, the non-deterministic next-state relation

$\delta(\vec{x}, \vec{y}, \vec{x'}, \vec{y'})$, and the set of states satisfying the verification condition $G(\vec{x}, \vec{y})$.[2] Additionally, we specify a function f such that $\vec{y} = f(\vec{x})$ should hold for all reachable states. The goal is still to verify that all reachable states satisfy G, but we now wish to use f to eliminate the BDD variables $\vec{y}$, thereby reducing BDD size, while simultaneously proving that $\vec{y} = f(\vec{x})$ in all reachable states.

At first glance, this task appears impossible. To reduce BDD size, we must eliminate the BDD variables for $\vec{y}$ by assuming $\vec{y} = f(\vec{x})$ and substituting $f(\vec{x})$ in place of $\vec{y}$, but if we assume $\vec{y} = f(\vec{x})$, how can we possibly check that the assumption is correct? Furthermore, we cannot take advantage of a high-level language description of δ to check the functional dependency step-by-step through the description as we did in Chapter 3 for image computation because, even if the functional dependency is true in every reachable state, the functional dependency might not hold at intermediate points in the next-state relation. For example, suppose the functional dependence is that $\vec{y} = \vec{x}$, and the next-state relation includes the following code:

```
x := 0; -- Assignment 1
-- Additional code could go here.
y := 0; -- Assignment 2
```

After the first assignment, $\vec{y}$ no longer necessarily equals $\vec{x}$. Indeed, at that point, $\vec{y}$ may not even be any function of $\vec{x}$, since $\vec{x}$ must be 0, whereas $\vec{y}$ can still have any value it used to have. After the second assignment, however, the functional dependence holds once again.

The solution comes through the use of Boolean functional vectors [21]. Recall from Chapter 3 that a Boolean functional vector represents the transition structure of a system as a function from the current state to the next state. For each BDD variable, a separate BDD computes the value of that variable in the next state as a function of the current state BDD variables. For example, if we have a system with exactly two state variables u and v, the Boolean functional vector (u, v) indicates the transition structure in which the state never changes (corresponding to transition

[2] As in Chapters 1 and 2, the pseudo-first-order notation simplify clarifies which variables each BDD refers to.

relation $(u' = u) \wedge (v' = v)$), the Boolean functional vector (v, u) indicates that the variables u and v swap values at each time step (corresponding to transition relation $(u' = v) \wedge (v' = u)$), and the Boolean functional vector $(v, u \oplus v)$ produces a linear feedback shift register (corresponding to transition relation $(u' = v) \wedge (v' = u \oplus v)$).

Suppose we were verifying a *deterministic* automaton with a total next-state function. In that case, we could compute a Boolean functional vector for the next state as a function of the current state: $\vec{x'} = \delta_x(\vec{x}, \vec{y})$ and $\vec{y'} = \delta_y(\vec{x}, \vec{y})$. We could then both check and assume the functional dependency as follows:

Theorem 5.1 *Suppose that $R_i(\vec{x})$ is the set of states reachable in i or fewer steps from the start states, and that the total deterministic next-state relation has next-state Boolean functional vectors $\delta_x(\vec{x}, \vec{y})$ and $\delta_y(\vec{x}, \vec{y})$. If the functional dependency $\vec{y} = f(\vec{x})$ is true for $R_i(\vec{x})$, then the functional dependency is also true for $R_{i+1}(\vec{x})$ iff*

$$R_i(\vec{x}) \Rightarrow [\delta_y(\vec{x}, f(\vec{x})) = f(\delta_x(\vec{x}, f(\vec{x})))] .$$

Proof: Since the functional dependency holds for $R_i(\vec{x})$, we can substitute $f(\vec{x})$ for $\vec{y}$ when computing the values of the next-state functions. The result follows trivially by substitution into the expression $R_i(\vec{x}) \Rightarrow [\vec{y'} = f(\vec{x'})]$, where the primed variables denote the value of the variable in the next state. □

The next-state relation δ is not, however, necessarily total and deterministic. We need, therefore, to extend the concept of Boolean functional vectors.

Definition 5.2 (MOCB) *A* **multi-set of conditional Boolean functional vectors** *(MOCB) is a multi-set of pairs (c_i, a_i), where the condition c_i is a Boolean-valued expression, and the action a_i is a Boolean functional vector. The value of a MOCB is defined to be the non-deterministic choice of one action a from the set of all actions a_i for which the corresponding condition c_i is true.*

Definition 5.3 (MOB) *A* **multi-set of Boolean functional vectors** *(MOB) is a MOCB in which all conditions c_i are identically equal to true.*

Both the MOCB and the MOB explicitly handle non-determinism. The more cumbersome MOCB can represent a partial next-state relation; the MOB can represent

only a total next-state relation, since Boolean functional vectors are total functions. In practice, restricting oneself to total next-state relations is easy: null actions can be represented by a self-looping transition, and terminating computations can be represented by a transition to a self-looping dead state. Furthermore, a high-level description language as described in Chapter 2 naturally generates a total next-state relation. Accordingly, I use the simpler and more efficient MOB; I will not consider the general MOCB in the remainder of this thesis.

With this extension of Boolean functional vectors, we can now consider the improved verification algorithm:

Algorithm 5.1 (Forward Traversal with Functionally Dependent Variables) *Let the state variables be $\vec{x}$, with functionally dependent variables $\vec{y}$ defined as $\vec{y} = f(\vec{x})$. Let the set of start states be $S(\vec{x})$ and the total, possibly non-deterministic next-state relation be $\delta(\vec{x}, \vec{y}, \vec{x'}, \vec{y'})$. Let the verification condition be $G(\vec{x}, \vec{y})$.*

1. *Compute the MOB for $\delta(\vec{x}, \vec{y}, \vec{x'}, \vec{y'})$.*
2. *Let $R_0(\vec{x}) = S(\vec{x})$.*
3. *Loop*
   (a) *Check $R_i(\vec{x}) \Rightarrow G(\vec{x}, f(\vec{x}))$. If not, generate counterexample trace.*
   (b) *For each Boolean functional vector (δ_x, δ_y) in the MOB, apply the check from Theorem 5.1:*

$$R_i(\vec{x}) \Rightarrow [\delta_y(\vec{x}, f(\vec{x})) = f(\delta_x(\vec{x}, f(\vec{x})))] .$$

   If not, generate counterexample trace.
   (c) *$R_{i+1}(\vec{x}) = R_i(\vec{x}) \vee Image(R_i, \delta)(\vec{x})$.*
   (d) *If $R_{i+1}(\vec{x}) = R_i(\vec{x})$, terminate successfully.*

As can be seen, this algorithm is essentially the standard forward traversal and is open to the standard optimizations. The only differences are the generation of the MOB, and the step to check that the functional dependency will be preserved on each iteration.

5.3 Implementation

I have implemented the above algorithm in the Ever verifier, described in Appendix A. To use functionally dependent variables, the user simply declares a variable to be functionally dependent, giving the function that specifies the dependency. From that point on, the verifier handles everything automatically. The interesting implementation issues are how to build the MOB and how to build the Boolean functional vectors within the MOB.

I assume the high-level description language has the same feature set and Algol-style structure as in Chapter 2. Imposing a few restrictions on the description language ensures that the transition relation is total (so we don't need a MOCB), and greatly simplifies translating a high-level description into a MOB:

- The description must be structured as the non-deterministic choice among deterministic actions, in the manner of both Murφ and Unity. On any branch of the parse tree for the next-state relation, therefore, all non-deterministic choice expressed as a disjunction between sub-relations occurs first. This restriction keeps the degree of non-deterministic choice reasonably small.

- A (sub-)next-state relation must not evaluate to false. To express a null action, use a self-loop; to express a terminating computation, create a transition to a self-looping dead state. This restriction, along with the next one, precludes writing partial next-state relations.

- All assignments must be deterministic. Eliminating non-deterministic assignment both discourages descriptions resulting in very large MOBs and also closes a loop-hole that would have allowed creating a partial next-state relation.

With these restrictions, the next-state relation is essentially a tree with deterministic next-state relations at the leaves. Hence, building the MOB consists simply of traversing the disjunctions at the top levels of the parse tree, collecting the Boolean functional vectors built for each deterministic next-state sub-relation.

Building the Boolean functional vector for a deterministic next-state relation is a

more complex computation. The computation is a syntax-directed recursive translation, much like the one described in Chapter 2. Here, however, as the translation routines traverse the high-level description, they always maintain a Boolean functional vector that indicates the values of all variables at that point in the description as a function of the original current-state variables. At the start, this Boolean functional vector is simply the array V of all BDD variables for the current state variables (defined on page 26), followed by the BDDs for $f(\vec{x})$ to give the values of the dependent variables. As each statement is processed, the Boolean functional vector is updated to reflect the effect of that statement.

More formally, two key differences distinguish the translation in this chapter from the translation in Chapter 2. The first difference is that the top-level translation is handled by a new function BFV, which returns a Boolean functional vector, rather than by the function TR, which returned a transition relation. The other difference is that the helper translation functions in Chapter 2 (e.g., BFUN, BVEC, SELECTBITS, etc.), must be modified to take a Boolean functional vector as an additional input parameter and to use this Boolean functional vector instead of the array V of BDD variables to supply the current values of the state variables. Let the suffix "_BFV" attached to the name of a function denote this modified version. For example, we can define SELECTBITS_BFV as follows (cf. p. 27):

$$\text{SELECTBITS_BFV}(\epsilon, \text{offset}, \text{size}, \delta) \stackrel{\text{def}}{=} \delta_{\text{offset}}[\text{size}]$$

$$\text{SELECTBITS_BFV}(\mathit{field_name} :: \sigma, \text{offset}, \text{size}, \delta) \stackrel{\text{def}}{=} \text{SELECTBITS_BFV}(\sigma, \text{offset} + \text{OFFSET}(\mathit{field_name}), \text{size}, \delta)$$

$$\text{SELECTBITS_BFV}(\mathit{index_expr} :: \sigma, \text{offset}, \text{size}, \delta) \stackrel{\text{def}}{=} \bigwedge_{i=l}^{u} \left[\begin{array}{c} \text{BVEC}(\mathit{index_expr}) \stackrel{*}{=} \text{BVEC}(i) \\ \stackrel{*}{\Rightarrow} \\ \text{SELECTBITS_BFV}(\sigma, \text{offset} + \text{OFFSET}(i), \text{size}, \delta) \end{array} \right],$$

where δ is the Boolean functional vector passed into the function and the other notation is the same as in Chapter 2. Since we can consider the array V to be a

Boolean functional vector, the functions in Chapter 2 are just special cases of the ones defined here, e.g., if the user does not specify any functionally dependent variables, then for any part of the description P, the function BFUN(P) is exactly the same as the function BFUN_BFV(P, V).

We can now define recursively the function BFV that takes (a fragment of) a deterministic high-level description and a Boolean functional vector giving the values of the variables at the start of the fragment and that returns the Boolean functional vector giving the values of the variables after the fragment has executed. Permissible description-language constructs are null statements, deterministic assignment statements, sequences of statements, and if-then-else statements. Define BFV as follows, where δ is the current Boolean functional vector:

Null Statement: The current Boolean functional vector is passed on unchanged:

$$\text{BFV}(\text{null}, \delta) \stackrel{\text{def}}{=} \delta.$$

Assignment: An assignment statement is handled by a helper function (described below) similar to the ASSIGN function defined in Chapter 2:

$$\text{BFV}(var := expr, \delta) \stackrel{\text{def}}{=} \text{ASSIGN_BFV}(\rho, 0, \text{SIZEOF}(\rho), expr, \delta),$$

where ASSIGN_BFV is defined below and ρ is the list of modifiers corresponding to the variable reference *var*, exactly as in Chapter 2 (page 30).

Sequence: For a sequence of statements, BFV called on the first statement returns the correct Boolean functional vector to start the processing of the rest of the statements:

$$\text{BFV}(s_1; s_2; \ldots; s_n, \delta) \stackrel{\text{def}}{=} \text{BFV}(s_2; \ldots; s_n, \text{BFV}(s_1, \delta)).$$

If-Then-Else: BFV simply uses the condition to select which branch applies:

$$\text{BFV}(\text{if } c \text{ then } s_1 \text{ else } s_2 \text{ endif}, \delta) \stackrel{\text{def}}{=}$$

$$(\text{BFUN_BFV}(c,\delta) \stackrel{*}{\Rightarrow} \text{BFV}(s_1,\delta)) \wedge$$
$$(\neg\text{BFUN_BFV}(c,\delta) \stackrel{*}{\Rightarrow} \text{BFV}(s_2,\delta)),$$

where $\stackrel{*}{\Rightarrow}$ denotes a single BDD implying each component of a vector of BDDs, and the $\wedge$ is applied component-wise.

The definition of the ASSIGN_BFV function is very similar to that of the ASSIGN function described in Chapter 2. The main difference is that ASSIGN_BFV collects vectors of BDDs giving the next value of each variable and assembles the vectors into longer vectors, whereas ASSIGN collects BDDs expressing the relationship between some current and next-state variables and conjoins them together into a single BDD relating all of the state variables. Let $\odot$ denote the operation of concatenating two vectors of BDDs into a longer vector, e.g., $(v, u \oplus v) \odot (u, v)$ is $(v, u \oplus v, u, v)$. ASSIGN_BFV is defined recursively as follows:

$$\text{ASSIGN_BFV}(\epsilon, \text{offset}, \text{size}, \text{expr}, \delta) \stackrel{\text{def}}{=} \text{BVEC_BFV}(\text{expr}, \delta)_1[\text{size}]$$

$$\text{ASSIGN_BFV}(\mathit{field_name} :: \sigma, \text{offset}, \text{size}, \text{expr}, \delta) \stackrel{\text{def}}{=}$$
$$\bigodot_{f \in \mathit{record_fields}} \left[\begin{array}{l} \text{if} \quad f = \mathit{field_name} \\ \text{then} \\ \quad \text{ASSIGN_BFV}(\sigma, \text{offset} + \text{OFFSET}(f), \text{size}, \text{expr}, \delta) \\ \text{else} \\ \quad \delta_{\text{offset}+\text{OFFSET}(f)}[\text{SIZEOF}(f)] \end{array} \right]$$

$$\text{ASSIGN_BFV}(\mathit{index_expr} :: \sigma, \text{offset}, \text{size}, \text{expr}, \delta) \stackrel{\text{def}}{=}$$
$$\bigodot_{i=l}^{u} \left[\begin{array}{l} \text{if} \quad \text{BVEC_BFV}(\mathit{index_expr}, \delta) \stackrel{*}{=} \text{BVEC}(i) \\ \text{then} \\ \quad \text{ASSIGN_BFV}(\sigma, \text{offset} + \text{OFFSET}(i), \text{size}, \text{expr}, \delta) \\ \text{else} \\ \quad \delta_{\text{offset}+\text{OFFSET}(i)}[\text{SIZEOF}(\mathit{element})] \end{array} \right],$$

where σ is a list of modifiers, ϵ is the empty list, :: is the list `cons` operator, l and u

are the lower and upper bounds of the array, and the notation $B_i[s]$ applied to any vector of BDDs B denotes the s BDDs starting with the ith one.

A final implementation issue is how to compute the image operators (p. 18). One possibility, since we've already computed Boolean functional vectors for the next-state relation, is the straightforward extension of the Boolean functional vector image computation techniques [22, 85] to handle the non-determinism of MOBs. For the example that follows, however, I have identified a special case that allows a simpler solution. If the description of the next-state relation of the system does not rely on the updated value of functionally dependent variables, assignments to functionally dependent variables are no-ops when computing images. Hence, we can reuse the existing efficient image computation from Chapter 3 by simply ignoring those assignments. (Of course, the assignments are not ignored when computing the MOB for the next-state relation.)

5.4 A Benchmark Example

In many of our industrial examples, we need to model a communication network that doesn't preserve message order. In addition, individual processors keep track of what messages they have outstanding in the network. Clearly, each processor's message status is a function of the network. Furthermore, the correctness of each processor's status is a crucial component of the overall correctness of the system. Thus, this situation perfectly illustrates functionally dependent variables arising in practice.

This section's benchmark example distills the essential properties from the scenario above. Suppose we have a set of processors that non-deterministically issue requests into the network. Each request contains the processor's ID as a return address. A server non-deterministically pulls messages out of the network (modeling a non-order-preserving network), and places an acknowledgement back into the network. Eventually, the processor will receive the acknowledgement. When a processor sends a request, it increments a counter. When the processor receives an acknowledgement, it decrements the counter. See Figure 5.1.

For simplicity, assume there are n processors and that the network can hold n

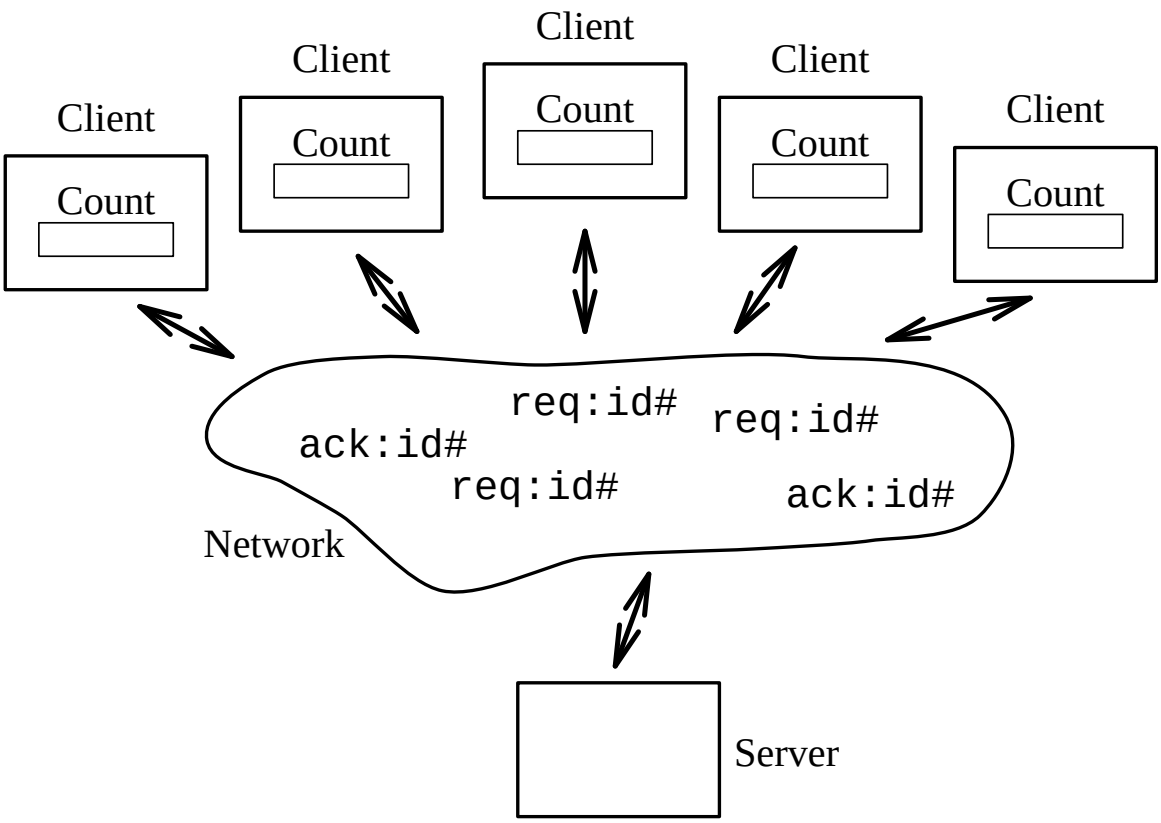


Figure 5.1: Network Example: Each client non-deterministically issues requests into a non-order-preserving network. A server non-deterministically receives requests and responds with acknowledgments to the requesting clients. Each client keeps a count of how many requests it has outstanding. If the system is operating correctly, this count should be a function of the network. The verification task is to verify that each client's count is correct.

messages. Integers will be k bits wide, with $k > \log_2 n$. Thus, we declare an array of processors numbered 0 through $n-1$. Each processor has a k-bit integer counter. We model the network as an n-element array (numbered 0 through $n-1$) of messages. Each message is a record with a valid bit, a request/acknowledge bit, and a k-bit integer return address. At each time step, any one processor can choose to send a request, or any one valid request in the network can be served (becoming an acknowledge), or any one acknowledgement in the network can be delivered back to the sender.

If we don't exploit the functional dependency, the BDD for the set of reachable states will clearly be exponential for any straightforward variable ordering. For instance, if the processor counters come first in the variable order, any truth assignment such that the sum of the counts is less than or equal to n may or may not be a reachable state of the system depending on what the truth assignment for the network is. (If the sum of the counts is greater than n, the state cannot be reachable since the network has size n.) Thus, any of these truth assignments to the processor counters is an element in a fooling set. The size of the fooling set is $\sum_{i=0}^{n} \binom{n-1+i}{i}$, since $\binom{n-1+i}{i}$ is the number of ways for the sum of the n processor counters to be exactly i. The summation evaluates to $\binom{2n}{n}$, which is exponential. I conjecture that the BDD will be exponentially sized for any variable ordering.

In contrast, if we declare the processor counts as functionally dependent variables, we can prove low-order polynomial bounds on the sizes of the BDDs for sets of reachable states. The BDDs will contain variables for only the network. Assume a non-interleaved order for the network array. (All BDD variables for each array element are kept together in the order.)

Theorem 5.2 *With a non-interleaved variable order, the size of the BDD for the set of all reachable states (R_∞ in Algorithm 5.1) is $n(c_n+1)+2$, where c_n is the number of BDD nodes needed to show that an integer is less than n. Since the BDD size for integer comparison is at most the number of bits in the integer, $c_n \leq \lceil \log_2 n \rceil$. In practice, c_n is bounded above by a constant, the logarithm of the largest possible value of n.*

Proof: Any state is reachable provided that for each element in the network array,

if the valid bit is set, the return address must be less than n (a valid return address). The test for each array element requires $c_n + 1$ BDD nodes. The BDD for the set of all reachable states is just the conjunction of the n tests. Since the variable order is non-interleaved, the size of the conjunction is $n(c_n+1)$. Add 2 for the leaf nodes (add 1 instead if using complement edges as most efficient BDD implementations do [5]). □

Theorem 5.3 *With a non-interleaved variable order, the size of the BDD for the set of all states reachable in i or fewer iterations (R_i in Algorithm 5.1) is $(n - i/2)(i + 1)(c_n + 2) + (c_n + 1)(i - n) + 2$. Again, c_n is logarithmic in n and can be treated as a constant in practice.*

Proof: The intuition behind the proof is as follows. After i iterations, at most i "events" could have happened and still be visible, where an event is either the creation of a new request or the servicing of a request (making it an acknowledge). Therefore, at iteration i, any state is reachable as long as (1) for each element in the network array, if the valid bit is set, the return address must be less than n (as in Theorem 5.2), and (2) the state does not require more than i events to reach. Any state can be generated in $2n$ events, so there are $2n$ iterations before convergence and $0 \leq i \leq 2n$. The BDD for R_i roughly consists of i identical columns: each column checks that every element with its valid bit set has a valid return address, and the different columns track how many events the BDD has seen so far.

More formally, we can enumerate the BDD nodes in R_i by laying the BDD nodes out on a chessboard. Imagine an $n \times (i + 1)$ chessboard, with the rows numbered $1 \ldots n$ from top to bottom, and the columns numbered $0 \ldots i$ from left to right. In the square at row r and column c are all BDD nodes pertaining to the rth element in the network array (valid bit, req/ack bit, and return address for array element r) and assuming that c events have been noted in the BDD already. Subdivide each square into upper and lower parts: the upper part contains the valid and req/ack nodes; the lower, the c_n nodes to test the validity of the return address. The BDD nodes are interconnected as follows: the false edge from the valid node at square (r, c) goes to the valid node at square $(r + 1, c)$, the true edge from the valid node goes to the req/ack node at square (r, c), the edges from the req/ack node go to the return address nodes of square $(r, c+1)$ for a request and square $(r, c+2)$ for an acknowledge,

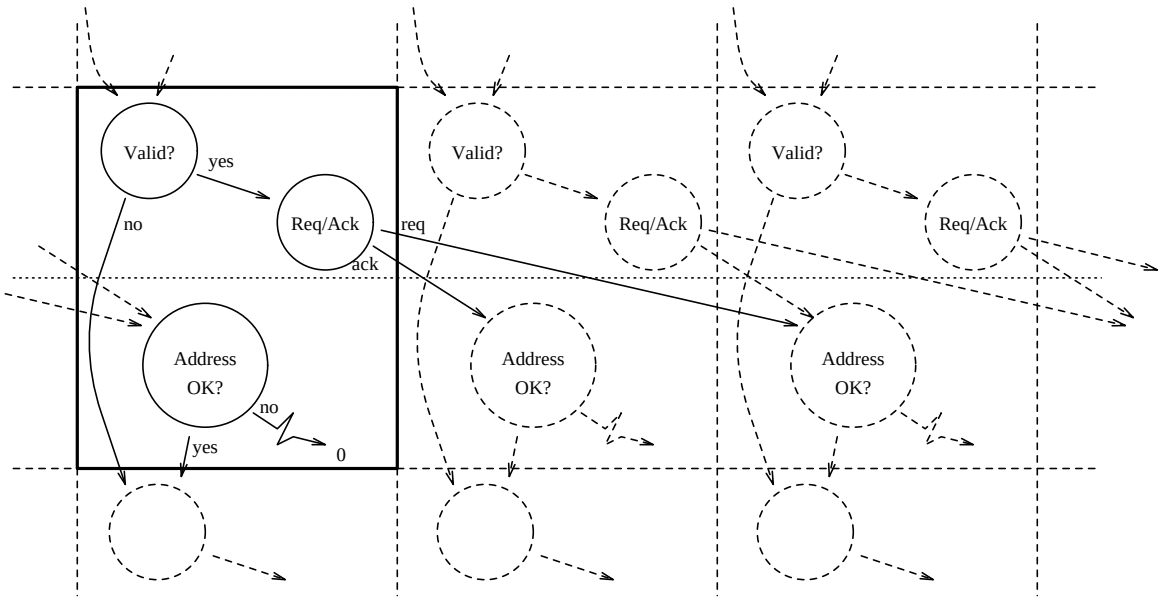


Figure 5.2: Each square of the chessboard contains BDD nodes arranged like this. The node labeled "Address OK?" is actually as many nodes as are needed to check the address. If the address is not OK, the edge leads directly to the 0 leaf node.

and the edges from the return address check at square (r, c) go to the valid node of square $(r + 1, c)$ if the address is valid and to the 0 leaf node if the address is invalid. See Figure 5.2. Any edge that crosses the right edge of the chessboard goes directly to the 0 leaf node; any edge that crosses the bottom of the chessboard goes to the 1 leaf node.

Note, however, that not every square has all of the BDD nodes in it. Since each network element can count as at most 2 events (if it's an acknowledgement), any square where $c > 2(r - 1)$ cannot be reached and therefore contains no BDD nodes. See Figure 5.3(a). Similarly, near the bottom of the BDD, if the number of possible events left cannot possibly exceed i, then the BDD need no longer count events, so the different columns collapse into a single chain. I will count these nodes separately as part of that chain. Therefore, I don't count the squares where $i - c > 2(n - r) + 1$. See Figure 5.3(b). The number of squares I do count is exactly $(n - i/2)(i + 1)$. In addition, of the squares that I do count, note that squares on the left edge of the board do not need nodes to check return addresses, since the left edge of the board is

reserved for the case where no messages have been valid, and note that the squares on the right edge of the board do not need the req/ack nodes, since either case results in too many events. Also, note that squares on the bottom boundary of the counted region do not have their address check nodes counted, as these will be collapsed into the chain counted separately since the event count no longer matters at that point. There are exactly $n - \lfloor i/2 \rfloor$ counted squares on the left edge and $n - \lceil i/2 \rceil$ counted squares on the right edge. See Figure 5.3(c). Now, count the nodes that check return addresses as part of the square below the one in which they occur. Therefore, the total number of BDD nodes is $(n - i/2)(i + 1)(c_n + 2)$ for the counted squares, minus $(n - \lfloor i/2 \rfloor)c_n$ for the missing return address nodes on the left edge, minus $n - \lceil i/2 \rceil$ for the missing req/ack nodes on the right edge, plus the BDD nodes in the separate chain. The separate chain consists of $\lfloor i/2 \rfloor$ layers of valid bit and return address checks, plus an extra return address check if i is odd. See Figure 5.3(d). Putting it all together yields:

$$\begin{aligned}(n - i/2)(i + 1)(c_n + 2) & \\ - \quad & (n - \lfloor i/2 \rfloor)c_n \\ - \quad & n - \lceil i/2 \rceil \\ + \quad & \lfloor i/2 \rfloor (c_n + 1) \\ + \quad & c_n[\text{if } i \text{ is odd}],\end{aligned}$$

which simplifies to the desired result. (Add 1 or 2 for the leaf nodes as in the previous theorem.) □

Corollary 5.4 *With a non-interleaved variable order, the size of the largest BDD for an intermediate reachable set (any R_i in Algorithm 5.1) is $(c_n/2 + 1)(n^2 + n) + 2$.*

Proof: The BDD-size formula in Theorem 5.3 defines a downward opening parabola as a function of i. Differentiating with respect to i, we find the derivative is positive at $i = n$ and negative at $i = n + 1$. Evaluating at these two points, we find that the largest BDD is the desired result at iteration $i = n$, with the BDD at the next iteration one node smaller. □

(a)

	0	1	2	3	4
1	vr/a				
2	vr/a	vr/a	vr/a		
3	vr/a	vr/a	vr/a	vr/a	vr/a
4	vr/a	vr/a	vr/a	vr/a	vr/a

(b)

	0	1	2	3	4
1	vr/a				
2	vr/a	vr/a	vr/a		
3		vr/a	vr/a	vr/a	vr/a
4				vr/a	vr/a

(c)

	0	1	2	3	4
1	vr/				
2	vr/	vr/a	vr/a		
3		vr/	vr/	vr/a	v /a
4				vr/	v /

(d)

		0	1	2	3	4
1		vr/				
2		vr/	vr/a	vr/a		
3	v/a		vr/	vr/	vr/a	v /a
4	v/a				vr/	v /

		0	1	2	3	4	5
1		vr/					
2	/a	vr/	vr/a	vr/a			
3	v/a			vr/	vr/a	vr/a	
4	v/a					vr/	v /

Figure 5.3: These diagrams show how we count the BDD nodes. All nodes in a row r deal with the rth network array element. Each column corresponds to how many events the BDD has counted so far. The letter v indicates that a square contains a node to check the valid bit; the letter r, a node to check the req/ack bit; and the letter a, the c_n BDD nodes needed to check the address. The slash divides the upper and lower parts of each square. Part (a) shows that we don't count the nodes from the unreachable squares in the upper-right corner. Part (b) shows that we don't count the nodes in the lower-left corner, since the event count doesn't matter at that point so the nodes collapse into a single column that we count separately. Part (c) shows that there are no address check nodes on the left edge, no req/ack nodes on the right edge, and no address check nodes on the bottom edge of the counted region. Part (d) shows the extra chain of BDD nodes that handles the case when the BDD no longer needs to count events. This chain is $\lfloor i/2 \rfloor$ squares long, plus an extra set of address check nodes if i is odd. (In the bottom figure, consider the square at row 2, column 0. If the valid bit is on, and the req/ack bit indicates a request, the BDD must branch to an address check in row 2, but it must also stop counting events, so there must be the extra address check nodes in row 2 in the separate column.)

The actual runs confirm these theoretical results. I used 4-bit integers and BDDs with complement edges, so the BDD size bounds from Theorem 5.2 and Corollary 5.4 are $5n+1$ and $3(n^2+n)+1$. The difference between linear or quadratic and exponential size is quite striking. Interestingly, although Algorithm 5.1 appears to be trading off time for space (since it's processing each deterministic choice separately and the additional check for Theorem 5.1 must be performed at each iteration), it actually runs faster, since the BDDs in the conventional approach are so large and the time for a BDD operation is a function of the size of the BDD. The results are summarized in the Table 5.1 and in Figs. 5.4 and 5.5.

Just as BDDs are no panacea, functionally dependent variables are no panacea, either. There are many other reasons why a BDD might become excessively large. On the other hand, functionally dependent variables are a common cause of BDD-size blowup, and the techniques presented here pay off dramatically, promising to extend the range of applicability of BDD-based verification. Although exploiting functional dependencies is not a cure-all, it is clearly a necessary step.

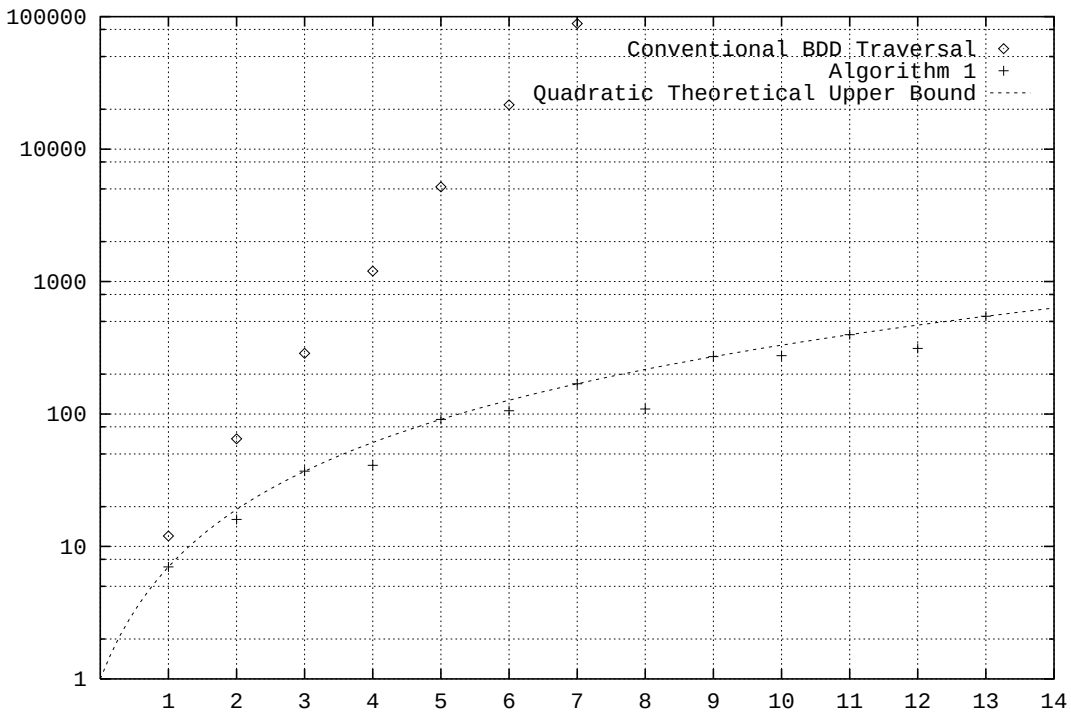


Figure 5.4: Largest Intermediate Reachable BDD Size (semi-log scale). "Conventional BDD Traversal" is the normal forward traversal without taking advantage of functional dependencies and clearly shows exponential growth. The theoretical bound is based on Corollary 5.4, using $c_n \leq 4$.

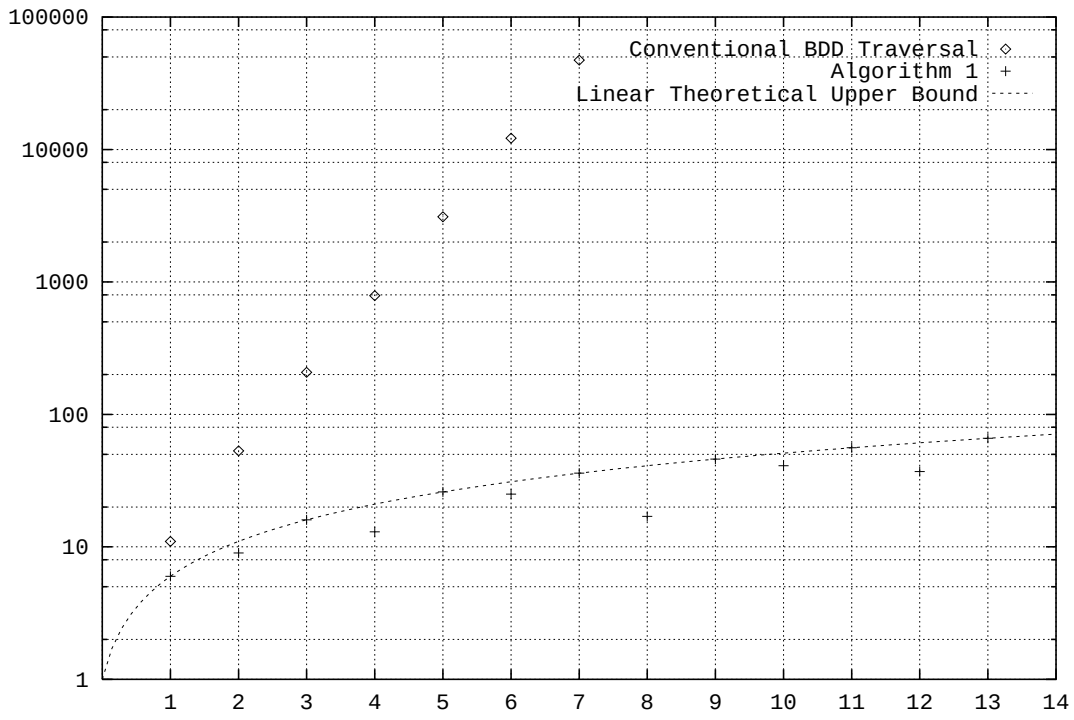


Figure 5.5: Final Reachable State BDD Size (semi-log scale). Again, the conventional approach clearly shows exponential growth. The theoretical bound is based on Theorem 5.2.

n	Method	Iterations	Time (min:sec)	Memory (KB)	Max BDD	Final BDD
1	Fwd	3	0:00.36	624	12	11
	Bkwd	1	0:00.38	636	11	11
	FDV	3	0:00.30	632	7	6
2	Fwd	5	0:00.67	816	65	53
	Bkwd	1	0:00.61	796	56	56
	FDV	5	0:00.79	776	16	9
3	Fwd	7	0:01.67	1188	287	208
	Bkwd	1	0:01.28	1072	236	236
	FDV	7	0:02.58	988	37	16
4	Fwd	9	0:04.73	1556	1198	788
	Bkwd	1	0:03.16	1316	994	994
	FDV	9	0:07.72	1320	41	13
5	Fwd	11	0:16.82	3904	5188	3104
	Bkwd	1	0:08.62	2404	3922	3922
	FDV	11	0:19.94	1556	91	26
6	Fwd	13	2:06.49	10644	21579	12177
	Bkwd	1	0:29.52	4636	15550	15550
	FDV	13	0:45.04	2416	106	25
7	Fwd	15	8:39.04	44800	88647	47267
	Bkwd	1	2:19.77	14848	61861	61861
	FDV	15	2:04.36	3260	169	36
8	Fwd			Space Out		
	Bkwd	1	23:47.25	47480	246829	246829
	FDV	17	3:56.32	4520	109	17
9	Fwd			Space Out		
	Bkwd			Space Out		
	FDV	19	7:16.76	6028	271	46
10	Fwd			Space Out		
	Bkwd			Space Out		
	FDV	21	12:50.99	8372	276	41

(continued on next page)

Table 5.1: Functional Dependency Results: n indicates the number of clients. The Method column indicates which algorithm was used: "Fwd" and "Bkwd" are the standard BDD-based forward and backward traversals described in Chapter 1, "FDV" is the forward traversal taking advantage of functional dependencies. All three methods used the efficient image computation procedure described in Chapter 3. Iterations is the number of iterations to convergence. Time is total execution time as reported by the UNIX shell. "Time Out" indicates execution time exceeded one hour. Memory is the total amount of memory used as reported by the UNIX shell. "Space Out" indicates total memory usage exceeded 60MB. Max BDD gives the size in BDD nodes of the largest R_i or G_i encountered during verification. Final BDD gives the size of the BDD at convergence. The conventional BDD-based approaches cannot handle the larger instances of this example. All results are from a Sun SPARCstation 2.

(continuation of Table 5.1)

n	Method	Iterations	Time (min:sec)	Memory (KB)	Max BDD	Final BDD
11	Fwd			Space Out		
	Bkwd			Space Out		
	FDV	23	20:50.98	10516	397	56
12	Fwd			Space Out		
	Bkwd			Space Out		
	FDV	25	34:16.82	14124	313	37
13	Fwd			Space Out		
	Bkwd			Space Out		
	FDV	27	53:13.27	19788	547	66
14	Fwd			Space Out		
	Bkwd			Space Out		
	FDV		Time Out			
15	Fwd			Space Out		
	Bkwd			Space Out		
	FDV		Time Out			
16	Fwd			Space Out		
	Bkwd			Space Out		
	FDV		Time Out			

Chapter 6

Implicitly Conjoined BDDs

Chapter Overview[1]

Like Chapter 5, this chapter presents an enhancement to a standard BDD-based verification algorithm to avoid an empirically common cause of BDD-size blow-up. The basic idea is to represent a set of states not by a single BDD, but rather by a list of BDDs whose conjunction represents the set of states. The chapter motivates the rationale behind implicitly conjoined BDDs, details the changes to the standard backward traversal algorithm required by the new representation, and demonstrates the effectiveness of the technique on some simple examples that illustrate commonly-occurring causes of BDD-size blow-up.

6.1 Introduction

In Chapter 4, we saw that the BDD representing a set of states of a system is large for all variable orderings if the BDD must encode numerous relationships among the state variables of the system. The functionally dependent variables technique in Chapter 5 attempts to avoid this situation by eliminating variables (and therefore relationships between variables). The implicitly conjoined BDDs technique in this

[1]This chapter is based on material first published in the *Fifth International Conference on Computer-Aided Verification*, 1993 [42] and the *31st Design Automation Conference*, 1994 [45].

chapter attempts to avoid this situation by representing a set of states by a list of BDDs, rather than a single BDD, so that each BDD ideally need only encode a few relationships among a few state variables.

The intuition behind implicitly conjoined BDDs is straightforward. When modeling at a high-level, we typically view the system as a collection of components c_i. Correct operation of the system requires some local correctness property p_i to hold of each component c_i. Commonly, however, correct operation also requires that for many pairs of components c_i and c_j, some consistency property p_{ij} hold between them. For example, recall the network example from Chapter 5 (p. 61). The network has a local correctness property that any valid message must have a legal return address. Between each processor and the network must hold the consistency property that the processor's count is equal to the number of messages relating to that processor. If the components of the system are reasonably small, then the BDD for each local correctness property and each consistency property is small by itself. To verify correct operation of the system, however, the standard BDD-based verification algorithms must essentially build a BDD for the conjunction of all of these properties. This BDD will encode numerous relationships among numerous state variables and will therefore be very large. How can we verify without building the huge BDD for this conjunction?

Suppose that instead of using a single BDD to represent a set of states, we used a list of BDDs, where the list represents the conjunction of the BDDs in the list. In other words, the BDDs in the list are implicitly assumed to be conjoined, and we never explicitly build the BDD for the conjunction. If we could modify a standard BDD-based verification algorithm to use these implicitly conjoined lists of BDDs, we could possibly avoid the BDD-size blow-up described above. Let us consider how to perform such a modification, and then we will see from examples how well the technique works.

6.2 Theoretical Basis

An important property of implicitly conjoined BDDs is that the BackImage operator (p. 18) distributes over conjunction:

Theorem 6.1

$$BackImage(\delta, Y \wedge Z) = BackImage(\delta, Y) \wedge BackImage(\delta, Z).$$

Proof:

$$\begin{aligned}
\text{BackImage}(\delta, Y \wedge Z) &= \forall v[\delta(u,v) \Rightarrow (Y(v) \wedge Z(v))] \\
&= \forall v[(\delta(u,v) \Rightarrow Y(v)) \wedge (\delta(u,v) \Rightarrow Z(v))] \\
&= \forall v[\delta(u,v) \Rightarrow Y(v)] \wedge \forall v[\delta(u,v) \Rightarrow Z(v)] \\
&= \text{BackImage}(\delta, Y) \wedge \text{BackImage}(\delta, Z). \quad \square
\end{aligned}$$

Thus, we can compute the BackImage of an implicitly conjoined list of BDDs by taking the BackImage of each BDD in the list. This theorem suggests that implicitly conjoined BDDs may work well with the backward traversal algorithm. Let us reexamine the backward traversal to see how to implement it with implicitly conjoined BDDs.

Figure 6.1 shows the backward traversal and counterexample algorithms, with all points that manipulate sets of states marked. Let us consider these points one-by-one.

Point A is easy. Suppose G_i is represented by an implicitly conjoined list of small BDDs. Let $G_i[j]$ denote the jth BDD in the list, so $G_i = G_i[1] \wedge \cdots \wedge G_i[n]$. Then $S \not\Rightarrow G_i$ if and only if there exists a j such that $S \not\Rightarrow G_i[j]$. We can therefore check for a violation of the verification condition by checking each conjunct independently from the other conjuncts.

Point B consists of two operations — the BackImage and the conjunction. The BackImage is straightforward, thanks to Theorem 6.1. We can compute the BackImage of each conjunct separately. For convenience, let $B_i[j]$ denote BackImage$(\delta, G_i[j])$, with B_i denoting the entire implicitly conjoined list of BDDs $B_i[1] \wedge \cdots \wedge B_i[n]$.

Backward Traversal:
Let $G_0 := G$ and $i := 0$.
Repeat
Loop Invariant: For all i, set G_i is the set of all states x such that any path from x to a state violating the property must have length greater than i.
If $S \not\Rightarrow G_i$, then print counterexample and exit. (Point A)
$G_{i+1} := G_0 \wedge \text{BackImage}(\delta, G_i)$. (Point B)
$i := i + 1$.
Until $G_i = G_{i-1}$. (Point C) *Termination guaranteed by monotonicity.*
Verification succeeds. Exit.

Counterexample:
Let $P := S$.
Loop
Loop Invariant: There exists a path from P of length i leading to a property violation, but no path of length $i - 1$.
Choose a state x that satisfies $P \wedge \neg G_i$. (Point D)
Print x.
If $i = 0$, then x is a violating state. Exit.
$P := \text{Image}(\delta, \{x\})$.
$i := i - 1$.
End Loop

Figure 6.1: Backward Traversal Revisited: This is the same backward traversal algorithm presented in Chapter 1, along with the counterexample algorithm. Points where the algorithm must be modified to handle implicitly conjoined BDDs are labeled. Points A and D are easy, as the operation in question distributes over conjunction. Points B and C require new heuristics.

The interesting part of Point B is the conjunction of G_0 with B_i, because there are myriad possibilities, all of them mathematically correct, but with different efficiency implications. On one extreme, for example, we could take the implicitly conjoined lists of BDDs representing G_0 and B_i and attempt to build the single BDD representing the conjunction of all of these small BDDs. Although this approach maintains the correctness of the backward traversal algorithm, it defeats our intention of using implicitly conjoined lists of BDDs to avoid BDD-size blowup. On the other extreme, we could simply concatenate the list of conjuncts representing G_0 with the list representing B_i, again maintaining the correctness of the algorithm, but in this case never building the BDD for any of the implied conjunctions. This choice has the undesirable property that the length of the list of conjuncts will grow on each iteration. Another option keeps this length constant on all iterations by setting G_{i+1} to be the implied conjunction $G_{i+1}[1] \wedge \cdots \wedge G_{i+1}[n]$ where each $G_{i+1}[j]$ is the BDD for $G_i[j] \wedge B_i[j]$ (explicitly evaluated as a BDD AND), essentially verifying each conjunct $G[j]$ of the verification condition G completely independently of the other conjuncts. This choice has the advantage that it trivializes proving termination for Point C but the disadvantage that each conjunct ignores information from the other conjuncts.

A key insight at this point exposes further options at Point B. The correctness of the backward traversal algorithm depends only on the value of the implied conjunction of an entire list of conjuncts, not on the specifics of each conjunct. Therefore, if one conjunct is false for some truth assignment x, the whole conjunction is false at x, and the values of all the other conjuncts at x don't matter at all. Accordingly, we can simplify the BDD representation for each conjunct based on the other conjuncts. For example, to simplify conjunct $G_i[j]$ by conjunct $G_i[k]$ $(j \neq k)$, we can replace $G_i[j]$ by any other Boolean function $\hat{G}_i[j]$ so long as $G_i[j] = \hat{G}_i[j]$ wherever $G_i[k]$ is true, and hopefully the BDD for $\hat{G}_i[j]$ is smaller than the BDD for $G_i[j]$. ($\neg G_i[k]$ can be viewed as a don't-care set to simplify Boolean function $G_i[j]$.) These simplifications have no effect on the correctness of the backward traversal algorithm, because they do not change the value of the (implied) conjunctions, but they can have a significant effect on the efficiency of the backward traversal algorithm. Obviously, there are many possible policies for applying these simplifications.

Testing for convergence at Point C could conceivably be problematic, as implicitly conjoined lists of BDDs, unlike a single BDD, are not a canonical representation for Boolean functions, complicating testing for equality. Even proving that we can detect convergence could be difficult, depending on how complex our choice of simplification policy at Point B is, since the don't-care simplification can introduce non-monotonicity in the sequence $G_0[j], G_1[j], \ldots$, for some values of j. (The entire conjunction, of course, forms a monotonic sequence $G_0, G_1, \ldots$ that converges, but we cannot build the BDD for the entire conjunction.)

Point D turns out to be straightforward. We know that there exists a path of length i from P that violates the property being verified, but no path of any length less than i. Therefore, $P \wedge \neg G_i$ is non-empty, implying that there exists a j such that $P \wedge \neg G_i[j]$ is non-empty and that any $x \in P \wedge \neg G_i[j]$ is also in $P \wedge \neg G_i$. Again, we can perform this operation on each conjunct independently.

The only open issues, therefore, are deciding which conjunctions to evaluate at Point B and testing for convergence at Point C. For these questions, we turn to heuristics.

6.3 Heuristics

I started with very simple heuristics and quickly found something that worked reasonably well on my test cases. I then developed more sophisticated heuristics to address the problems with the simple ones. Descriptions of the heuristics follow. Results are in Section 6.4.

6.3.1 A Simple Approach

A very simple answer to these two issues gives reasonable results. Compute $G_{i+1}[j] = G_i[j] \wedge B_i[j]$ as described above, but then simplify each $G_{i+1}[j]$ by all $G_{i+1}[k]$ for $k < j$. If G_0 is an initial implicit conjunction of properties we wish to verify, then this approach resembles verifying each property independently of the others, except that the don't-care simplification uses information from the earlier properties to simplify

the verification of the later ones. The BDD simplification operator I use is Coudert, Berthet, and Madre's Restrict operator (introduced in [22], named in [23]), which takes two BDDs as input and produces a BDD that agrees with the first input BDD wherever the second input BDD evaluates to true. (The second BDD specifies a care set to simplify the first BDD.) Another choice is the Constrain [21] or generalized cofactor [85] operator.[2]

This simple evaluation policy tends to maintain correspondence between the jth entry of the list on different iterations, i.e., ignoring the effect of the simplification operator, for a given value of j, the sequence of BDDs $G_0[j], G_1[j], \ldots, G_i[j]$ is just the sequence of BDDs required to verify the jth property in G_0. Therefore, a simple component-wise convergence test, checking that $G_i[j] = G_{i-1}[j]$ for all j, seems reasonable. This condition is obviously sufficient for convergence, so the verifier will never terminate prematurely. On the other hand, as mentioned earlier, a simplification operator could conceivably introduce enough non-monotonicity to prevent this test from detecting convergence. In practice, this simple test has consistently performed correctly, even with very aggressive BDD-simplification policies.

This simple approach has two main drawbacks. First, it makes no effort to find good BDD combinations in the implicitly conjoined list. In particular, note that if G_0 is just a single BDD, this approach degenerates into the conventional backward traversal with BDDs. Addressing this problem exacerbates the second drawback: That termination is not proven is already disquieting; if our algorithm further modifies the lists of BDDs, the possibility of never detecting termination becomes too great. Thus, we look for more sophisticated heuristics to address these issues.

6.3.2 Greedy Evaluation

On each iteration, the backward traversal algorithm computes an implicitly conjoined list of BDDs for $G_{i+1} = G_0 \wedge \text{BackImage}(\delta, G_i)$. We seek to find an equivalent list of BDDs that is smaller overall. More abstractly, given function X expressed as the

[2] Shiple *et al.* [82] survey several heuristics for BDD simplification. Of the heuristics studied, Restrict is the fastest and generally produces very good results.

implicit conjunction of BDDs $X_1 \wedge \cdots \wedge X_n$, we want to find an implicit conjunction with smaller overall size $Y = Y_1 \wedge \cdots \wedge Y_m$, such that $X = Y$.

We will again use the Restrict operator for BDD simplification. While this operator doesn't always reduce the size of the BDD it is applied to,[3] it seems generally effective, so we first simplify each BDD X_i by every other BDD X_j that is smaller than it. (Simplifying a small BDD by a large BDD generally does little good.) The remaining problem is simply to decide which conjunctions to evaluate to minimize the total size of the implicitly conjoined list.

At first glance, this problem appears an ideal combinatorial optimization problem. For every subset of the BDDs in the list, we can replace that subset by the single BDD that's the conjunction of all the BDDs in the subset. Thus, we arrive at a set-covering problem:

> Let X be a set of n conjuncts $X = \{X_1, \ldots, X_n\}$. For every subset $s \subseteq X$, define the cost of that subset $c(s)$ to be the size of the (single) BDD that represents the conjunction of all the members of s: $c(s) = \text{BDDSize}\left(\bigwedge_{X_i \in s} X_i\right)$. Find the minimum cost set S of subsets that covers all the conjuncts in X, i.e., find the S that minimizes $\sum_{s \in S} c(s)$ subject to $\forall i \exists s \in S[X_i \in s]$.

Unfortunately, this approach yields an instance of Minimum Weight Cover, and Minimum Weight Cover is clearly NP-hard by reduction from Minimum Cover [35], even if restricted to subsets of three or fewer conjuncts. (The constraints on the cost function imposed by BDD properties, however, might make this problem easier than Minimum Weight Cover in general.) If we restrict ourselves to pairwise subsets only, we can solve the problem in polynomial time:

Theorem 6.2 *Finding the min cost pairwise cover is polynomial time.*

[3] The belief that Restrict never increases the size of the BDD to which it is applied is a common misconception. For a counterexample, consider the family of functions $f_n = x_1 \oplus \cdots \oplus x_n$ and $c_n = x_1 \vee \cdots \vee x_n$. Variable order is irrelevant because of symmetry. For plain BDDs, f_n has $2n + 1$ nodes (including the terminals), whereas Restrict(f_n, c_n) has $3n - 2$ nodes. For BDDs with complement edges, f_n has $n + 1$ nodes whereas Restrict(f_n, c_n) has $2n - 1$ nodes. The Constrain operator behaves identically on these functions. I would like to thank Jerry Burch for suggesting that this choice of f_n and c_n would likely produce the desired counterexample.

Proof: Draw a complete graph with a vertex for each conjunct. Label each edge with the size of the BDD for the conjunction of the BDDs on the two incident vertices. Next, make a copy of each vertex. Connect each original vertex to its copy; label that edge with the minimum of the size of the BDD at that vertex and the labels of all other incident edges. (This edge indicates the cheapest way to include this conjunct ignoring all the other conjuncts.) Connect all the copy vertices to each other with weight 0 edges.[4] Minimum weighted matching, which is polynomial time (e.g. [73]), on this graph gives the optimum cover. □

Even this result is of limited practical value because in reality, for efficient BDD implementations, BDD sizes do not add, since all BDDs in the system can share nodes with each other [5]. Using a complex "optimum" algorithm for a rough approximation to a problem makes little sense. Thus, we turn to a greedy heuristic.

The heuristic I propose is fairly simple, yet accounts for some degree of node sharing among different BDDs. The intuition is to find the pair of BDDs for which evaluating the conjunction gives the greatest savings over not evaluating the conjunction. We replace the pair of BDDs with the single BDD for the conjunction and repeat the process. The process terminates when the best conjunction to evaluate doesn't give sufficient savings. The algorithm is given in Figure 6.2.

6.3.3 Exact Termination Testing

Deciding termination in the verification algorithms requires testing whether the iteration has converged — is $R_i = R_{i+1}$ or $G_i = G_{i+1}$? (Actually, checking implication suffices since these sequences are monotonic.) The simple termination test proposed earlier relied on the structure of the simple evaluation policy to provide a good chance of operating correctly. Given that the evaluation and simplification technique proposed in the preceding section can extensively modify an implicitly conjoined list of BDDs, a method to compare two arbitrary implicitly conjoined lists of BDDs seems necessary for reliable termination testing. Furthermore, verification, by nature, should favor a method that is guaranteed correct, but possibly slow, over a method that is

[4] I would like to thank Eric Torng for this construction.

Conjunction Evaluation:
Let GrowThreshold = 1.0.
Build a table P of all pairwise conjunctions: $P_{ij} := X_i \wedge X_j$.
Loop
    Find the i, j (with $i \neq j$) that minimizes the ratio:
        $r = \text{BDDSize}(P_{ij})/\text{BDDSize}(X_i, X_j)$
        Note: BDDSize(X_i, X_j) takes node-sharing into account.
    If $r_{\min} >$ GrowThreshold, then exit.
    Replace X_i and X_j with P_{ij}.
    Update P to reflect the modified conjunct list.
EndLoop

Figure 6.2: Greedy Evaluation Algorithm: This algorithm is a greedy algorithm to find a good set of conjunctions to evaluate in an implicitly conjoined list of BDDs. I have set the GrowThreshold to 1.0 with satisfactory results. Additional tuning could improve results further: a smaller threshold holds BDD size down, but can get caught in a local minimum, whereas any threshold greater than 1 could theoretically allow us to build exponentially-sized BDDs. Note that we can abort the computation of P_{ij} as soon as its size exceeds GrowThreshold times BDDSize(X_i, X_j).

fast, but possibly wrong. Thus, we look for an exact test of implication for arbitrary implicitly conjoined BDDs.

Suppose we have two implicitly conjoined lists of BDDs $X = \{X_1 \wedge \cdots \wedge X_n\}$ and $Y = \{Y_1 \wedge \cdots \wedge Y_m\}$. Our task is to determine whether or not $X \Rightarrow Y$, without building the BDDs for X or for Y, since those BDDs are presumably too big to build. Decomposing the problem is the most intuitive way to explain my solution. The original implementation reflected this decomposition [45], but the current implementation described below is more efficient. Note that

$$\begin{aligned} X \Rightarrow Y &= X \Rightarrow (Y_1 \wedge \cdots \wedge Y_m) \\ &= (X \Rightarrow Y_1) \wedge \cdots \wedge (X \Rightarrow Y_m), \end{aligned}$$

so we can check each case separately. All the cases are identical, so, for brevity, let's consider only the $X \Rightarrow Y_1$ case. Checking whether $X \Rightarrow Y_1$ is true is equivalent to checking whether $\neg X \vee Y_1$ is a tautology, which is actually checking whether $\neg X_1 \vee \cdots \vee \neg X_n \vee Y_1$ is a tautology. Thus, we have reduced the problem of checking whether one implicitly conjoined list of BDDs implies another to the problem of checking whether the disjunction of a list of BDDs is a tautology. We can't simply build the BDD for this disjunction, as that would still blow up, so we further decompose the problem into smaller, more manageable pieces. The strategy here is to look for easy special cases first (e.g., any disjunct is just True, two disjuncts are complements, etc.) and if that fails, to perform a Shannon expansion on the implicit disjunction.

Although this decomposition is intuitively appealing, it suffers from two major inefficiencies. First, finding that $X \not\Rightarrow Y_i$ might be easy for some i, but the decomposed version of the algorithm may spend considerable time working on other cases first. Second, each instance of tautology checking of the list of BDDs $\neg X_1 \vee \cdots \vee \neg X_n \vee Y_i$ shares all of the same disjuncts except for the last one, so the cost of the Shannon decomposition of the X should not be repeated m times for the conjuncts of Y. Accordingly, although the current implementation follows the intuition from the decomposed problem, it actually attacks the problem of $X \stackrel{?}{\Rightarrow} Y$ directly. Specifically, it performs the following steps in sequence:

1. If any BDD in the X list is the constant False, return True.
   If any BDD in the X list is the constant True, discard it.
   If any BDD in the Y list is the constant False, replace Y by the constant False.
   If any BDD in the Y list is the constant True, discard it.
   If the Y list is empty, return True.
   If the Y list is non-empty and the X list is empty, return False.

2. If any two BDDs in the X list are complements, return True. (Recall that negation is fast in efficient BDD implementations, so this is a cheap test.)

3. If any two BDDs in the Y list are complements, replace Y by the constant False.

4. If any two BDDs in X are identical, discard one. Similarly for Y.

5. Simplify each BDD in X by every other BDD in X. Simplify each BDD in Y by every other BDD in Y and by every other BDD in X. (After a BDD is simplified, the new, simplified BDD is used subsequently to simplify the other BDDs.) This step reduces BDD sizes in the list, and also handles several special cases (see below).

6. Repeat Step 1 to catch any special cases resulting from the simplification.

7. If all else fails, choose a BDD variable from a BDD in the X list, perform a Shannon expansion, and check tautology recursively on both cofactors. (The positive and negative cofactors will each be an implication between implicitly conjoined lists of BDDs.) For simplicity, the current implementation selects the top BDD variable of the first BDD in the X list as the variable to cofactor on.

These steps are quite straightforward; obviously, many other variations are possible. The following theorems explain how Step 5 handles several special cases:

Theorem 6.3 *For any two Boolean functions a and b, with b not equal to the constant False:*

$$\textit{Restrict}(a, b) \textit{ returns the constant True } \textit{iff}$$

Constrain(a, b) *returns the constant True iff*
$(b \Rightarrow a)$ *is the constant True (tautology).*

Theorem 6.4 *For any two Boolean functions a and b, with b not equal to the constant False:*

Restrict(a, b) *returns the constant False iff*
Constrain(a, b) *returns the constant False iff*
$(a \wedge b)$ *is the constant False (contradiction).*

Proof of Theorem 6.3: Let us first consider Restrict. The proof is by induction on the number of variables in a and b. The base cases, when a is either the constant True or the constant False or when b is the constant True, are easy to verify. The inductive step relies on the recursive definition of Restrict, which defines Restrict(f, c) in terms of the Shannon cofactors f_x, $f_{\bar{x}}$, c_x, and $c_{\bar{x}}$, where x is a BDD variable in f or in c: (The exact definition of x is irrelevant here.)

$$\text{Restrict}(f, c) = \begin{cases} \text{Restrict}(f, c_x \vee c_{\bar{x}}) & \text{if } f_x = f_{\bar{x}} \\ \text{Restrict}(f_x, c_x) & \text{if } c_{\bar{x}} = \text{False} \\ \text{Restrict}(f_{\bar{x}}, c_{\bar{x}}) & \text{if } c_x = \text{False} \\ (x \wedge \text{Restrict}(f_x, c_x)) \vee (\bar{x} \wedge \text{Restrict}(f_{\bar{x}}, c_{\bar{x}})) & \text{otherwise} \end{cases}$$

Case 1: If a is independent of x, then Restrict(a, b) is defined to be Restrict$(a, b_x \vee b_{\bar{x}})$, which is the constant True iff $(b_x \vee b_{\bar{x}} \Rightarrow a)$ is a tautology (by the inductive hypothesis). The expression $b_x \vee b_{\bar{x}} \Rightarrow a$ is equivalent to the expression $(b_x \Rightarrow a_x) \wedge (b_{\bar{x}} \Rightarrow a_{\bar{x}})$ (since $a = a_x = a_{\bar{x}}$ in this case), which is a tautology iff $(b \Rightarrow a)$.

Case 2: If $b_{\bar{x}}$ is the constant False, then Restrict(a, b) is defined to be Restrict(a_x, b_x), which is the constant True iff $b_x \Rightarrow a_x$ (by the inductive hypothesis), which is equivalent to $(b_x \Rightarrow a_x) \wedge (b_{\bar{x}} \Rightarrow a_{\bar{x}})$ (since $b_{\bar{x}}$ is the constant False), which is a tautology iff $(b \Rightarrow a)$ is.

Case 3: The case where b_x is the constant False is very similar to the preceding case.

Case 4: Otherwise, Restrict(a, b) equals $(x \wedge \text{Restrict}(a_x, b_x)) \vee (\bar{x} \wedge \text{Restrict}(a_{\bar{x}}, b_{\bar{x}}))$, which is identically equal to the constant True iff Restrict(a_x, b_x) is the constant True, and Restrict$(a_{\bar{x}}, b_{\bar{x}})$ is the constant True. By the inductive hypothesis, this occurs iff $(b_x \Rightarrow a_x)$ is a tautology, and $(b_{\bar{x}} \Rightarrow a_{\bar{x}})$ is a tautology, which occurs iff $(b \Rightarrow a)$ is a tautology.

The operator Constrain is identical to Restrict, except Case 1 is deleted. Thus, the proof for Constrain is identical to the preceding argument with Case 1 omitted. □

Proof of Theorem 6.4: This proof is structured identically to the preceding one. Again, let us first consider Restrict. The base cases, when a is either the constant True or the constant False or when b is the constant True, are easy to verify. The inductive step again relies on the recursive definition of Restrict:

Case 1: If a is independent of x, then Restrict(a, b) is defined to be Restrict$(a, b_x \vee b_{\bar{x}})$, which is the constant False iff $(a \wedge (b_x \vee b_{\bar{x}}))$ is a contradiction (by the inductive hypothesis). The expression $(a \wedge (b_x \vee b_{\bar{x}}))$ is equivalent to the expression $(a_x \wedge b_x) \vee (a_{\bar{x}} \wedge b_{\bar{x}})$ (since $a = a_x = a_{\bar{x}}$ in this case), which is a contradiction iff $(a \wedge b)$ is a contradiction.

Case 2: If $b_{\bar{x}}$ is the constant False, then Restrict(a, b) is defined to be Restrict(a_x, b_x), which is the constant False iff $(a_x \wedge b_x)$ is a contradiction (by the inductive hypothesis). The expression $(a_x \wedge b_x)$ is equal to $(a_x \wedge b_x) \vee (a_{\bar{x}} \wedge b_{\bar{x}})$ (since $b_{\bar{x}}$ is the constant False), which is a contradiction iff $(a \wedge b)$ is.

Case 3: The case where b_x is the constant False is very similar to the preceding case.

Case 4: Otherwise, Restrict(a, b) equals $(x \wedge \text{Restrict}(a_x, b_x)) \vee (\bar{x} \wedge \text{Restrict}(a_{\bar{x}}, b_{\bar{x}}))$, which is identically equal to the constant False iff Restrict(a_x, b_x) is the constant False, and Restrict$(a_{\bar{x}}, b_{\bar{x}})$ is the constant False. By the inductive hypothesis, this occurs iff $(a_x \wedge b_x)$ is a contradiction, and $(a_{\bar{x}} \wedge b_{\bar{x}})$ is a contradiction, which occurs iff $(a \wedge b)$ is a contradiction.

As in the preceding proof, apply this argument to the Constrain operator by deleting Case 1. □

Therefore, by using Restrict or Constrain in Step 5, followed by a clean-up step, we efficiently handle the special cases where a conjunct in the antecedent implies another conjunct in the antecedent or consequent (delete the larger conjunct), where a conjunct in the consequent implies another conjunct in the consequent (delete the larger conjunct), where two conjuncts in the antecedent contradict each other (return true for the implication), and where a conjunct in the consequent contradicts another conjunct (return false unless the antecedent is also a contradiction).

6.4 Experimental Results

To demonstrate the practical usefulness of these ideas, let's apply the above method to some simple examples. These examples share two important properties. First, although they are simple, they are intractable by the standard BDD-based verification algorithms. More importantly, these examples illustrate core characteristics of numerous, real verification problems. If we can't handle these simple examples, there's an enormous range of important, practical verification problems that we will never be able to touch.

6.4.1 A Typed FIFO Buffer

In most high-level BDD-based verifiers, integer types are encoded as bit vectors [83, 53, 44]. The most natural encoding scheme is simply to use the binary representation of the integer. In many instances, however, especially when performing high-level verification, the number of possible values is not exactly a power of 2. For example, in a communication protocol, a variable might indicate which one of 17 different message types is being sent, or another variable might contain parity-encoded data. Furthermore, if we use complex data structures, a variable might contain a compound, non-numeric value like a processor instruction or a network message, and not all possible values will be legal. Therefore, all legal states of the system will obey a set of properties like "$x < 17$" or "y has odd parity" or "z contains a legal instruction". I call these properties "type invariants".

By itself, each of these type invariants typically has a small BDD. The system as a whole, however, must satisfy all of the type invariants, leading to a BDD for the conjunction of all of them that may be very large, depending on the variable-ordering used. Consider, for example, a FIFO buffer of d words, each of which is k bits wide. (Such a buffer occurs frequently in verification models, both to model actual queues in a system and also to time-delay computed results to check an implementation against a differently timed specification.) Each word of the buffer must satisfy some property P that depends on most of the k bits. (A typical example is a subrange constraint.) Let w be the width of any cut of the BDD for P; for example, if P says that the word must be less than a constant, then $w = 2$. Furthermore, suppose we have interleaved the bits for the FIFO buffer, putting the high-order bit of all words first, followed by the next most significant bit of all words, etc. This variable ordering is generally necessary in datapaths to minimize BDD size for comparisons and arithmetic [83, 53, 51]. In this case, the BDD for the conjunction of all the type invariants is of size $O(kw^d)$, since in the BDD, after each bitslice, we must encode all w^d possible intermediate states of each type invariant, and there are k bitslices in all. If we attempt to perform a forward traversal on this system, we will be forced to build this exponentially-sized BDD, as any reachable state must satisfy all type invariants. Performing a conventional backward traversal requires building this exponentially-sized BDD just to start the verification process. Leaving the conjunction implicit, however, keeps the various type invariants separate throughout the verification process, thereby avoiding the BDD-size blowup.

A different approach to this same problem is to change the encoding scheme so that unused binary combinations become don't-cares by assigning multiple encodings to some values [83, 53]. This encoding eliminates the type invariants altogether, as every possible bit pattern encodes a legal value. This approach seems promising for small, low-level systems (for which it was designed) where we might be performing state encoding for a small controller. For larger systems and high-level verification, however, this approach has two major drawbacks. First, by complicating the encoding scheme, designing general algorithms for high-level operations like arithmetic,

array-indexing, or comparison becomes impossible. Indeed, when using this don't-care-encoding approach, one must define specific operators (like addition) for each combination of different types. The second major drawback is that *checking* that the type invariants are satisfied becomes impossible, since we've already encoded them out of existence. Again, at the low-level this isn't an issue, but for high-level verification, checking that type properties hold (e.g. a counter doesn't go out-of-range) is essential.

For the concrete example, the buffer is n words deep; each word is 8-bits wide. At each time step, each value in the buffer is copied to the next word, and a new non-deterministic value less than or equal to 128 is added to the end of the buffer. The verification condition is that all entries in the buffer are less than or equal to 128. Table 6.1 summarizes results for this example. Clearly, using implicitly conjoined BDDs greatly reduces both the memory and the time required for the verification, and the more complex heuristics from Sections 6.3.2 and 6.3.3 are only slightly slower than the simple approach from Section 6.3.1.

6.4.2 A Simple Network

Next, let's reconsider the network example from Section 5.4. Note that any functional dependency $y = f(x)$ can also be viewed as an additional property we must verify, which we can add to an implicit conjunction. Accordingly, for this example, the initial implicit conjunction is just the conjunction over all processes that each process's count is equal to the number of messages in the network relating to that process.

Results from this example are in Table 6.2 As we've seen previously, conventional BDD approaches explode on this example. The functional dependency method gives the smallest BDD sizes, but at substantial cost in runtime because of the additional proof obligation at each iteration. The implicit conjunction methods give reasonable BDD sizes and fast execution times. The computational overhead of the more complicated implicit conjunction heuristics is minimal for this example.

n	Method	Iterations	Time (sec)	Memory	Max BDD	(breakdown)
2	Fwd	3	0.34	592K	31	
	Bkwd	1	0.29	624K	31	
	IC	1	0.28	596K	17	(2×9)
	XIC	1	0.29	596K	17	(2×9)
3	Fwd	4	0.62	724K	87	
	Bkwd	1	0.43	652K	87	
	IC	1	0.41	592K	25	(3×9)
	XIC	1	0.41	632K	25	(3×9)
4	Fwd	5	1.30	936K	223	
	Bkwd	1	0.80	776K	223	
	IC	1	0.58	684K	33	(4×9)
	XIC	1	0.60	688K	33	(4×9)
5	Fwd	6	3.69	1024K	543	
	Bkwd	1	1.69	1048K	543	
	IC	1	0.82	748K	41	(5×9)
	XIC	1	0.86	752K	41	(5×9)
6	Fwd	7	9.33	1124K	1279	
	Bkwd	1	3.48	1032K	1279	
	IC	1	1.15	824K	49	(6×9)
	XIC	1	1.20	832K	49	(6×9)

(continued on next page)

Table 6.1: Typed FIFO Buffer Results: The buffer is 8 bits wide. n indicates the depth of the buffer. The Method column indicates which algorithm was used: "Fwd" and "Bkwd" are the standard BDD-based forward and backward traversals described in Chapter 1, "IC" is the implicitly conjoined BDD method with the simple heuristics from Section 6.3.1, "XIC" is with the fancy heuristics from Sections 6.3.2 and 6.3.3. All methods used the efficient image computation procedure described in Chapter 3. Iterations is the number of iterations to convergence. Time is total execution time as reported by the UNIX shell. Memory is the total amount of memory used as reported by the UNIX shell. Max BDD gives the size in BDD nodes of the largest R_i or G_i encountered during verification. For implicit conjunctions, the number reported is the total number of nodes used by all BDDs in the conjunct list. The breakdown shows the sizes of the individual BDDs in the list. For example, "$(2 \times 9, 37)$" indicates a list containing two BDDs with 9 nodes each, and a third BDD with 37 nodes. The number of nodes do not add up because of node sharing. Clearly, using implicitly conjoined BDDs results in substantial savings of time and memory. All results are from the Ever verifier (Appendix A) running on a Sun SPARCstation 2.

(continuation of Table 6.1)

n	Method	Iterations	Time (sec)	Memory	Max BDD	(breakdown)
7	Fwd	8	21.81	1636K	2943	
	Bkwd	1	8.99	1556K	2943	
	IC	1	1.55	860K	57	(7×9)
	XIC	1	1.61	896K	57	(7×9)
8	Fwd	9	54.72	3380K	6655	
	Bkwd	1	21.06	2672K	6655	
	IC	1	1.99	944K	65	(8×9)
	XIC	1	2.07	960K	65	(8×9)
9	Fwd	10	132.22	6428K	14847	
	Bkwd	1	50.77	5240K	14847	
	IC	1	2.52	984K	73	(9×9)
	XIC	1	2.63	1008K	73	(9×9)
10	Fwd	11	313.40	13872K	32767	
	Bkwd	1	115.11	9268K	32767	
	IC	1	3.30	1140K	81	(10×9)
	XIC	1	3.43	1156K	81	(10×9)

6.4.3 A Moving-Average Filter

The next example is a moving-average filter, a common building block in digital signal processing. The filter continuously computes the average of the last n samples seen. For this example, I compare an implementation using a pipelined tree of adders (trades latency for higher throughput) against a specification that computes the average combinationally in a single clock cycle and then buffers the result to match the pipeline latency of the implementation. See Figure 6.3. The verification task is to prove that the outputs of the implementation and of the specification always agree. I will use 8-bit samples and verify filters of depth 4, 8, and 16.

Note that the verification task is not given as a conjunction of simpler properties. The simple heuristics from Section 6.3.1, therefore, degenerate into the conventional BDD backward traversal. Nothing prohibits us, however, from adding more properties to verify, which we can implicitly conjoin with the specified verification property. In this example, not only must the outputs of the implementation and specification agree, but each entry in the FIFO buffer of the specification must equal the average of the corresponding layer of the adder tree in the implementation. If we attempt to build a single BDD for the conjunction of all of these properties, the BDD will

n	Method	Iterations	Time (sec)	Memory	Max BDD	(breakdown)
1	Fwd	3	0.36	624K	12	
	Bkwd	1	0.38	636K	11	
	FDV	3	0.30	632K	7	
	IC	1	0.35	588K	11	(11)
	XIC	1	0.33	632K	11	(11)
2	Fwd	5	0.67	816K	65	
	Bkwd	1	0.61	796K	56	
	FDV	5	0.79	776K	16	
	IC	1	0.62	788K	45	(2×23)
	XIC	1	1.01	796K	45	(2×23)
3	Fwd	7	1.67	1188K	287	
	Bkwd	1	1.28	1072K	236	
	FDV	7	2.58	988K	37	
	IC	1	1.28	1060K	115	(3×39)
	XIC	1	1.29	1072K	115	(3×39)
4	Fwd	9	4.73	1556K	1198	
	Bkwd	1	3.16	1316K	994	
	FDV	9	7.72	1320K	41	
	IC	1	2.71	1304K	245	(4×62)
	XIC	1	2.79	1312K	245	(4×62)
5	Fwd	11	16.82	3904K	5188	
	Bkwd	1	8.62	2404K	3922	
	FDV	11	19.94	1556K	91	
	IC	1	5.43	1848K	436	(5×88)
	XIC	1	5.90	1896K	436	(5×88)

(continued on next page)

Table 6.2: Simple Network Results: n indicates the number of clients. The Method column indicates which algorithm was used: "Fwd" and "Bkwd" are the standard BDD-based forward and backward traversals described in Chapter 1, "FDV" is the functional dependency method described in Chapter 5, "IC" is the implicitly conjoined BDD method with the simple heuristics from Section 6.3.1, "XIC" is with the fancy heuristics from Sections 6.3.2 and 6.3.3. Other columns are as in Table 6.1. "TIME OUT" indicates that verification could not complete in one hour of processor time; "SPACE OUT", that verification could not complete in 60MB of memory. The normal BDD algorithms are unable to handle the larger instances of this example. The functional dependency method produces the smallest BDDs — the BDD blow-up in this example is entirely due to functionally dependent variables — but requires longer run times than the implicitly conjoined BDDs methods, which still produce reasonably-sized BDDs.

(continuation of Table 6.2)

n	Method	Iterations	Time (sec)	Memory	Max BDD	(breakdown)
6	Fwd	13	126.49	10644K	21579	
	Bkwd	1	29.52	4636K	15550	
	FDV	13	45.04	2416K	106	
	IC	1	9.62	2476K	715	(6×120)
	XIC	1	10.25	2592K	715	(6×120)
7	Fwd	15	519.04	44800K	88647	
	Bkwd	1	139.77	14848K	61861	
	FDV	15	124.36	3260K	169	
	IC	1	14.86	3356K	1086	(7×156)
	XIC	1	18.53	4080K	1086	(7×156)
8	Fwd			SPACE OUT		
	Bkwd	1	1427.25	47480K	246829	
	FDV	17	236.32	4520K	109	
	IC	1	22.97	4660K	1585	(8×199)
	XIC	1	27.58	5532K	1585	(8×199)
9	Fwd			SPACE OUT		
	Bkwd			SPACE OUT		
	FDV	19	436.76	6028K	271	
	IC	1	34.44	5968K	2197	(9×245)
	XIC	1	38.49	7464K	2197	(9×245)
10	Fwd			SPACE OUT		
	Bkwd			SPACE OUT		
	FDV	21	770.99	8372K	276	
	IC	1	48.04	8368K	2961	(10×297)
	XIC	1	54.45	8368K	2961	(10×297)
11	Fwd			SPACE OUT		
	Bkwd			SPACE OUT		
	FDV	23	1250.98	10516K	397	
	IC	1	66.73	11752K	3873	(11×353)
	XIC	1	77.20	11996K	3873	(11×353)

(continued on next page)

(continuation of Table 6.2)

n	Method	Iterations	Time (sec)	Memory	Max BDD	(breakdown)
12	Fwd			SPACE OUT		
	Bkwd			SPACE OUT		
	FDV	25	2056.82	14124K	313	
	IC	1	93.03	14728K	4981	(12×416)
	XIC	1	106.10	16020K	4981	(12×416)
13	Fwd			SPACE OUT		
	Bkwd			SPACE OUT		
	FDV	27	3193.27	19788K	547	
	IC	1	130.24	16672K	6254	(13×482)
	XIC	1	142.08	20012K	6254	(13×482)
14	Fwd			SPACE OUT		
	Bkwd			SPACE OUT		
	FDV		TIME OUT			
	IC	1	170.16	22708K	7743	(14×554)
	XIC	1	189.96	24700K	7743	(14×554)
15	Fwd			SPACE OUT		
	Bkwd			SPACE OUT		
	FDV		TIME OUT			
	IC	1	207.92	31400K	9436	(15×630)
	XIC	1	249.48	31612K	9436	(15×630)
16	Fwd			SPACE OUT		
	Bkwd			SPACE OUT		
	FDV		TIME OUT			
	IC	1	265.56	38108K	11345	(16×710)
	XIC	1	348.83	38264K	11345	(16×710)

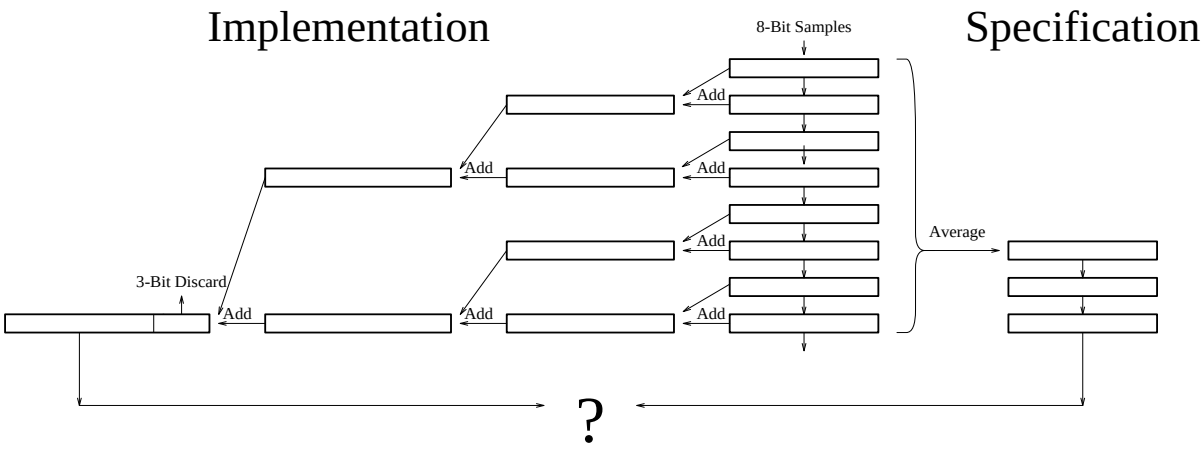


Figure 6.3: Diagram of Size 8 Moving-Average Filter: The verification task is to prove that the implementation using a pipelined tree of adders gives the same result as the specification, which computes the average directly and then delays the results in a FIFO to match the latency of the implementation.

blow up, but if we combine the properties into an implicit conjunction, verification completes quickly and efficiently because the BDD that specifies the property for one layer of the filter greatly simplifies the BDD that specifies the property for the next layer of the filter. Implicitly conjoined BDDs, therefore, provide a natural mechanism for the user to assist the verifier by introducing additional information.

Table 6.3 gives results for this example. Again, we see that the conventional BDD approaches cannot handle the larger instances of this example, but the methods using implicit conjunctions can. Note also that the more sophisticated heuristics automatically derived the BDDs for the additional properties needed to complete the verification, whereas the the simpler heuristics required the user to specify them.

6.4.4 A Simple Pipelined Processor

The last example is to verify a simple pipelined processor against a non-pipelined specification. To reduce the size of the model, and since I am only concerned with the processor, I will abstract away the memory. Instead, both versions of the processor will execute the same non-deterministically-generated stream of instructions. Instructions are encoded as a 3-bit opcode, followed by fields specifying the source and

n	Method	Iterations	Time (sec)	Memory	Max BDD	(breakdown)
4	Fwd	3	48.58	7204K	11267	
	Bkwd	1	4.97	1400K	490	
	IC	1	3.17	1020K	146	(102, 45)
	XIC	1	3.60	1000K	146	(102, 45)
8	Fwd			SPACE OUT		
	Bkwd		TIME OUT			
	IC	1	23.70	3920K	638	(390, 169, 81)
	XIC	1	23.31	3924K	638	(390, 169, 81)
16	Fwd			SPACE OUT		
	Bkwd			SPACE OUT		
	IC	1	189.56	22316K	2558	(1501, 629, 290, 141)
	XIC	1	181.12	18772K	2558	(1501, 629, 290, 141)

n	Method	Iterations	Time (sec)	Memory	Max BDD	(breakdown)
4	Fwd	3	64.18	6560K	11267	
	Bkwd	2	4.92	1380K	490	
	IC	2	4.86	1412K	490	(490)
	XIC	2	3.40	1124K	146	(102, 45)
8	Fwd			SPACE OUT		
	Bkwd		TIME OUT			
	IC		TIME OUT			
	XIC	3	29.47	5008K	638	(390, 169, 81)
16	Fwd			SPACE OUT		
	Bkwd		TIME OUT			
	IC		TIME OUT			
	XIC	4	308.94	22976K	2558	(1501, 629, 290, 141)

Table 6.3: Moving-Average Filter Results: The upper table shows results for verification with the additional user-specified property that each layer of the implementation must agree with the corresponding buffer element in the specification. The lower table shows results if the user simply asks the verifier to prove that the outputs always agree. Only the more complex heuristics (from Sections 6.3.2 and 6.3.3) can handle this case. n indicates the number of 8-bit values in the moving average. All columns labels and notations are as in the preceding tables.

destination registers, followed by a field for specifying immediate data values. There are eight instructions: NOP, BR, LD, ST, ADD, SUB, MOV, and SR. NOP performs no operation. BR models a branch instruction. Since the instruction stream is generated non-deterministically, the model has no program counter, so the BR instruction essentially performs no operation. (It does, however, stall the pipeline, as we'll see later.) LD loads the specified destination register with the contents of the immediate field. ST is a no-op, since we aren't modeling memory. ADD adds the contents of the specified source register into the specified destination register. SUB subtracts the source register from the destination register. MOV copies the source register into the destination register. SR shifts the contents of the specified destination register right by one bit.

The pipeline is three stages deep. The first stage fetches the next instruction from the non-deterministic instruction stream. The second stage decodes the instruction, fetches the appropriate values from the register file (or the immediate field for a LD) and computes the result. The last stage writes the result back into the register file. There are, of course, some complications. First, if one instruction relies on the result of the preceding instruction, the result won't be written back by the Writeback stage in time for the Execute stage to fetch the correct value, eg.:

```
; assume r0=0 and r1=0
LD r1, #1      ; make r1=1
ADD r0, r1     ; add r1 to r0
```

After executing this code fragment, `r0` should be equal to 1. As described, however, the pipelined processor would not have updated `r1` to be 1 in time for the ADD instruction. The standard solution to this problem is to add a "register bypass path" to the pipeline: If the Execute unit detects that the current instruction needs the result of the previous instruction, it bypasses the register file and gets the value needed directly from the Writeback unit. The example includes such a register bypass path. Another complication occurs because of branches. In a real machine, the Instruction Fetch unit does not know where the next instruction will be until the Writeback unit updates the program counter. For the example, I adopt a standard solution — the branch stall. If any stage in the pipeline contains a BR instruction, the pipeline is

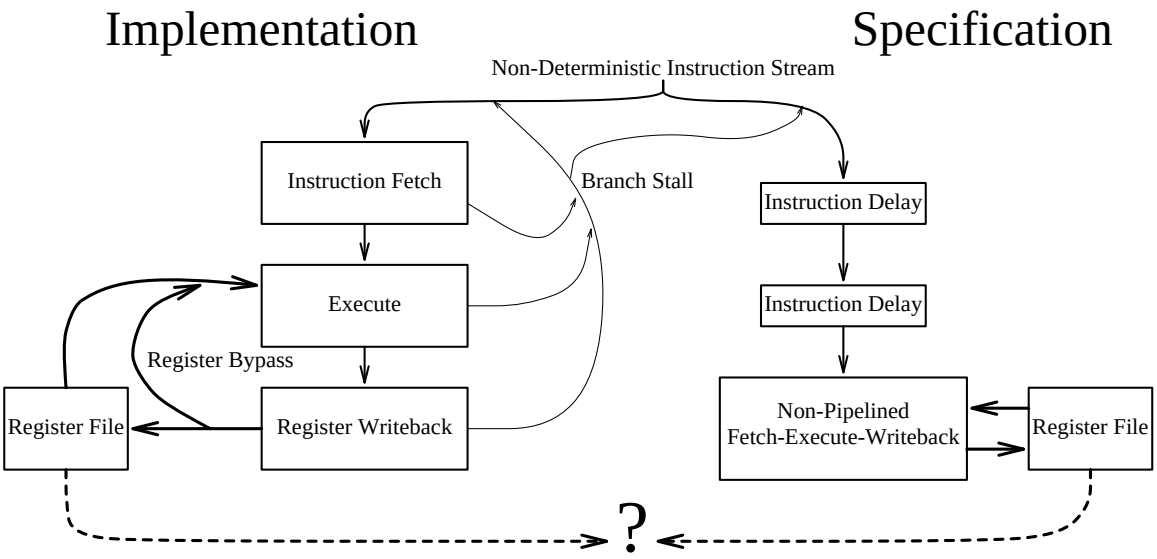


Figure 6.4: Diagram of Pipelined Processor Example. A pipelined and a non-pipelined version of a processor execute the same non-deterministically generated stream of instructions. The verification task is to prove that the register files of the two versions always agree.

forced to stall. (I implement the stall by forcing NOP instructions into the Fetch unit until the BR clears the Writeback unit.)

The verification task is to show that the register files of the pipelined and non-pipelined processors always agree when executing the same sequence of instructions. In order to keep the two descriptions synchronized, the non-pipelined processor buffers incoming instructions for two cycles to match the pipelined processor. Also, a branch stall in the pipeline will also stall the non-pipelined processor. This example is summarized in Figure 6.4. As in the preceding examples, there are many relationships between the state variables of the description, leading us to expect BDD-size blow-up. Table 6.4 summarizes the results for this verification example for various numbers of registers and datapath sizes (the bit-width of the registers and the immediate field).

6.5 Comments

As in the preceding chapter, the techniques presented here are not the last word in BDD-based formal verification. Countless other causes of BDD-size blow-up exist.

Size	Method	Iter.	Time (sec)	Memory	Max BDD	(breakdown)
2R, 1B	Fwd	4	296.03	46816K	284745	
	Bkwd	4	24.82	3884K	10745	
	IC	4	24.04	3884K	10745	(10745)
	XIC	4	9.01	1488K	910	(646, 172, 86, 9)
2R, 2B	Fwd			SPACE OUT		
	Bkwd			SPACE OUT		
	IC			SPACE OUT		
	XIC	4	92.53	4192K	8485	(6657, 1345, 441, 45)
2R, 3B	Fwd			SPACE OUT		
	Bkwd			SPACE OUT		
	IC			SPACE OUT		
	XIC	4	1232.99	19384K	57510	(45230, 9591, 2503, 189)
4R, 1B	Fwd			SPACE OUT		
	Bkwd			SPACE OUT		
	IC			SPACE OUT		
	XIC	4	380.65	10304K	12947	(10767, 1290, 849, 45)

Table 6.4: Pipelined Processor Results: The size column shows the number of registers and the width of the datapath in bits. All other columns are as in the preceding tables. Again, we find that only the implicitly conjoined BDD method with the more sophisticated heuristics is able to handle the larger instances.

On the other hand, these techniques do apply to empirically common causes of BDD-size blow-up and pay-off dramatically when they work. The examples presented here are not impressively large. Rather, they are embarrassingly small — embarrassing for the standard BDD approaches that cannot handle what appear to be straightforward, pedestrian verification problems. This chapter enables us to solve these problems, providing a necessary step to more practical formal verification.

It is worth noting that some non-canonical, generalized BDD data structures published previously, such as Jeong *et al.*'s Extended BDD [50] or Jain *et al.*'s Indexed BDD [48], can be used to represent the implicit conjunction of a set of BDDs as well as many other combinations of BDDs. By restricting ourselves to the special case of implicit conjunctions — an empirically-motivated and useful special case — we can produce a more efficient implementation and the fast, specialized manipulation routines that gave us good results in this chapter. In the future, we may need to move toward the more general data structures, but we should do so only to the extent demanded by the real examples we try to verify.

Chapter 7

Contributions and Future Work

Chapter Overview

This chapter summarizes the main contributions of the thesis, notes the importance of a problem-driven research approach to maintain a practical focus, and suggests some avenues for further research.

7.1 Contributions

All the disparate parts of this thesis have a unified direction: they are all necessary steps we must take to make high-level automatic formal verification useful on real problems in the real world. In particular, the main contributions of this thesis are as follows:

- I have identified a set of description language constructs that are efficiently translated into BDDs, yet are powerful and expressive enough to handle real problems and are natural and easy-to-use for the user. I have described the translation process in detail and implemented it as well. This automated translation is a crucial step to making BDD-based verification easy enough for practical use. The user must be not be burdened with translating complex, real designs into Boolean logic.

- I have highlighted the BDD-size explosion problem and explained intuitively why it's so common in high-level verification. Previously, most researchers had expected a naive, direct use of BDDs to surmount the state-explosion problem in most practical verification problems. We now know this is not the case for high-level verification, and, more importantly, we have intuition behind what causes the BDD-size blow-up.

- I have developed a technique for image computation without building the BDD for the transition relation, thereby avoiding a major cause of BDD-size blow up. This technique, an extension of Burch *et al.*'s disjunctive partitioned transition relation [9], enables computation with large, practical-sized, high-level descriptions, something that had not in general been possible previously. Obviously, if we are unable to build a computationally usable description of the system being verified, we cannot even begin to perform verification, so this technique is a key enabling technology.

- I have identified some empirically common causes of BDD-size blow-up and have developed two modifications — functionally dependent variables and implicitly conjoined BDDs — to the standard BDD-based verification algorithms in order to address these causes of BDD-size blow-up. On some small but relevant examples, these methods demonstrate considerable improvement in speed and memory usage over the normal BDD-based algorithms.

- I have developed good heuristics for dealing with implicitly conjoined BDDs. These heuristics are important in reducing the BDD-expertise required of the user of a verification system. Such increases in automation are critical to the future practicality of formal verification.

7.2 Future Work

This thesis expands the range of problems that can be verified automatically. We have not, however, made enough progress: simple examples that still cannot be verified automatically are distressingly common. Much work remains to be done.

Obviously, any general advances in BDD algorithms or any improved BDD variant will pay off immediately for formal verification. For instance, I mentioned in Chapter 1 that there already exists a menagerie of BDD variants, and more are being published regularly. When the smoke clears, we may find a new data structure that is superior for our purposes. On a different front, finding good heuristics for BDD variable ordering is still an open problem. Some heuristics exist for gate-level digital circuits (e.g., [34, 63, 51, 33]), but little work has been done for high-level verification. In some sense, variable ordering should be easier at the high-level, because the structure of the system being described is more apparent. Many people, especially those working at the gate level or below, consider dynamic variable reordering [81] to be the ultimate solution. I disagree. Dynamic variable reordering is outstanding for (1) verifying systems in which the good variable orders change significantly over time (something I have never encountered in practice although other researchers claim to have), (2) presenting an application interface which completely hides the issue of variable ordering, and (3) providing passable performance when the user has no clue about what might make a good order. The last two cases are especially valuable, as they reduce the BDD expertise required of the user and increase the degree of automation. These two cases, however, are also completely orthogonal to the issue of *dynamically reordering* the variables; if we had a good variable-ordering heuristic, we could order the variables statically at the start and get better performance. (Indeed, a common use of dynamic variable reordering is to use dynamic ordering for a while, interrupt the program and dump the current variable order, and then restart the program from the beginning using the dumped variable order as a static variable order.) In my own experience with high-level verification, activating dynamic variable reordering cut BDD sizes somewhat (say, in half) at the cost of roughly an order of magnitude increase in run time. Although dynamic variable reordering is certainly an important and valuable technique, it definitely does not obviate the need for good static variable ordering heuristics.

Another direction is to use conventional BDDs in unconventional ways. This thesis largely falls in this category and has successfully eliminated several BDD-size problems. Further research in this direction could produce additional results. For

instance, the heuristics for simplifying implicitly conjoined BDDs are rather crude. I have been experimenting with techniques to use externally-generated invariants or user-supplied structural information to help find good partitions into implicitly conjoined lists of BDDs.

Combining state-exploration methods with automated theorem-proving has recently started attracting interest. The promise is that the expressiveness of the theorem prover can support abstraction and decomposition of complex designs in a safe way. The theorem prover can then rely on state-exploration verification techniques as powerful decision procedures for finite-state pieces of the design, while the state-exploration verifier can rely on the theorem prover for don't-care information derived from the decomposition process. Implicitly conjoined BDDs, used to add additional verification properties as described in Section 6.4.3, may be an effective means for a theorem-prover to guide a model-checker. Preliminary reports of combined theorem-provers and model-checkers are very promising [52, 80].

My research has been strongly problem-driven. Specific examples of real problems exposed the limitations of the conventional BDD-based approaches and also pointed to workable solutions. Additional problem-driven research should be equally fruitful. One possibility is to target domain-specific verification. By choosing a narrow application domain that is still interesting (e.g. directory-based cache protocols, pipelined microprocessors, DSP algorithms, etc.) we may find specific properties unique to the domain that we can exploit to improve verification efficiency.

The most important lesson I've learned and a lesson that will guide my future research is the necessity of continually testing one's ideas on practical, real problems. Time and again I have ignored this point only to find myself churning out results that bring me no closer to solving problems that matter. There is an infinite number of ideas and variations to explore, all of which are theoretically valid. One could easily spend a lifetime, perhaps a very fulfilling one, never producing anything of practical importance. A constant grounding in practicality is an efficient heuristic for separating the useless ideas from the useful ones. And useful ideas are what we need to advance the state of the art.

You don't like these ideas?
I got others.

—*"Marshall McLuhan" in*
Ann Bogart's The Medium

Appendix A

Ever Verifier Reference Manual

A.1 Introduction

This appendix serves two purposes. First, I intend to make a version of the Ever verifier publicly available shortly, and some people might actually try to use it. For them, this appendix is a tutorial and reference manual. I hope they find it helpful. If you want a copy of Ever, try anonymous ftp to `snooze.stanford.edu` and poke around there. If that fails, try the Web at `http://www-cs-students.stanford.edu/~ajh`. If that fails, you can try to email me at `ajh@cs.stanford.edu`. The second, more important purpose is to help others who are building their own verification systems. I made several mistakes while designing Ever, and I hope others can learn from my errors. Throughout this appendix, I will point out decisions I was particularly happy or unhappy with.

Ever is an automatic BDD-based verifier, using symbolic state-enumeration techniques. The user describes the system being verified using high-level language constructs, specifies the set of start states of the system, and gives a (non-temporal, propositional) property that should hold in every reachable state of the system. The verifier returns (eventually, if it doesn't run out of memory) either indicating that the verification succeeded or providing a counterexample trace from a start state to a state that violates the property being checked. Details about the theory behind Ever can be found in the rest of this thesis.

Historically, Ever was intended as a BDD-based back-end to the Murφ verification system. It is a separate program with its own input language solely for software engineering reasons (to separate two programmers with incompatible coding styles). The origin of Ever as a quick-and-dirty intermediate form explains many of its peculiarities. Since then, the Murφ project has moved on in different directions, and Ever has evolved as a research workbench for trying new verification ideas. This creeping-featuristic evolution has grafted many additional peculiarities onto the Ever system.

Some features that distinguish Ever from other BDD-based verifiers include:

The Good — The main positive features of Ever stem from its high-level orientation. Ever supports unsigned integer and enumerated types, records, and arrays much the way a high-level language would. Ever supplies numerous built-in operators for arithmetic and logical operations. Ever models are usually written with sequential semantics: control flow constructs include sequencing, if-then-else, and non-deterministic choice. Sequential semantics, as opposed to the dataflow model in many other verifiers, have proven to be particularly easy to use for high-level verification. The other main advantage of Ever is that it implements all of techniques in this thesis.

The Bad — Many minor annoyances result from Ever's humble beginning as an intermediate form. For example, Ever doesn't provide symbolic constants or full support of enumerated types, as I had assumed the front-end would handle these issues. Other problems reflect Ever's link to Murφ: e.g., Ever is designed to support an asynchronous interleaving model of concurrency as in Murφ, so modeling synchronous composition of machines is particularly unpleasant (not impossible, though) and nullifies many of Ever's features and advantages. Finally, some problems are simply design mistakes. For example, Ever's type system has severe limitations. I will discuss these issues in more detail at the appropriate points in this appendix.

The Ugly — Ever syntax is strange — not bad, per se, but very strange. Friends have described it as looking like a cross between LISP, Pascal, and FORTRAN.

> The syntax is yet another result of Ever's origins as an intermediate language. The goal was to produce syntax that was trivially easy to generate, compactly equivalent to Murφ programs, trivially easy to parse, and somewhat human-readable and writable to ease debugging the front and back ends separately. The crude syntax has made modifying the language very easy, and the language is generally not painful to read or write. However, the strange syntax tends to dissuade the casual reader, is a bit verbose at times, and occasionally makes code hard to read. If I could go back in time and do it over again, I might still choose the strange, but simple syntax. If I were designing a new verifier today, however, I'd definitely choose more conventional syntax.

Ever is written in C with the help of lex and yacc. It uses David Long's BDD and memory management packages [61], which have proven quite reliable. I've found the code generally portable across a few different versions of UNIX. You can play with Ever on a small machine, but to verify interesting properties of interesting systems, you will need a lot of physical memory. All of the examples in this thesis can run on a Sun SPARCstation 2 with 128MB of RAM; most examples require much less memory.

A.2 A Tutorial Example

An example is perhaps the easiest way to acquaint oneself with the Ever language. Let's consider a toy link-level protocol. See Figure A.1. There are four processes: a source, a transmitter, a receiver, and a destination. These four operate asynchronously, handshaking via shared variables. The source has a string of data, which it sends a character at a time to the transmitter. The transmitter packs the characters into packets, appends a checksum, and sends the packet to the receiver. The receiver unpacks the characters, and sends them one at a time to the destination. The destination receives characters and stores them in a buffer.

Let's write an Ever description for this example. An Ever program consists of a sequential list of declarations and commands. Within a declaration or command, expressions are in LISP-like prefix notation. Array and record accesses are denoted

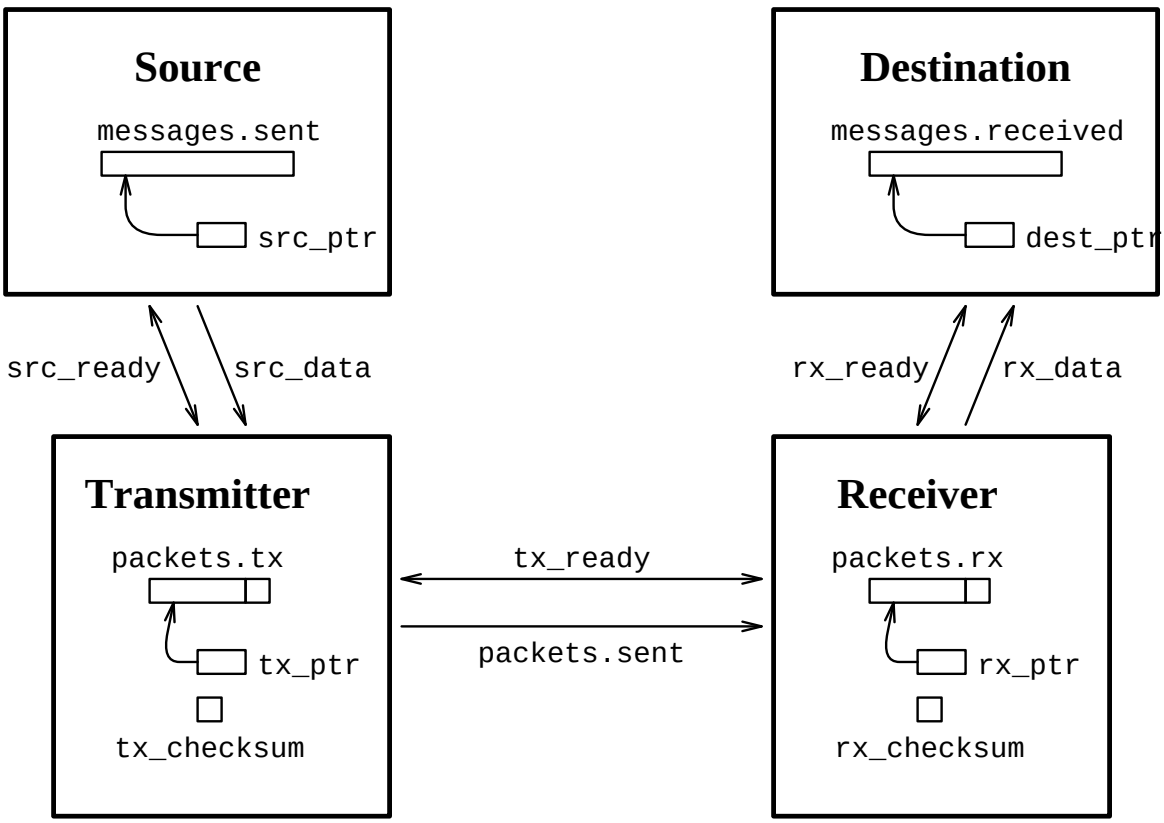


Figure A.1: In this simple example, a message travels from source to destination. The transmitter and receiver handle packets and checksums.

by square brackets and periods, as in Pascal or C.

The description starts with some type declarations:

```
deftype signal (bits 1 "low" "high");
deftype char (bits 1 "a" "b");
deftype integer (bits 4);
deftype packet (record data (array 0 3 char) check char);
```

In Ever, we declare scalar types by the number of bits needed. (The strings following the bit width are used for output.) Thus, types `signal` and `char` are both 1 bit, type `integer` is 4 bits, and type `packet` is a record with two fields: `data`, a four-element array, and `check`, of type `char`.

The next few lines declare variables for the source and destination:

```
defvar src_ready signal;  -- Variable src_ready is of type signal,
defvar src_ptr integer;   -- and so forth.
defvar src_data char;

defvar dest_ptr integer;

defvar messages (interleaved record
                                sent (array 0 7 char)
                                received (array 0 7 char)
                );
```

The source will send the contents of `messages.sent` to the destination, which writes the data into `messages.received`. The characters `--` introduce comments, which extend to the end of the line. The `interleaved` keyword controls BDD variable ordering: without the keyword, BDD variables are in the order declared, with all of the BDD variables for one record field or array element preceding the BDD variables for the next field or element; with the `interleaved` keyword, the BDD variables for all fields of a record or all elements of an array are interleaved (bitslices grouped together). The user cannot control whether low-order or high-order bits come first, except by changing the source and recompiling. Although the change is easy, (Edit

function `create_new_var` in file `eversymtab.c`.) forcing the user to edit the source is a misfeature that should be fixed in the future.

We then declare variables for the transmitter and receiver.

```
defvar packets (interleaved record
                        tx packet
                        sent packet
                        rx packet
                );

defvar tx_ptr integer;
defvar tx_checksum char;
defvar tx_ready signal;

defvar rx_ptr integer;
defvar rx_data char;
defvar rx_checksum char;
defvar rx_ready signal;
```

The next lines define the set of possible start states:

```
defprop StartState
        (and
            (eq src_ptr^c 0)   -- src_ptr must be 0
            (eq src_ready^c 0) -- src_ready must be 0
            (eq tx_ptr^c 0)    -- etc.
            (eq tx_ready^c 0)
            (eq rx_ptr^c 4)
            (eq rx_ready^c 0)
            (eq dest_ptr^c 0)
        );
```

The suffix `^c` specifies the current value of the variable, whereas the suffix `^n` specifies the next value. The formula initializes the control variables, but leaves the data

unspecified, allowing them to assume any value.

Next, we define the transition relation. The definition is in parts. We will define a transition relation for each part of the system, and then compose these transition relations to form the transition relation of the whole system. First, we define the source:

```
defprop Source
        (if (and (le src_ptr^c 7) (eq src_ready^c 0))
           -- If there is another char to send and Tx is ready
           -- then send the char to Tx
           (compose
                (becomes src_data^n messages.sent[src_ptr^c]^c)
                (becomes src_ptr^n (add src_ptr^c 1))
                (becomes src_ready^n 1)
           )
           -- else do nothing
           CurNextEq
        );
```

which simply writes the next character to a port if the port is ready. The `becomes` operator provides deterministic assignment: it generates the BDD transition relation that assigns an expression to a variable, while keeping all other variables constant. Combining the `becomes` operator with the `compose` operator, which provides transition relation composition, produces a transition relation that corresponds to executing a series of assignment statements sequentially. Note that built-in arithmetic operators make it easy to maintain counters. The `CurNextEq` keyword denotes the identity transition relation, which forces all variables to maintain their current values. Next, we define the transmitter:

```
defprop Tx
        (if (and (le tx_ptr^c 3) (eq src_ready^c 1))
            -- If the current packet isn't full and
            --    there is an incoming char
```

```
        -- then add it to the packet and update the checksum
        (compose
            (becomes packets.tx.check^n
                     (add packets.tx.check^c src_data^c))
            (becomes packets.tx.data[tx_ptr^c]^n src_data^c)
            (becomes tx_ptr^n (add tx_ptr^c 1))
            (becomes src_ready^n 0)
        )
        -- else if current packet is full and channel is ready
        (if (and (gt tx_ptr^c 3) (eq tx_ready^c 0))
            (compose    --then send the packet to Rx
                (becomes packets.sent^n packets.tx^c)
                (becomes packets.tx^n 0)
                (becomes tx_ptr^n 0)
                (becomes tx_ready^n 1)
            )
            CurNextEq   --else do nothing
        )
    );
```

The transmitter reads characters from the source, appends them to a packet, and computes a checksum. Whenever it has a complete packet and the channel to the receiver is clear, it sends the packet. The receiver unpacks characters from the packets:

```
defprop Rx
        -- If Rx can receive packet and packet is ready
        (if (and (gt rx_ptr^c 3) (eq tx_ready^c 1))
            -- then get the packet
            (compose
                (becomes packets.rx^n packets.sent^c)
                (becomes packets.sent^n 0)
                -- Error detection and recovery code would go here.
                (becomes rx_ptr^n 0)
```

```
            (becomes tx_ready^n 0)
        )
        -- else if Rx has data for Dest and Dest is ready
        (if (and (le rx_ptr^c 3) (eq rx_ready^c 0))
            (compose    --then send it
                (becomes rx_data^n packets.rx.data[rx_ptr^c]^c)
                (becomes rx_ptr^n (add rx_ptr^c 1))
                (becomes rx_ready^n 1)
            )
            CurNextEq   --else do nothing
        )
    );
```

In a real link-level protocol, we would insert code to check the checksum and handle errors at the indicated point. The destination is easy:

```
defprop Destination
    (if (eq rx_ready^c 1) --If Rx has data for us
        (compose          --then get it
            (becomes messages.received[dest_ptr^c]^n rx_data^c)
            (becomes dest_ptr^n (add dest_ptr^c 1))
            (becomes rx_ready^n 0)
        )
        CurNextEq         --else do nothing
    );
```

We define the transition relation for the entire system

```
defprop NextState (or (Source) (Tx) (Rx) (Destination));
```

as the non-deterministic choice of the preceding transition relations.

The remaining lines of the program:

```
defprop Ok (if (gt dest_ptr^c 7)
```

```
(eq messages.sentˆc messages.receivedˆc));

printtrace (StartState) (NextState) (not (Ok));
```

define a simple verification condition and invoke the reachability verifier. The state-reachability computation starts from all states that satisfy the start-state formula and, using the specified transition relation, outputs a trace that reaches the end condition (or is as long as possible).

This program takes three and a half minutes on a Sun SPARCstation 2, using less than 11MB of memory, to compute all reachable states and to find and print a longest acyclic trace (37 states long). Interestingly, even this short example has 84 billion reachable states (out of a state space of 1.8×10^{16} states) putting it well-beyond the reach of non-symbolic state-enumeration verifiers.

A.3 Ever Language Details

From the preceding example, we have a rough feel for the Ever language. A program consists of a sequence of commands that are executed one after another. There is no looping. Each command can be a declaration or an invocation of a verification algorithm. Now, let's look at the language in more detail.

A.3.1 Lexical Units

Comments

Two hyphens `--` introduce a comment. The comment extends to the end of the line. This comment convention has worked well, although occasionally I've wanted nestable, mult-line comments to comment out blocks of code.

Reserved Words

Ever has a very large number of reserved words, reflecting the ad hoc additions of numerous commands and operators. The following is a list of all reserved words. Case is significant.

AF	becomes	defvar	lt
AG	bequiv	defvec	not
AU	bif	end	or
AX	bimplies	eq	orcompress
CurNextEq	bits	equiv	printprop
EF	bnot	eval	printsize
EG	bor	exists	printstring
EU	bxor	forall	printtrace
EX	checkinvariant	ge	reachable
FALSE	checkinvariantfwd	gt	reachgoal
PrevCurEq	checkwithapprox	hidecur	record
PrevNextEq	checkwithsupport	hidenext	simrel
TRUE	compose	hideprev	sub
add	constrain	if	using
and	ctlmodelcheck	implies	vector
andcompress	defdepvar	include	xor
array	defpred	interleaved	
assuming	defprop	lambda	
band	deftype	le	

Identifiers

Identifiers can be any sequence of one or more letters, digits, or the underscore character _, provided it doesn't start with a digit. Case is significant.

Numbers

Ever only supports unsigned integers, so a number is denoted by a string of digits. The number is always assumed to be written in decimal.

I have been able to perform verification of systems requiring negative numbers (e.g. an SRT divider) by considering unsigned numbers to be in 2s complement. This solution is far from ideal; a verifier designed to handle such systems should provide explicit support for both signed and unsigned integers.

By default, Ever allocates the minimum number of bits required to represent a number. Occasionally, the user needs control over the size of an integer. The syntax `'`w`'`n, where w and n are strings of digits, specifies the number represented by n, but tells Ever to consider the number to be w bits wide. For example, `12` in Ever represents the number 12 and is 4 bits wide, whereas `'6'12` in Ever also represents the number 12 but is 6 bits wide. (The width specifier is actually handled by the parser, not the lexer. In normal usage, the user won't notice the difference.)

Strings

Ever doesn't really support strings. String constants exist in the language simply to enhance the readability of output. A string is any sequence of characters (including newlines, etc.) enclosed in double quotes `"`. There is no way to represent double quotes in a string.

Other Lexical Items

There are a few other lexical tokens that Ever uses: the "version designators" `^p`, `^c`, `^n`; parentheses to group expressions; square brackets for array indexing; a period for record accessing; a semicolon at the end of each command; and the left and right shift operators `<<` and `>>`. I mention these here only for completeness; we will see what they do later.

Ever ignores all whitespace, except in strings and to separate tokens.

A.3.2 Type System

The tutorial example shows that Ever supports arrays and records. This fact may lead one to expect that Ever has a type system similar to a normal high-level language. Such an expectation is wrong. The Ever type system is idiosyncratic and perhaps the greatest design error I made.

Ever allows the user to define types consisting of arrays and records of previously defined types, just as a normal high-level language does. In Ever, however, these types apply only to variables, not to the results of expressions. When a variable is

referenced, it is immediately considered to be a bit vector. For example, consider the following variable declaration:

```
defvar x (record f1 (bits 4) f2 (array 0 3 (bits 3)));
```

which declares `x` to be a record with two fields: the first is a 4-bit integer; the second, an array of 3-bit integers. Ever knows the type of `x`, so it is legal to write expressions like `x^c`, `x.f2^c`, and `x.f2[2]^c` in Ever. As soon as Ever sees these expressions, however, it discards any type information, so it is also legal to write the expression:

```
(add x^c x.f2^c x.f2[2]^c)
```

even though the three arguments are intuitively of different types. In Ever, this expression is just the sum of three different-sized bit vectors, producing another bit vector as the result.

As we can see, Ever expressions do not obey the type declarations for variables. Instead, Ever expressions can only be one of two types: propositions and bit vectors. A proposition can take on the values true and false. A bit vector can be considered to be the binary representation of an unsigned number. Ever considers propositions to be a different type from bit vectors of size 1.

Similarly, user-defined types do not apply to user-defined predicates or their parameters. Instead, user-defined predicates are always considered to be of type proposition, and parameters are always considered to be of type bit vector. This limitation is particularly onerous.

The rationale behind this type system was to trivialize the type-checking portion of the Ever verifier. The type of every expression is uniquely determined by the top-level operator in that expression, so type checking is a local syntactic check. The weak typing means almost no type casting and type checking is required. The drawbacks of this type system are that it is cumbersome to use, error-prone, difficult to extend or modify, and not conducive to code modularization. I strongly encourage the use of a more conventional type system in future verifiers.

A.3.3 Variable Versions

Ever tries to hide many of the details of BDD-based verification from the user. When designing Ever, however, I was not sure how successful this hiding could be. As a result, many low-level BDD details are still visible and accessible to the user. While these low-level escape hatches do provide flexibility when used very carefully, they also add unintuitive ugliness to the language.

The visibility of different versions of the variables is a particularly prominent ugliness. When the user declares a variable, rather than creating a single copy of the variable, Ever creates three copies: the previous version, the current version, and the next version. A BDD-based verifier must maintain at least two versions of each variable internally in order to model transition relations (since BDDs are propositional and we need a BDD to represent a relation between the current and next states of the system). The third copy permits computing relational products between transition relations.

Ever makes these three versions of the variable explicitly available to the user. Every variable reference must be followed by `^p` to specify the previous version, `^c` to specify the current version, or `^n` to specify the next version. This explicit access lets the user create and manipulate transition relations directly. For example, if a system has only two variables:

```
defvar x (bits 4);
defvar y (bits 4);
```

we can create a transition relation that only affects `x`, such as `(eq x^n (add x^c 1))`, and another transition relation that only affects `y`, such as `(eq y^n (add y^c 3))`. We could then create the synchronous composition of these two by simply conjoining the relations:

```
defprop synch (and (eq x^n (add x^c 1)) (eq y^n (add y^c 3));
```

However, low-level manipulation is error-prone, especially when mixed with the high-level features that distinguish Ever. For example:

```
defprop synch (and (eq x^n (add x^c 1)) (becomes y^n (add y^c 3));
```

results in an empty transition relation. For the future, I recommend forbidding the user direct access to the different versions of the BDD variables, providing instead the semantics the user requires in a safe and high-level manner.

A.3.4 Expressions

As we've seen, all expressions in Ever are of two types: propositions and bit vectors. In describing the syntax of the expressions in this section, let p and p_i denote arbitrary expressions of type proposition, and let b and b_i denote arbitrary expressions of type bit vector.

Bit Vectors

The following list describes all ways to form an expression of type bit vector.

- *identifier*

  This denotes the bit vector bound to *identifier*. The identifier must be a formal parameter in a predicate definition or must have been previously declared using the `defvec` command.

- *number*

  This denotes the bit vector that is the binary representation of *number*.

- *variable-designator*

  This denotes the bit vector corresponding to the specified bits of the variable. The variable designator must specify which version (previous, current, next) of the variable to use.

- `(vector` v_1 ... v_n `)`

  This expression returns the bit vector consisting of the concatenation of all its arguments. The arguments v_1 ... v_n can be expressions of type bit vector or proposition.

- `(add` b_1 ... b_n `)`

  Adds its arguments, which must be expressions of type bit vector. Discards any carry. Overflow wraps. The size of the result is the size of the larger of the two addends. If there are more than two arguments, the addition is left-associative. Because of the sizing and associativity rules, `(add 10 1 1)` is equal to 12, whereas `(add 1 1 10)` equals 10. You may use sized integers to avoid this wraparound. For example, `(add '4'1 1 10)` equals 12.

- `(sub` b_1 b_2 `)`

  Both arguments must be bit vectors. Returns the difference formed by the first argument minus the second, discarding the borrow. Underflow wraps. The size of the result is the larger of the sizes of the arguments.

- `(<<` b n `)`

- `(>>` b n `)`

  The first argument must be a bit vector. The second argument must be an integer constant. Returns a copy of the first argument logically left-shifted or right-shifted by the amount of the second argument. The size of the result is adjusted by the amount of shift.

- `(band` b_1 ... b_n `)`

- `(bor` b_1 ... b_n `)`

  Bitwise Boolean AND/OR with low-order bits aligned. Associativity and sizing rules are exactly as for `add`.

- `(bnot` b `)`

  Complement the bits in bit vector b.

- `(bxor` b_1 b_2 `)`

  Bitwise Boolean exclusive-or.

- `(bequiv` b_1 b_2 `)`

  Bitwise Boolean exclusive-nor.

- `(bimplies` b_1 b_2 `)`

  Bitwise Boolean implication.

- `(bif` p b_1 b_2 `)`

  Argument p must be an expression of type proposition. The other two arguments must be bit vectors. Returns a bit vector that is equal to b_1 when p is true and that is equal to b_2 when p is false.

Propositions

The following list describes all ways to form an expression of type proposition:

- `TRUE`
- `FALSE`

  These reserved words are predefined constants for the values true and false.

- `PrevCurEq`
- `CurNextEq`
- `PrevNextEq`

  These reserved words are predefined constants that return relations forcing equality between the specified versions of the variables.

- `(` *identifier* `)`

  The value of this expression is whatever proposition *identifier* was previously declared to be using the `defprop` command.

- `(` *identifier* b_1 ... b_n `)`

  The value of this expression is whatever proposition *identifier* was previously declared to be using the `defpred` command, but with the actual parameters

b_1 ... b_n substituted (via macro substitution on the parse tree) in place of the formal parameters. The actual parameters must be expressions of type bit vector.

- `(eval` p `)`

  The value of this expression is just the value of expression p, which must be of type proposition. However, `eval` forces the (re)building of the BDD for the expression p, even if one exists already. This operator is a holdover from when I had a different notion of when the BDDs for expressions would be built. Currently, no BDDs are built until they are needed by a specific verification algorithm, and we can force evaluation by tricking the verification algorithm. **I do not recommend using this operator.**

- `(and` p_1 ... p_n `)`

- `(or` p_1 ... p_n `)`

  Returns the logical conjunction/disjunction of the expressions p_1 ... p_n, which must be expressions of type proposition.

- `(not` p `)`

  Returns the logical negation of the expression p, which must be an expression of type proposition.

- `(xor` p_1 p_2 `)`

  Returns the exclusive-or of propositions p_1 and p_2.

- `(equiv` p_1 p_2 `)`

  Returns the exclusive-nor of propositions p_1 and p_2.

- `(implies` p_1 p_2 `)`

- `(if` p_1 p_2 `)`

  Returns the logical implication $p_1 \Rightarrow p_2$.

- `(if` p_1 p_2 p_3 `)`

  Returns the if-then-else expression "if p_1 then p_2 else p_3".

- `(forall` *variable-list* p `)`

- `(exists` *variable-list* p `)`

  Returns the proposition p with the variables in the variable list universally or existentially quantified. **Currently unimplemented.**

- `(hideprev` p `)`

- `(hidecur` p `)`

- `(hidenext` p `)`

  Returns p with all of the specified versions of the variables existentially quantified.

- `(reachable` p_1 p_2 `)`

  Returns a proposition which specifies the set of all reachable states starting from the set specified by p_1 and using p_2 as the transition relation. **Currently unimplemented.**

- `(reachgoal` p_1 p_2 p_3 `)`

  Like `reachable`, except it returns as soon as the set intersects p_3, rather than computing the entire reachable state set. **Currently unimplemented.**

- `(AG` p_1 p_2 `)`

  Returns the set of states that satisfy the CTL formula AGp_1. p_2 is a holdover from when I thought I would specify the transition relation as part of the expression. **Only works with the `ctlmodelcheck` command.**

- `(EG` p_1 p_2 `)`

  Returns the set of states that satisfy the CTL formula EGp_1. p_2 is a holdover from when I thought I would specify the transition relation as part of the expression. **Currently unimplemented.**

- `(AF` p_1 p_2 `)`

  Returns the set of states that satisfy the CTL formula AFp_1. p_2 is a holdover from when I thought I would specify the transition relation as part of the expression. **Only works with the `ctlmodelcheck` command.**

- `(EF` p_1 p_2 `)`

  Returns the set of states that satisfy the CTL formula EFp_1. p_2 is a holdover from when I thought I would specify the transition relation as part of the expression. **Currently unimplemented.**

- `(AX` p_1 p_2 `)`

  Returns the set of states that satisfy the CTL formula AXp_1. p_2 is a holdover from when I thought I would specify the transition relation as part of the expression. **Only works with the `ctlmodelcheck` command.**

- `(EX` p_1 p_2 `)`

  Returns the set of states that satisfy the CTL formula EXp_1. p_2 is a holdover from when I thought I would specify the transition relation as part of the expression. **Currently unimplemented.**

- `(AU` p_1 p_2 p_3 `)`

  Returns the set of states that satisfy the CTL formula A[p_1Up_2]. p_3 is a holdover from when I thought I would specify the transition relation as part of the expression. **Only works with the `ctlmodelcheck` command.**

- `(EU` p_1 p_2 p_3 `)`

  Returns the set of states that satisfy the CTL formula E[p_1Up_2]. p_3 is a holdover from when I thought I would specify the transition relation as part of the expression. **Currently unimplemented.**

- `(simrel` p_1 p_2 `)`

  Computes a simulation relation between the transition relations given by propositions p_1 and p_2. **Currently unimplemented.**

- `(becomes` *variable-designator* b `)`

  Computes a transition relation that corresponds to an assignment statement. (The variable being assigned gets the new value, and all other variables keep their current values.) The expression b must be of type bit vector and should only refer to the current state versions of the variables. *variable-designator* specifies the variable or the part of a variable being assigned to. It should use the next state version of the variable. This operator correctly handles array indexing with arbitrary bit-vector-valued index expressions.

- `(constrain` *variable-designator* p `)`

  A non-deterministic version of `becomes`. Computes a transition relation that allows the designated variable to take any value that satisfies proposition p and holds all other variables to their current values. Both the *variable-designator* and p should refer to the next state versions of the variables.

- `(compose` p_1 ... p_n `)`

  Indicates transition relation composition. If p_1 ... p_n are transition relations between the current and next state versions of the variables, `compose` will return the transition relation between the current and next state versions of the variables corresponding to the relational product of p_1 through p_n (intuitively equivalent to executing the transition relations one after the other).

- `(gt` b_1 b_2 `)`
- `(ge` b_1 b_2 `)`
- `(eq` b_1 b_2 `)`
- `(le` b_1 b_2 `)`
- `(lt` b_1 b_2 `)`

  Unsigned numerical comparison between bit vectors b_1 and b_2. Operators provide greater-than, greater-or-equal, equal, less-or-equal, and less-than.

- `(andcompress` b `)`
- `(orcompress` b `)`

  Returns the result of computing the logical conjunction/disjunction of all of the bits in bit vector b. **Currently unimplemented.**

A.3.5 Declarations

So far, we have been using declarations freely. Now we can take a closer look. There are six kinds of declarations: types, variables, dependent variables, propositions, predicates, and bit vectors. Let's consider these one at a time.

Type Declarations

The `deftype` command declares a new type.

`deftype` *identifier type-descriptor* ;

This command declares that *identifier* is the name of a new type. The type descriptor must be one of the following:

- *identifier*

  where *identifier* is the name of a previously declared type.

- `(bits` n s_0 ... s_{2^n-1} `)`

  where n is a positive integer and s_0 ... s_{2^n-1} are string constants. This creates a type that is n bits wide. The strings are optional. They are used to output values of this type. If a variable of this type has value i, it will output as s_i.

- `(array` n_1 n_2 *type-descriptor* `)`

  where n_1 and n_2 are integers with $n_1 < n_2$. This creates an array type that is indexed from n_1 to n_2 inclusive. The array elements are of the specified type.

- `(record` id_1 $type_1$ ... id_n $type_n$ `)`

  This creates a record type. The identifier id_i is the name of the ith field, and its type is given by the type descriptor $type_i$.

In addition, the reserved words `array` and `record` can be preceded by the reserved word `interleaved` to specify an interleaved BDD variable order.

Variable Declarations

The `defvar` command declares a new variable.

> `defvar` *identifier type-descriptor* ;

This command declares that *identifier* is the name of a new variable whose type is given by the type descriptor.

Dependent Variable Declarations

Dependent variables must be declared to use the functionally dependent variable method presented in Chapter 5. The declaration is just like a normal variable declaration, except that the user must also specify the value the variable should have. For example, if we want to declare `z` to be a dependent variable that is always the sum of variables `x` and `y`, we would type the following:

```
defvar x (bits 4);
defvar y (bits 4);
defdepvar z (bits 4) (add x^c y^c);
```

The general syntax is

> `defvar` *identifier type-descriptor bit-vector* ;

Proposition Declarations

A proposition declaration simply binds an expression of type proposition to an identifier, providing a shorthand for repetitive expressions. The syntax is

> `defprop` *identifier proposition* ;

Predicate Declarations

Predicate declarations are like proposition declarations, but with parameters. The distinction of propositions versus predicates is entirely artificial and results from a design error I made initially. The syntax is

`defpred` *identifier* (*formal-list*) *proposition* ;

where *formal-list* is a space-separated list of identifiers that are the formal parameters of the predicate.

Note that the formal parameters are untyped. They are assumed to be bit vectors. This is a major design flaw, as it prevents using the type information of complex data structures passed as parameters.

I had originally intended to provide a lambda operator to create a parameterized predicate from a non-parameterized expression. This feature is currently unimplemented.

Bit Vector Declarations

Bit vector declarations are a convenient shorthand, much like proposition declarations. The syntax is

`defvec` *identifier bit-vector* ;

where *bit-vector* is any expression that returns a bit vector.

Note the absence of parameters and of a type descriptor for the return value. These are design errors that seriously limit the usefulness of bit vector declarations.

Scope

The scoping rules in Ever are trivial. There is a single global scope. All identifiers must be declared before use. There is no recursion and no forward declarations.

The only other scope is within the declaration for a predicate. In this scope, the formal parameters are visible and obscure any conflicting declarations from the global scope. At the end of the declaration, the formal parameters disappear, returning the preexisting global scope.

A.3.6 Commands

An Ever program is simply a sequence of commands. All of the declarations are commands. The following is a list of all other commands.

- `end;`

  Exits the verifier.

- `include` *filename* ;

  where *filename* is a string constant that contains the name of a file. The verifier will read the input from the specified file as if it were typed to the verifier at this point. The `include` command does not nest.

- `printstring` *string* ;

  Prints the given string.

- `printprop` p ;

  Prints the BDD for proposition p.

- `printsize` p ;

  Prints the number of BDD nodes used to represent p. **This command is only partially implemented.** If the BDD for p has already been constructed, this command will print its size. If the BDD for p has not yet been constructed, this command will attempt to count the number of nodes in all the BDDs that make up p. However, in this case, the count may be wrong if p contains operators that the count function doesn't know about. `printsize` will print warning messages if the size reported might be wrong.

- `printtrace` p_1 p_2 p_3 ;

  where p_1, p_2, and p_3 are propositions. Performs a forward traversal, starting from the states specified by p_1, using p_2 as the transition relation, trying to reach a state that satisfies p_3.

If the system contains declared dependent variables, `printtrace` will automatically use the method of functionally dependent variables from Chapter 5. This method, however, is not fully implemented. In particular, `and`, `or`, `not`, `eq`, `vector`, `add`, `sub`, and `bif` are currently the only operators supported when using functionally dependent variables.

- `checkinvariant` p_1 p_2 p_3 ;

  performs a backward traversal. p_1 specifies the start states, p_2 specifies the transition relation, and p_3 specifies the set of good states. If p_3 is the conjunction of several properties, `checkinvariant` automatically uses implicitly conjoined BDDs. To prevent that, wrap an `or` around p_3.

  Switching between the simpler and more sophisticated heuristics for implicit conjunctions requires editing the source code and recompiling. See the functions `converged` and `conjunct_and_simplify` in the file `everinv.c`.

  This command does not currently support functionally dependent variables.

- `ctlmodelcheck` p_1 p_2 p_3 ;

  performs CTL model checking. p_1 specifies the states to be checked, p_2 specifies the transition relation, and p_3 specifies the CTL formula to check.

  This is currently a partial implementation. Only ACTL is supported, and the only logical connectives permitted between modal formulas are `and`, `or`, and `not`.

  This command does not currently support functionally dependent variables. **I do not recommend using this command until further notice.**

- `checkwithapprox` p_1 p_2 p_3 `assuming` p_4 ;

  This is an experimental command to use approximations to help the backward traversal (not described in this thesis). p_1 through p_3 are as in `checkinvariant`. p_4 is a proposition that gives assumptions we can use. As in `checkinvariant`, a top-level `and` operator causes the creation of an implicit conjunction.

This command does not currently support functionally dependent variables. **I do not recommend using this command until further notice.**

- `checkwithsupport` p_1 p_2 p_3 `using` *bit-vector-list* ;

  This is another experimental command. It attempts to use the user-specified bit-vector-list to create good implicit conjunctions (not described in this thesis).

  This command does not currently support functionally dependent variables. **I do not recommend using this command until further notice.**

- `checkinvfwd` p_1 p_2 p_3 ;

  This is another experimental command. It is the dual of checkinvariant using implicit conjunctions. It performs a forward traversal using implicit disjunctions.

  This command does not currently support functionally dependent variables. **I do not recommend using this command until further notice.**

The structure of Ever programs as an arbitrary sequence of commands was designed to provide flexibility. For a research workbench like Ever, such flexibility is convenient. For a production tool, however, I recommend adopting a more structured language that prevents the user from writing blatantly nonsensical descriptions.

Bibliography

[1] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. In *International Conference on Computer-Aided Design*, pages 188–191. IEEE, 1993.

[2] Derek L. Beatty, Randal E. Bryant, and Carl-Johann H. Seger. Synchronous circuit verification by symbolic simulation: An illustration. In William J. Dally, editor, *Advanced Research in VLSI: Sixth MIT Conference*. MIT Press, 1990.

[3] S. Bose and A. Fisher. Automatic verification of synchronous circuits using symbolic logic simulation and temporal logic. In Luc J. M. Claesen, editor, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*. North Holland, 1989.

[4] Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, 1979. As of this writing, current information on nqthm (the Boyer-Moore theorem prover) is available at `http://www.cli.com/software/nqthm/index.html`.

[5] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

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

[7] Randal E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, February 1991.

[8] Randal E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. Technical Report CMU-CS-92-160, Carnegie Mellon University, July 1992. A version of this report was published in *Computing Surveys*, Vol. 24, No. 3, September 1992.

[9] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In *VLSI '91: International Conference on Very Large Scale Integration*, Edinburgh, Great Britain, 1991. IFIP TC 10/WG 10.5.

[10] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Conference on Logic in Computer Science*, pages 428–439, 1990. An extended version of this paper appeared in *Information and Computation*, Vol. 98, No. 2, June 1992.

[11] J. R. Burch, E. M. Clarke, K. L. McMillan, and David L. Dill. Sequential circuit verification using symbolic model checking. In *27th ACM/IEEE Design Automation Conference*, pages 46–51, 1990.

[12] Gianpiero Cabodi, Paolo Camurati, and Stefano Quer. Efficient state space pruning in symbolic backward traversal. In *International Conference on Computer Design*, pages 230–235. IEEE, October 1994.

[13] K. Mani Chandy and Jayadev Misra. *Parallel Program Design*. Addison-Wesley, 1988.

[14] Kwang-Ting Cheng and Vishwani D. Agrawal. *Unified Methods for VLSI Simulation and Test Generation*. Kluwer Academic Publishers, 1989.

[15] Massimiliano Chiodo, Thomas R. Shipe, Alberto Sangiovanni-Vincentelli, and Robert K. Brayton. Automatic compositional minimization in CTL model checking. In *IEEE International Conference on Computer-Aided Design*, pages 172–178, 1992.

[16] Hyunwoo Cho, Gary Hachtel, Seh-Woong Jeong, Bernard Plessier, Eric Schwarz, and Fabio Somenzi. ATPG aspects of FSM verification. In *International Conference on Computer-Aided Design*, pages 134–137. IEEE, 1990.

[17] E. M. Clarke, E. A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *Symposium on Principles of Programming Languages*, pages 117–126. ACM, 1983. An extended version of this paper appeared in *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 2, April 1986.

[18] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Workshop on Logics of Programs*, pages 52–71, May 1981. Published 1982 as Lecture Notes in Computer Science Number 131.

[19] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *Symposium on Principles of Programming Languages*, pages 343–354. ACM, 1992.

[20] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139, 1989.

[21] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of sequential machines using boolean functional vectors. In Luc J. M. Claesen, editor, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*. North Holland, 1989.

[22] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*. Springer-Verlag, 1989. Lecture Notes in Computer Science Number 407.

[23] Olivier Coudert and Jean Christophe Madre. A unified framework for the formal verification of sequential circuits. In *International Conference on Computer-Aided Design*, pages 126–129. IEEE, 1990.

[24] Patrick Cousot. Methods and logics for proving programs. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, chapter 15, pages 841–993. MIT Press, 1994.

[25] Keith Cross, Thomas J. Kosnik, and Modesto Maidique. Microsoft LAN manager. Case S-MT-11, Stanford Graduate School of Business, 1990. The same McKinsey & Co. study is also noted in Port *et al* [78].

[26] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[27] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design*. IEEE, October 1992.

[28] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, chapter 16, pages 995–1072. MIT Press, 1994.

[29] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: On branching versus linear time. In *Symposium on Principles of Programming Languages*, pages 127–140. ACM, 1983.

[30] Thomas Filkorn. Functional extension of symbolic model checking. In K. G. Larsen and A. Skou, editors, *Computer-Aided Verification: Third International Workshop*. Springer-Verlag, July 1991. Published 1992 as Lecture Notes in Computer Science Number 575.

[31] Robert W. Floyd. Assigning meanings to programs. In Jacob T. Schwartz, editor, *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967. Proceedings of 19th Symposium in Applied Mathematics, 1966.

[32] Steven J. Friedman and Kenneth J. Supowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions on Computers*, 39(5):710–713, May 1990.

[33] Hiroshige Fujii, Goichi Ootomo, and Chikahiro Hori. Interleaving based variable ordering methods for ordered binary decision diagrams. In *International Conference on Computer-Aided Design*, pages 38–41. IEEE, 1993.

[34] M. Fujita, H. Fujisawa, and N. Kawato. Evaluations and improvements of a boolean comparison program based on binary decision diagrams. In *International Conference on Computer-Aided Design*, pages 2–5. IEEE, 1988.

[35] Michael R. Garey and David S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, 1979. p. 222.

[36] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993. As of this writing, up-to-date information and instructions for obtaining HOL are available at `http://www.cl.cam.ac.uk/Research/HVG/HOL/index.html` or `http://lal.cs.byu.edu/lal/hol-documentation.html`.

[37] Mike Gordon. HOL: A machine oriented formulation of higher order logic. Technical Report 68, University of Cambridge Computer Laboratory, 1985.

[38] Aarti Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1:151–238, 1992.

[39] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[40] Robert D. Hof. Intel takes a bullet — and barely breaks stride. *Business Week*, pages 38–39, January 30 1995.

[41] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[42] Alan J. Hu and David L. Dill. Efficient verification with BDDs using implicitly conjoined invariants. In *Computer-Aided Verification: Fifth International Conference.* Springer-Verlag, 1993. Lecture Notes in Computer Science Number 697.

[43] Alan J. Hu and David L. Dill. Reducing BDD size by exploiting functional dependencies. In *30th Design Automation Conference*, pages 266–271. ACM/IEEE, 1993.

[44] Alan J. Hu, David L. Dill, Andreas J. Drexler, and C. Han Yang. Higher-level specification and verification with BDDs. In *Computer-Aided Verification: Fourth International Workshop*. Springer-Verlag, July 1992. Published in 1993 as Lecture Notes in Computer Science Number 663.

[45] Alan J. Hu, Gary York, and David L. Dill. New techniques for efficient verification with implicitly conjoined BDDs. In *31st Design Automation Conference*, pages 276–282. ACM/IEEE, 1994.

[46] Warren A. Hunt, Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.

[47] Nagisa Ishiura and Shuzo Yajima. A class of logic functions expressible by polynomial-size binary decision diagrams. In R. W. Dutton, editor, *VLSI Logic Synthesis and Design*, pages 48–54. IOS, 1990. Published in 1991.

[48] Jawahar Jain, Magdy Abadir, James Bitner, Donald. S. Fussell, and Jacob A. Abraham. IBDDs: An efficient functional representation for digital circuits. In *European Conference on Design Automation (EDAC)*, pages 440–446, 1992.

[49] Jawahar Jain, Jim Bitner, Donald S. Fussell, and Jacob A. Abraham. Probabilistic design verification. In *International Conference on Computer-Aided Design*, pages 468–471. IEEE, 1991.

[50] S.-W. Jeong, B. Plessier, G. Hachtel, and F. Somenzi. Extended bdd's: Trading off canonicity for structure in verification algorithms. In *International Conference on Computer-Aided Design*, pages 464–467. IEEE, 1991.

[51] S.-W. Jeong, B. Plessier, G. D. Hachtel, and F. Somenzi. Variable ordering for FSM traversal. In *International Workshop on Logic Synthesis*, Research Triangle Park, NC, May 1991. MCNC.

[52] Jeffrey Joyce and Carl Seger. The HOL-Voss system: Model-checking inside a general-purpose theorem-prover. In *Higher Order Logic Theorem Proving and Its Applications*, pages 185–198. Springer-Verlag, 1993. Lecture Notes in Computer Science Number 780.

[53] Timothy Y. K. Kam and Robert K. Brayton. Multi-valued decision diagrams. Technical Report UCB/ERL M90/125, University of California, Berkeley, December 1990.

[54] Kevin Karplus. Representing boolean functions with if-then-else DAGs. Technical Report UCSC-CRL-88-28, University of California, Santa Cruz, November 1988.

[55] U. Kebschull, E. Schubert, and W. Rosenstiel. Multilevel logic based on functional decision diagrams. In *European Conference on Design Automation (EDAC)*, pages 43–47, 1992.

[56] Dexter Kozen and Jerzy Tiuryn. Logics of programs. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, chapter 14, pages 789–840. MIT Press, 1994.

[57] Yung-Te Lai and Sarma Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *29th Design Automation Conference*, pages 608–613. ACM/IEEE, 1992.

[58] Leslie Lamport. Verification and specification of concurrent programs. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency: Reflections and Perspectives*, pages 347–374. Springer-Verlag, 1994. Proceedings of 1993 REX School/Symposium. Published 1994 as Lecture Notes in Computer Science Vol. 803.

[59] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The stanford dash multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[60] Nancy G. Leveson. *Safeware: System Safety and Computers.* Addison-Wesley, 1995.

[61] David E. Long. As of this writing, a reasonably current copy of David Long's BDD package is available via anonymous ftp from `emc.cs.cmu.edu`, directory `pub/bdd`, file `bddlib.tar.Z`.

[62] David E. Long. *Model Checking, Abstraction, and Compositional Verification.* PhD thesis, Carnegie-Mellon University, 1993. Available as CMU Tech Report CMU-CS-93-178.

[63] S. Malik, A. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *International Conference on Computer-Aided Design*, pages 6–9. IEEE, 1988.

[64] Zohar Manna and Amir Pnueli. How to cook a temporal proof system for your pet language. In *Symposium on Principles of Programming Languages*, pages 141–154. ACM, 1983.

[65] Yusuke Matsunaga, Patrick C. McGeer, and Robert K. Brayton. On computing the transitive closure of a state transition relation. In *30th Design Automation Conference*, pages 260–265. ACM/IEEE, 1993.

[66] John McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 4*, chapter 26, pages 463–502. American Elsevier, 1969.

[67] K. L. McMillan and J. Schwalbe. Formal verification of the gigamax cache-consistency protocol. In *International Symposium on Shared Memory Multiprocessing*, pages 242–251. Information Processing Society of Japan, 1991.

[68] Christoph Meinel. Branching programs — an efficient data structure for computer-aided circuit design. *Bulletin of the European Association for Theoretical Computer Science*, (46):149–170, February 1992.

[69] Hiroyuki Ochi. *Algorithms and Data Structures for Manipulating Boolean Functions*. PhD thesis, Kyoto University, 1993.

[70] Hiroyuki Ochi, Koichi Yasuoka, and Shuzo Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *International Conference on Computer-Aided Design*, pages 48–55. IEEE, 1993.

[71] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *International Conference on Automated Deduction*, pages 748–752, 1992. Lecture Notes in Artificial Intelligence Number 607.

[72] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *Transactions on Software Engineering*, 21(2):107–125, February 1995. As of this writing, up-to-date information and instructions for obtaining a copy of PVS are available at `http://www.csl.sri.com/pvs.html`.

[73] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization*. Prentice-Hall, 1982. p. 262.

[74] Jaehong Park and M. Ray Mercer. An efficient symbolic design verification system. In *International Conference on Computer Design*, pages 294–298. IEEE, 1993.

[75] Carl Pixley. A computational theory and implementation of sequential hardware equivalence. In *Computer-Aided Verification: Second International Workshop*, pages 293–320, 1990. Published in 1991 as Volume 3 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science.

[76] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–55, 1977.

[77] Amir Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency: Overviews*

and Tutorials, pages 510–584. Springer-Verlag, 1986. Lecture Notes in Computer Science Number 224.

[78] Otis Port, Zachary Schiller, and Resa W. King. A smarter way to manufacture: How 'concurrent engineering' can reinvigorate American industry. *Business Week*, pages 110–117, April 30 1990. Business Week Special Report. Reprints available from Business Week Reprints, P.O. Box 457, Hightstown, NJ 08520, (609)426-5494.

[79] Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Seventeenth Annual Symposium on Foundations of Computer Science*, pages 109–121, 1976.

[80] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification: Seventh International Conference*, pages 84–97. Springer-Verlag, July 1995. Lecture Notes in Computer Science Number 939.

[81] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Conference on Computer-Aided Design*, pages 42–47. IEEE, 1993.

[82] Thomas R. Shiple, Ramin Hojati, Alberto L. Sangiovanni-Vincentelli, and Robert K. Brayton. Heuristic minimization of BDDs using don't cares. Technical Report UCB/ERL M93/58, University of California, Berkeley, July 1993. A version of this paper also appears in the 1994 Design Automation Conference, pp. 225–231.

[83] Arvind Srinivasan, Timothy Kam, Sharad Malik, and Robert K. Brayton. Algorithms for discrete function manipulation. In *International Conference on Computer-Aided Design*, pages 92–95. IEEE, 1990.

[84] Mandayam K. Srivas and Steven P. Miller. Applying formal verification to a commercial microprocessor. In *International Conference on Computer Hardware Description Languages*. IFIP, August 1995.

[85] Herve J. Touati, Hamid Savoj, Bill Lin, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. In *International Conference on Computer-Aided Design*, pages 130–133. IEEE, 1990.

[86] Loring Wirbel. UltraSparc engineers tout emulation, advanced simulation tools. *Electronic Engineering Times*, (854):24, June 26 1995. CMP Publications Inc., 600 Community Drive, Manhassett, NY 11030.

[87] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications*. Prentice-Hall, 1984. A second edition was published in 1992.

[88] Lawrence Yang, David Gao, Jamshid Mostoufi, Raju Joshi, and Paul Loewenstein. System design methodology of UltraSPARC-I. In *32nd Design Automation Conference*, pages 7–12. ACM/IEEE, 1995.