Electronic Notes in Theoretical Computer Science 56 (2001)
http://www.elsevier.nl/locate/entcs/volume56.html 190 pages

Formal Verification based on Boolean Expression Diagrams

Poul Frederick Williams

Abstract

This dissertation examines the use of a new data structure called Boolean Expression Diagrams (BEDs) in the area of formal verification. The recently developed data structure allows fast and efficient manipulation of Boolean formulae. Many problems in formal verification can be cast as problems on Boolean formulae. We chose a number of such problems and show how to solve them using BEDs.

Equivalence checking of combinational circuits is a formal verification problem which translates into tautology checking of Boolean formulae. Using BEDs we are able to preserve much of the structure of the circuits within the Boolean formulae. We show how to exploit the structural information in the verification process.

Sometimes combinational circuits are specified in a hierarchical or modular way. We present a method for verifying equivalence between two such circuits. The method builds on *cut propagation*. Assuming that the two circuits are given identical inputs, we propagate this knowledge through the circuits from the inputs to the outputs. The result is the knowledge of how the outputs of the two circuits correspond, e.g., are the outputs of the two circuits pairwise equivalent? The circuits and the movements of cuts can be described using Boolean formulae.

Symbolic model checking is a technique for verifying temporal specifications of finite state machines. It is well known how finite state machines and the evaluation of the temporal specifications can be expressed using Boolean formulae. We show how to do these manipulations using BEDs. We concentrate on examples which are hard for standard symbolic model checking methods.

Determining whether a formula is satisfiability is a problem which occurs in verification of combinational circuits and in symbolic model checking. Often satisfiability checking is associated with detecting errors. We examine how satisfiability checking can be done using the BED data structure.

Finally, we take a look at how it is possible to extend the BED data

structure. Among other operations, we introduce an operator for computing minimal *p-cuts* in fault trees. A fault tree is a Boolean formula expressing whether a system fails based on the condition ("failure" or "working") of each of the components. A minimal *p-cut* is a representation of the most likely reasons for system failure. This method can be used to calculate approximately the probability of system failure given the failure probabilities of each of the components.

As part of this research, we have developed a BED package. The appendix describes the package from a user's point of view.

Note Added in Print

This book is a slight revision of the author's Ph.D. thesis [Wil00].

Acknowledgements

I would like to thank my two supervisors at the Technical University of Denmark (now both with the IT University of Copenhagen), Associate Professors Henrik Reif Andersen and Henrik Hulgaard, for giving me the opportunity to work on this project. During my graduate studies, they have guided me, given me valuable comments and criticism on my work, and patiently answered my many questions. For this I am them grateful.

A special thanks is due to Professor Edmund Clarke from Carnegie Mellon University. I spent six months working with him and his model checking group in Pittsburgh. His vast knowledge of computer science and the inspiring atmosphere surrounding him and his group made it a joy to work with him.

Thanks are also due to *Chargé de Recherches* Antoine Rauzy from *Centre National de la Recherche Scientifique*, Associate Professor David Sherman and Assistant Professor Macha Nikolskaïa from *Université Bordeaux I*. During David and Macha's visit to Lyngby and my subsequent visit to Bordeaux, we worked together and became friends in the process.

During my research, I have had long and fruitful discussions with fellow doctoral students. Especially Jørn Lind-Nielsen from Technical University of Denmark and Anubhav Gupta from Carnegie Mellon University have lent an ear, made helpful suggestions, and answered my numerous questions. Rune Møller Jensen, Ken Friis Larsen, Jakob Lichtenberg, Jesper Møller and many others have also been helpful.

I am indebted to the members of my Ph.D. committee: Associate Professor Hans Henrik Løvengreen from Technical University of Denmark, Professor Parosh Abdulla from Chalmers University, and Nils Klarlund from AT&T in New Jersey. Thank you for valuable comments and criticism on my thesis.

Contents

List of Algorithms and Figures

List of Tables

Symbols

The following notation is used throughout this dissertation:

Notation	Description	Reference
	LOGIC	
0	False	
1	True	
a, b, c	Boolean variables	
x, y	Boolean variables or vectors of Boolean variables	
ϕ, θ	Formulae	
f, g, h	Formulae	
$\phi[\theta/x]$	Substitution of formula θ for variable x in ϕ	
$\neg$	Negation	
$K0$	Connective, constant 0	Table 2.1
$\wedge$	Connective, conjunction	Table 2.1
$\not\rightarrow$	Connective, negated implication	Table 2.1
π_1	Connective, projection on first argument	Table 2.1
$\not\leftarrow$	Connective, negated left-implication	Table 2.1
π_2	Connective, projection on second argument	Table 2.1

(continued on next page)

Notation	Description	Reference
$\oplus$	Connective, exclusive or	Table 2.1
$\vee$	Connective, disjunction	Table 2.1
$\bar{\vee}$	Connective, negated disjunction	Table 2.1
$\leftrightarrow$	Connective, biimplication	Table 2.1
$\bar{\pi}_2$	Connective, negation of second argument	Table 2.1
$\leftarrow$	Connective, left-implication	Table 2.1
$\bar{\pi}_1$	Connective, negation of first argument	Table 2.1
$\rightarrow$	Connective, implication	Table 2.1
$\bar{\wedge}$	Connective, negated conjunction	Table 2.1
$K1$	Connective, constant 1	Table 2.1
$x \rightarrow \phi, \theta$	If-then-else operator	Equation 2.1
$\odot$	Any binary Boolean connective	
$\exists$	Existential quantification	
$\forall$	Universal quantification	
SAT$(\cdot)$	Satisfiability problem	Definition 1.2.2
TAUT$(\cdot)$	Tautology problem	Definition 1.2.3
Ξ_S	Characteristic function for set S	Definition 1.2.5
	BED	
0	Zero terminal	Section 2.1
1	One terminal	Section 2.1
u, v, l, h	Vertices	
$low(v)$, $high(v)$	Low and high attributes of vertex	Definition 2.1.1
$var(v)$, $op(v)$	Variable and operator attributes of vertex	Definition 2.1.1
α	Either a variable or an operator. $\alpha(v)$ is short for either $var(v)$ or $op(v)$.	
f^v	Boolean function defined by vertex	Definition 2.1.2
$sup(v)$	Support of BED	Definition 2.1.5

(continued on next page)

Notation	Description	Reference
$\|v\|$	Size of BED	Definition 2.1.6
BED for ϕ	The BED obtained by a straightforward translation of formula ϕ	Definition 2.1.8
BED u	The BED rooted at vertex u	Section 2.1
$<$	Ordering of variables or vertices	
$depth(v)$	The depth of a vertex v	Definition 3.3.1
$u \rightsquigarrow v$	Path from vertex u to vertex v	Definition 2.1.7
	STÅLMARCK	
$\sim$, R	Equivalence relation	Section 3.4
k	Saturation depth	Section 3.4
$\mathcal{C}_\sim$	Set of axioms from equivalence relation $\sim$	Equation 3.2
$\mathcal{A}_u$	Set of rules from vertex (triplet) u	Section 3.4
	HIERARCHICAL CIRCUITS	
$HCC(C, c)$	Hierarchical combinational circuit. It consists of a set of cells C and a top cell $c \in C$.	Definition 4.2.1
c	Cell	Definition 4.2.2
i	Instantiation of a cell	Definition 4.2.3
s,t,u,v,x,y	Variables or vectors of variables	
p	Path in a container cell	Definition 4.2.4
K	Cut	Definition 4.2.5
H	Cut-relation	Definition 4.2.6
R_{in}	Input relation	Equation 4.2
R_{out}	Output relation	(Equation 4.2)
$[Map]$	Renaming of variables	

(continued on next page)

Notation	Description	Reference
	MODEL CHECKING	
M	Finite state machine	Definition 5.2.2
T	Transition function / relation	Definitions 5.2.1 and 5.2.2
I	Set of initial states	Definition 5.2.2
X	Set of inputs	Definition 5.2.2
S	Set of states	Definition 5.2.2
s	State	
x	Input	
ℓ	Labeling of states	Definition 5.2.2
$\mathcal{A}$	Set of atomic propositions	Definition 5.2.2
$s^i \overset{x}{\rightsquigarrow} s^j$	Transition from state s^i to state s^j on input x	Definition 5.2.3
$s^i \rightsquigarrow_k s^j$	Path of length k from state s^i to state s^j	Definition 5.2.4
$M \models \phi$	System M models specification ϕ	Definitions 5.2.5 and 5.2.14
$[\![\phi]\!]$	Semantics of CTL formula ϕ	Definition 5.2.10
$[\![\phi]\!]_k$	Similar to $[\![\phi]\!]$ but with exactly k iterations in each fixed point computation	Definition 5.5.4
$[\phi]_k$	Similar to $[\![\phi]\!]_k$ but with input variables still in the formulae	Definition 5.5.7
$\phi =_s \theta$	CTL formulae ϕ and θ are s-equivalent	Definition 5.5.9
	FAULT TREES	
x	Boolean variable	Section 7.3
L, X	Sets of variables	Section 7.3
$x.L$	Union of sets $\{x\}$ and L	Section 7.3
π	Product of only positive literals	Section 7.3
π^c_X	Minterm for formula f obtained by adding to π the negative literals formed over all the variables in f but not π	Section 7.3

(continued on next page)

Notation	Description	Reference
$\Pi(f)$	Set of minimal p-cuts for f	Section 7.3
$\Pi_k(f)$	Set of minimal p-cuts for f with at most k literals in length	Section 7.3
	MISCELLANEOUS	
$\mathbb{B}$	The set $\{0, 1\}$	
$\mathbb{R}$	The set of all real numbers	
$\mathbb{Z}$	The set of all integers	
$\mapsto$	Mapping	
$\leftarrow$	Assignment	
$\mathcal{P}(\cdot)$	Power set	
τ	Set transformer	Definition 5.2.7

Chapter 1

Introduction

Once I gave a seminar at a company about my research. I presented the combinational circuits in Figure 1.1, and asked the question whether the two circuits were equivalent?[1] I demonstrated my formal verification technique

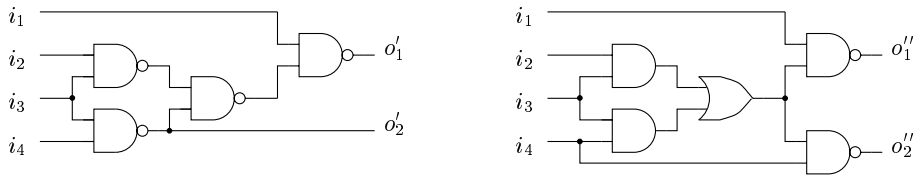

Figure 1.1: Two equivalent combinational circuits.

by proving that output o_1' and output o_1'' are equivalent. After the seminar a couple of people from the audience came to me and said that the second pair of outputs were not equivalent. They argued that output o_2' only depends on inputs i_3 and i_4, while output o_2'' depends on i_2, i_3, and i_4. Their argument is correct. However, a more careful analysis of output o_2'' shows that the value of input i_2 is masked by the circuitry and therefore cannot influence the value of the output. Thus the two circuits are indeed equivalent.

Stories like this one show that human reasoning is prone to errors. We cannot always rely on our intuition. We need a systematic way of handling these tasks such that we know for sure that we have covered all possibilities and left nothing out.

Another complicating factor is the complexity of the problems we want

[1] For the purpose of simplicity, we chose to ignore transients, glitches, and other real-life phenomena in circuits.

to solve. Again the equivalence checking problem is a good example. Each input can be either high or low. With four inputs, we have $2^4 = 16$ input combinations, which is no big deal as we can just handle each combination by itself. A larger circuit may have a hundred inputs. This leads to $2^{100} = 1,267,650,600,228,229,401,496,703,205,376$ input combinations. Or more than a thousand billion billion billion combinations. Such a large number of combinations cannot be handled one by one.

If we cannot try all the input combinations to see if they result in the expected behavior for our system, how can we be certain that there are no errors? One of the untested input combinations may provoke an erroneous behavior of the system. But if we do not try that input combination, we never find the error.

Mathematicians have solved similar problems for centuries. Consider the mathematical theorem which says that for real numbers, multiplication distributes over addition:

$$a \cdot (b + c) \;=\; a \cdot b + a \cdot c\,.$$

The variables a, b and c are abstractions of real numbers. Mathematicians have proven that for any value of the three variables, the above equation holds. Nobody would, or could, try all the infinitely many possible values of a, b and c.

In much the same way we can recognize structure in the systems we deal with. For example, we may recognize that one input i of the 100 inputs to a circuit is only used if another input j is high. Otherwise i is ignored. There is therefore no need to consider input combinations which differ only in i when j is low.

Given, for example, the two combinational circuits in Figure 1.1, how do we proceed with proving that they are equivalent? What should our overall strategy be? Consider Figure 1.2. It shows the real and the formal worlds. Given a real world problem, we want to find a real world solution. This is indicated by edge 1. However, in practice it is hard to solve problems in the real world. A possibility is to formalize the problem (edge 2). A formal problem is one described in mathematics and logic. Staying in the formal world, we can solve the formal problem (edge 3), and transform the formal solution back to a real solution (edge 4).

The route 2-3-4 might seem as a detour over just taking edge 1. But going via the formal world, we can reason about our methods, and we have a set of mathematical and logical tools available.

In this dissertation we stay exclusively in the formal world. We assume that the transitions between the two worlds (edges 2 and 4) are either triv-

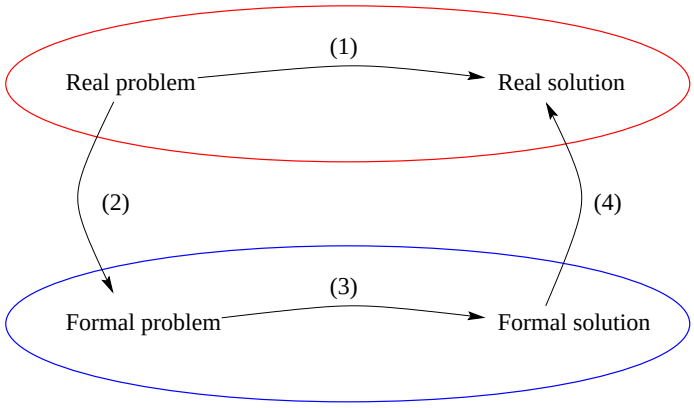

Figure 1.2: The real and the formal worlds illustrated by the two circles. The edges indicate how to solve a real world problem by either staying in the real world (edge 1) or going via the formal world (edges 2, 3, and 4).

ially simple or that someone else takes care of them. This does not mean that those transitions are uninteresting. On the contrary, they are very interesting. The whole idea of how to formalize a problem, including deciding which parts to abstract away and which parts to model, is a major research area these days.

1.1 Formal Verification

The title of this dissertation is "Formal Verification Based on Boolean Expression Diagrams". In this section we define what we mean by formal verification. According to Merriam-Webster's dictionary[2], the word "verification" means "the act or process of verifying" and "to verify" means "to establish the truth, accuracy, or reality of." The word "formal" means "relating to or involving the outward form, structure, relationships, or arrangement of elements rather than content." So formal verification is the process of establishing the truth using outward form, structure, relationships, or arrangement of elements rather than content.

To get a workable definition of formal verification, we propose "the act of proving whether a system has a given property." For this definition to be complete, we need to specify what we mean by system, by property, and by proving (or proof):

[2]See `http://www.m-w.com`.

- A *system* is a mathematical or logical description of something that we want to examine.
- A *property* is a mathematical or logical statement about a *system*.
- Given a *system*, a *proof* of a *property* is a sequence of valid mathematical or logical reasonings which establishes that the *system* has the *property*.

Sometimes the property is called the specification. The following section describes the logic we need as the foundation for the reasonings in this dissertation.

1.2 Logic

The word "logic" stems from *logos*, the Greek word for reason. Propositional logic is the reasoning of propositions. A proposition is a statement that is either true or false; for example, "the sun is shining" or "4 is prime". We use 0 to mean false and 1 to mean true, and we use Boolean variables to represent basic propositions. These constants and variables are our atomic formulae. Atomic formulae can be connected using Boolean connectives forming compound formulae. There are two connectives of one argument: negation and projection. We only use negation, and we write it $\neg$, where negation is defined as $\neg 0 = 1$ and $\neg 1 = 0$. There are 16 Boolean connectives of two arguments. Not all 16 Boolean connectives are necessary. For example, it is enough to have only negation and disjunction since the remaining 14 connectives can be constructed in terms of those two. One can think of the remaining Boolean connectives as syntactic sugar.

Definition 1.2.1. A formula in propositional logic can be generated from the following grammar:

$$f \ ::= \ 0 \mid 1 \mid \mathit{variable} \mid \neg f \mid f \vee f \,.$$

A variable assignment is an assignment of either 0 or 1 to each variable in a set of variables. Typically the set is a singleton set or the set of all variables in a formula. In these cases we refer to a variable assignment for a variable or for a formula instead of for the corresponding sets. Given a variable assignment for formula ϕ, we can evaluate ϕ to either 0 or 1 by replacing all variables in ϕ with their assigned value and then use the truth tables of the operators to propagate the constants to the top of the formula.

A formula is said to be a *tautology* if it evaluates to 1 for all possible variable assignments. Likewise, a formula is said to be a *contradiction* if it evaluates to 0 for all possible variable assignments. We say that a contradiction is *unsatisfiable* since no variable assignment makes it evaluate to 1. A formula which is *not* a contradiction is *satisfiable*.

We now define two related problems in propositional logic.

Definition 1.2.2 (Satisfiability Problem). Let ϕ be a formula in propositional logic. Determine whether a variable assignment exists for ϕ, such that ϕ evaluates to 1 for this assignment.

Definition 1.2.3 (Tautology Problem). Let ϕ be a formula in propositional logic. Determine if ϕ evaluates to 1 for all possible variable assignments.

We use $\mathrm{SAT}(\phi)$ to denote the function that is 1 if ϕ is satisfiable and 0 otherwise. Likewise, $\mathrm{TAUT}(\phi)$ denotes the function that is 1 if ϕ is a tautology and 0 otherwise. Note that $\mathrm{SAT}(\phi) = \neg\mathrm{TAUT}(\neg\phi)$.

Propositional logic can be extended to quantified Boolean formulae (QBF) by introducing the existential quantifier $\exists$:

Definition 1.2.4. A formula in QBF can be generated from the following grammar:

$$f \ ::= \ 0 \mid 1 \mid \mathit{variable} \mid \neg f \mid f \vee f \mid \exists \ \mathit{variable} : f \, .$$

The semantics of the existential quantifier is

$$\exists x : \phi \ \equiv \ \phi[0/x] \ \vee \ \phi[1/x] \, , \tag{1.1}$$

where $\phi[b/x]$ means a substitution of b for x in ϕ. The universal quantifier $\forall$ can be obtained from the existential quantifier using negation:

$$\forall x : \phi \ \equiv \ \neg\exists x : \neg\phi \ \equiv \ \phi[0/x] \ \wedge \ \phi[1/x] \, . \tag{1.2}$$

A variable is said to be *free* in formula ϕ if it is not bound by a quantifier. Note that solving the satisfiability (tautology) problem for ϕ corresponds to adding existential (universal) quantifiers for all free variables in ϕ and expanding the resulting QBF to a propositional logic formula using (1.1) and (1.2). The resulting propositional logic formula contains no variables and can easily be reduced to either 0 or 1 using the truth tables for the operators.

Computation Tree Logic (CTL) is a temporal logic used to describe the specification of a finite state machine. In Chapter 5 we describe both CTL and finite state machines in detail.

In the rest of this dissertation we often encode sets in propositional logic. We use the term *characteristic function* for the function encoding a set. Using characteristic functions it is often possible to greatly reduce the memory needed to represent a set. Another advantage is that characteristic functions allow us to work on the whole set as opposed to working on the elements of a set one at a time.

Definition 1.2.5 (Characteristic Function). Let S be a set. The *characteristic function* $\Xi_S : S \mapsto \mathbb{B}$ for S is given by:

$$\Xi_S(s) = \begin{cases} 1 & : \text{ if } s \in S \\ 0 & : \text{ otherwise} \end{cases}$$

Since characteristic functions are used extensively, we often omit the Ξ and just mention that a set is represented by its characteristic function.

The following example illustrates characteristic functions. Assume we want to represent sets of integer numbers $\{0, 1, 2, 3, 4, 5, 6, 7\}$. Using three bits $\langle s_2 s_1 s_0 \rangle$ we can represent all eight numbers using their binary representation such that $\langle s_2 s_1 s_0 \rangle = \langle 000 \rangle$ represents the number 0, $\langle 001 \rangle$ the number 1, and so on up to $\langle 111 \rangle$ for the number 7. Now, the characteristic function:

$$\Xi = s_1 \wedge \neg s_0$$

represents the set $\{2, 6\}$ because the encoding for two ($\langle 010 \rangle$) and six ($\langle 110 \rangle$) are the only numbers to have s_1 set to true and s_0 set to false.

1.3 Aim of This Dissertation

In this dissertation we look at ways to solve problems in the domain of formal verification. We want our solutions to be systematic so they can be implemented on a computer. We also want our solutions to be able to deal with complex problems as they occur often in industry.

The basic guideline throughout this research is the use of a data structure called Boolean Expression Diagrams. Our aim is to apply this data structure to formal verification problems. We have chosen to concentrate on the following problems within formal verification:

- Equivalence checking of combinational circuits,

- Model checking of transition systems, and
- Fault tree analysis.

Equivalence checking is the problem of determining whether two combinational circuits implement the same Boolean functions. The problem arises in a number of CAD applications related to validating the correctness of a circuit design. Design automation tools are used to manipulate circuits. The circuits may also be manually modified. Figure 1.1 is an example of two such circuits. To ensure that no errors are introduced, we can check that the circuits before and after such manipulations are equivalent. The equivalence checking problem also occurs as a subproblem of other verification problems. For example, when verifying arithmetic circuits by checking that they satisfy a given recurrence equation [Fuj96] or when verifying the equivalence of two state machines without performing a state traversal [vE98].

Model checking is the problem of determining whether a system satisfies its temporal specification. Like the equivalence checking problem, the model checking problem arises in a number of CAD applications like design of digital circuits and communication protocols. For example, an electronic system controls a four-way traffic light intersection. We want to know if it is always the case that we have red light in at least one direction. This is a temporal specification and we can check whether our traffic light system satisfies it.

Fault tree analysis is the problem of calculating certain values based on a fault tree for a system. A fault tree is a Boolean function describing the conditions under which the system fails based on the condition ("failure" or "working") of each of the components. Examples are nuclear power plants and airplanes. For both kinds of systems it is important to keep the probability of failure down.

These three problems represent different areas within formal verification.

- In equivalence checking we compare two objects of the same kind: combinational circuits. One circuit takes the role of the system, the other takes the role of the property. We use propositional logic to describe both.
- In model checking we compare two different kinds of objects: A finite state machine and a CTL specification. We encode the finite state machine in propositional logic. Based on the CTL specification, we compute a set of states which are valid initial states for the finite state machines. Finally we compare this set of states with the actual initial states.

- In fault tree analysis we compare a value to a set of acceptable values. The fault tree is the system. We describe it in propositional logic. We consider the property to be a set of numbers which are acceptable as failure probabilities. The verification task is then to compute the probability of a system failure given the failure probabilities of each component.

There are, of course, other areas within formal verification than the ones we deal with in this dissertation. Gupta has written a thorough survey of formal verification methods with respect to hardware [Gup92]. We do not hesitate to recommend her paper to readers interested in getting an overview of formal verification. Clarke and Wing have written a paper on the state-of-the-art and future directions for formal methods [CW96]. It contains a wealth of references to examples where formal methods (including formal verification) have been applied with success.

1.4 Safety and Reliability

In the previous sections we have presented formal verification as a means to ensure that a system has a property. By going via the formal world and using techniques based on mathematics and logic, we can completely ensure that our systems are correct.

Or can we?

The answer is, unfortunately, no. We cannot completely ensure correctness of a system — at least not if we by "correct" imply that the system is safe and reliable. The problem is twofold:

First of all, we have no guarantee that the property we verify is the correct one. It may be that the property holds, but we never consider another property which is also critical for the system. Think of the four-way traffic light intersection example. We verify that we have red light in at least one direction at all times. Assume that our particular traffic light intersection has this property. Is it a correct, safe and reliable intersection? No, not necessarily. We have, for example, not verified that the lights actually change. A bug in our system may cause the traffic lights to show red in both directions at all times. This is naturally not a correct behavior of a traffic light intersection, but our original property did not capture this error.

Second, when verifying a system, we implicitly assume that it is isolated from the context in which it is to function. We verify, so to say, a stand-alone version of the system. However, as Dr. Leveson points out [Lev99], many of the failures of complex systems today arise in the *interfaces* between the

components, where the components may be hardware, software or human. A typical error is "mode confusion" where the assisting computer is in one mode but the human believes it to be in another mode. For example, an aircraft control computer may be in "flight mode", but the operator believes it to be in "landing mode". While the computer works correctly by itself, the interface between the computer and the pilot causes problems.

There is more to obtaining correct systems than to formally verify them. Good design methodologies and testing of the final products catch errors. Knowledge of psychology and cognitive engineering is also important to avoid interface errors.

In this dissertation we only consider formal verification. We want, however, to stress that formal verification is not *the* solution to obtaining correct systems. It is one among several methods; each of which has strengths and weaknesses. Ideally, one should apply a range of such methods.

1.5 Overview

Chapter 2 introduces Boolean Expression Diagrams as a data structure for representing and manipulating Boolean formulae. We explain how to implement the data structure. The chapter gives a number of properties for Boolean Expression Diagrams. Finally, the chapter contains a number of algorithms for working with Boolean Expression Diagrams – especially for constructing them and for converting them to Binary Decision Diagrams.

In Chapter 3 we look at how Boolean Expression Diagrams can be used in equivalence checking of flat combinational circuits. The idea is to model the circuit outputs as Boolean formulae over the inputs. Two circuits are equivalent if their output formulae are pairwise equivalent. We use Boolean Expression Diagrams to represent the formulae. The equivalence checking problem can then be viewed as an instance of the tautology problem $\mathrm{TAUT}(\phi_1 \leftrightarrow \phi_2)$, where ϕ_1 and ϕ_2 are the formulae for a pair of corresponding outputs for the two circuits in question. We consider a number of ideas including simplification of the Boolean Expression Diagrams, variable ordering heuristics, and SAT-procedures. The ideas are evaluated on a large set of combinational circuits. This chapter is based on the papers [HWA97] and [HWA99]:

[HWA97] H. Hulgaard, P. F. Williams, and H. R. Andersen. Combinational logic-level verification using boolean expression diagrams. In *3rd International Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, September 1997.

[**HWA99**] H. Hulgaard, P. F. Williams, and H. R. Andersen. Equivalence checking of combinational circuits using boolean expression diagrams. *IEEE Transactions on Computer Aided Design*, July 1999.

Where the previous chapter dealt with flat circuits, Chapter 4 considers combinational circuits described in a hierarchical or modular way. Such circuits may be viewed as a number of cells, where each cell may have one or more instantiations. We have devised a method for utilizing the hierarchical structure of these circuits. The goal is to avoid constructing formulae for the functionality for whole circuits. Instead, we aim to only represent relations between circuits. The idea of talking about relations between circuits and not the functionality of the circuits fits well with equivalence checking. Here we assume that the two circuits have pairwise equivalent inputs and we verify that it leads to pairwise equivalent outputs. In both cases we relate the two circuits instead of talking about their functionality. This chapter is based on the paper [WHA99]:

[**WHA99**] P. F. Williams, H. Hulgaard, and H. R. Andersen. Equivalence checking of hierarchical combinational circuits. In *IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, September 1999.

In Chapter 5 we discuss symbolic model checking. We verify whether temporal logic specifications hold for finite state machines. The mathematics behind the verification is well known. Typically, symbolic model checking is done using Binary Decision Diagrams as the underlying data structure. In this chapter we replace the Binary Decision Diagrams with Boolean Expression Diagrams. This has some consequences, both positive and negative. For example, one of the consequences is that with Boolean Expression Diagrams instead of Binary Decision Diagrams we shift the complexity from constructing the diagrams to showing semantical equivalence between two diagrams. We discuss these consequences and show how to deal with them. This chapter is partly based on the paper [WBCG00]:

[**WBCG00**] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, Chicago, U.S.A., pages 124–138, July 2000. Springer-Verlag.

Chapter 6 discusses how to determine satisfiability using the Boolean Expression Diagram data structure. Most SAT-solvers today require that

the input formula is in conjunctive normal form (CNF). However, most problems in formal verification are not naturally described in CNF and it is therefore necessary to convert the formulae into CNF. The conversion is expensive as it either enlarges the state space by adding extra variables or results in an explosion in the size of the CNF representation. By doing satisfiability checking directly on the Boolean Expression Diagram data structure, we eliminate the conversion to CNF. The chapter is based on the paper [WAH01]:

[WAH01] P. F. Williams, H. R. Andersen, and H. Hulgaard. Satisfiability checking using boolean expression diagrams. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, 2001.

Chapter 7 extends the Boolean Expression Diagram data structure. We introduce quantification and substitution as part of the data structure. As an example of a more complex extension, we add a *p-cut* operator for fault tree analysis. Using this *p-cut* operator, we are able to deal with fault trees from the industry in a more efficient way than using standard methods. The work on fault tree analysis is based on the paper [WNR00]:

[WNR00] P. F. Williams, M. Nikolskaïa, and A. Rauzy. Bypassing BDD construction for reliability analysis. *Information Processing Letters*, 75(1-2):85–89, July 2000.

Chapter 8 contains the conclusions. We give an outline of the results we have obtained. We characterize both the problems on which our methods work well and the problems on which out methods do not work well. Finally, we identify topics for future research.

In order to examine the Boolean Expression Diagram data structure, we have made an implementation in the programming language C. It is a set of library routines for constructing and manipulating Boolean Expression Diagrams. On top of the library, we have built a shell-like interface. Here the user can interactively enter, manipulate and examine Boolean Expression Diagrams. Appendix A describes this interface. The library and the shell interface form the core with which the experiments in this dissertation have been performed. Both are available on online[3].

[3] See `http://www.it-c.dk/research/bed` for more information.

Chapter 2

Boolean Expression Diagrams

In 1997, Andersen and Hulgaard proposed a new data structure for representing and manipulating Boolean formulae [AH, AH97]. The data structure is called Boolean Expression Diagrams, or BEDs for short. It is a generalization of Bryant's Binary Decision Diagrams (BDDs) [Bry86, Bry92]. In this chapter we present the BED data structure, its properties, and the algorithms for working with it. Part of this chapter is a review of Andersen and Hulgaard's work.

2.1 Data Structure

A Boolean Expression Diagram is a data structure for representing and manipulating Boolean formulas.

Definition 2.1.1 (Boolean Expression Diagram). A *Boolean Expression Diagram* (BED) is a rooted directed acyclic graph $G = (V, E)$ with vertex set V and edge set E. The vertex set V contains three types of vertices: terminal, variable, and operator vertices.

- A *terminal vertex* v has as attribute a value $val(v) \in \{0, 1\}$.
- A *variable vertex* v has as attributes a Boolean variable $var(v)$, and two children $low(v), high(v) \in V$.
- An *operator vertex* v has as attributes a binary Boolean operator $op(v)$, and two children $low(v)$, $high(v) \in V$.

The edge set E is defined by

$$E \;=\; \left\{(v, low(v)), (v, high(v)) \;\middle|\; v \in V \text{ and } v \text{ is a non-terminal vertex } \right\}.$$

We identify a BED by its root vertex. For example, let u be a BED vertex. We then use the term "the BED u" to refer to the BED rooted at vertex u.

We use **0** and **1** to denote the two terminal vertices. The relation between a BED and the Boolean function it represents is straightforward. Terminal vertices correspond to the constants 0 and 1. Variable vertices have the same semantics as vertices of BDDs and correspond to the *if-then-else* operator $x \to f_1, f_0$ defined by

$$x \to f_1, f_0 \;=\; (x \wedge f_1) \vee (\neg x \wedge f_0)\,. \tag{2.1}$$

Operator vertices correspond to their respective Boolean connectives, see Table 2.1. This leads to the following correspondence between BEDs and Boolean functions:

Definition 2.1.2 (Semantics). A vertex v in a BED denotes a Boolean function f^v defined recursively as:

- If v is a terminal vertex, then $f^v = val(v)$.
- If v is a variable vertex, then $f^v = var(v) \to f^{high(v)}, f^{low(v)}$.
- If v is an operator vertex, then $f^v = f^{low(v)}\ op(v)\ f^{high(v)}$.

The unary operator negation is not part of the BED definitions. Negation can be obtained by using the $\bar{\pi}_1$ operator with a dummy second argument. For readability, we use $\neg$ for negation in BEDs instead of $\bar{\pi}_1$.

Definition 2.1.3 (Reduced). A BED is called *reduced* if it has the following properties:

- No two vertices are identical, i.e, they have the same attributes.
- No variable or operator vertex has two identical children.
- No operator vertex has a terminal child.

Definition 2.1.4 (Free). A BED is called *free* if on any path from the top vertex to a terminal vertex we encounter at most one instance of every free variable.

op	$\frac{op(x,y)}{\frac{x:0011}{y:0101}}$	Name of Boolean connective
$K0$	0000	Constant 0
$\wedge$	0001	Conjunction
$\not\rightarrow$	0010	Negated implication
π_1	0011	Projection on first argument
$\not\leftarrow$	0100	Negated left-implication
π_2	0101	Projection on second argument
$\oplus$	0110	Exclusive or
$\vee$	0111	Disjunction
$\bar{\vee}$	1000	Negated disjunction
$\leftrightarrow$	1001	Biimplication
$\bar{\pi}_2$	1010	Negation of second argument
$\leftarrow$	1011	Left-implication
$\bar{\pi}_1$	1100	Negation of first argument
$\rightarrow$	1101	Implication
$\bar{\wedge}$	1110	Negated conjunction
$K1$	1111	Constant 1

Table 2.1: The 16 binary Boolean connectives and their truth tables.

We assume that all BED data structures are reduced and free as per Definition 2.1.3 and 2.1.4.

Definition 2.1.5 (Support). The *support* of a BED u, written $sup(u)$, is the set of free variables[1] in f^u.

Definition 2.1.6 (Size). The *size* of a BED u, written $|u|$, is the number of vertices in u.

Definition 2.1.7 (Path). There is a *path* from vertex u to vertex v, and we write $u \rightsquigarrow v$, if there exists a finite sequence of vertices $\langle u_1, u_2, \ldots, u_n \rangle$ $(n \geq 1)$ such that:

- $u = u_1$
- $v = u_n$
- For all $i = 1, \ldots, n-1$, either $u_{i+1} = low(u_i)$ or $u_{i+1} = high(u_i)$

It is convenient to talk about the BED for a formula. We use this terminology to mean the BED defined in Definition 2.1.8:

Definition 2.1.8 (BED for Formula). Given a propositional formula f, the *BED for formula* f is the BED representing the same Boolean function as the formula, such that:

- Each Boolean connective in formula f corresponds to an operator vertex in the BED. The low (high) child of the vertex corresponds to the BED for the formula for the left (right) argument of the operator.
- Each Boolean variable in the formula corresponds to a variable vertex in the BED with low child **0** and high child **1**.
- 0 and 1 in the formula correspond to **0** and **1** in the BED.

As an example, Figure 2.1 shows a BED for the formula $a \leftrightarrow a \wedge (a \vee b)$. The BED is both reduced and free. The support is $\{a, b\}$. The size if 7.

The implementation of BEDs is inspired by the BDD implementation described in [BRB90]. The internal data structure is an array. Each entry in the array represents a vertex and has the fields *op*, *var*, *low*, and *high*

[1] All variables are free in a free BED. However, we later introduce quantifiers to the BED and quantified variables are not free.

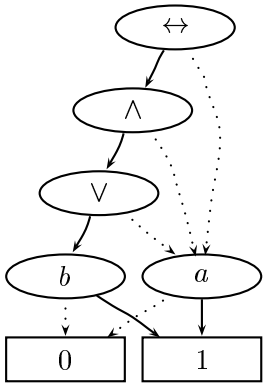

Figure 2.1: The BED for $a \leftrightarrow a \wedge (a \vee b)$. The dotted edges are the low ones.

corresponding to the operator, variable, low child, and high child attributes of a vertex. Each vertex is identified by it position in the array. Position 0 and 1 correspond to the vertices **0** and **1**. To find a vertex given its attributes, we use a hash table. For convenience we place the hash table in the same array as the BED vertices. To do this we add two extra fields to each entry in the array: H and $next$. The H field in entry n contains the index of the vertex with hash value n. The $next$ field is used to resolve collisions. To find a vertex $(\alpha, low, high)$, where α is either a variable or an operator, we compute a hash value h for it. If the vertex exists in the BED array, it is found in entry $H(h)$ or in the chain of vertices pointed to by $next(H(h))$, $next(next(H(h)))$, and so on until a null-pointer is reached.

Some algorithms require marking vertices. We therefore include a marking field $mark$ as well. We group the fields $mark$, var and op in one memory word. The remaining fields each require one word. On a 32 bit machine architecture, each vertex takes up 20 bytes of memory (5 words of 4 bytes each). Figure 2.2 shows the BED array. Vertex 6 represents the formula $a \leftrightarrow a \wedge (a \vee b)$. This is the same formula as for the BED in Figure 2.1.

Vertices are created by the MK algorithm; see Algorithm 2.3. The call $\text{MK}(\alpha, l, h)$ returns the identity of a (α, l, h) vertex. If α is a variable, then (α, l, h) is a variable vertex with variable α, low child l and $high$ child h. If α is an operator, then (α, l, h) is an operator vertex with operator α and low and $high$ children l and h. Lines 1 through 6 handle the three requirements of reduced BEDs; see Definition 2.1.3. Line 8 creates a new vertex.

Using MK as the only means of creating vertices ensures that the BEDs are always reduced. If we do not create variable vertices with non-terminal children, then the resulting BEDs are always free. Hence, the BED for a formula is both reduced and free.

The BED array does not contain a field for reference counting of the

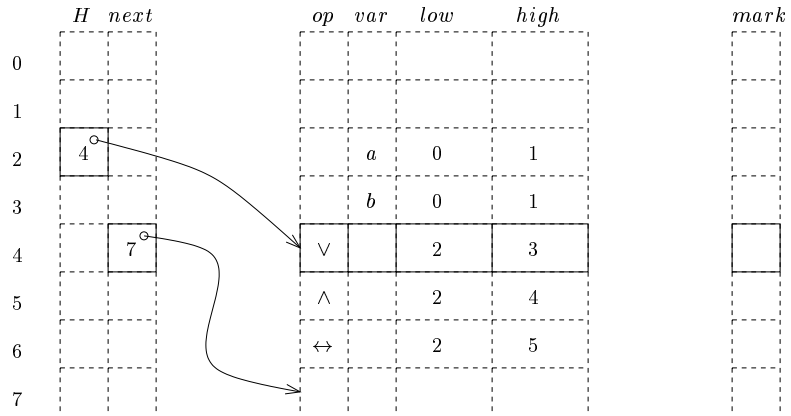

Figure 2.2: Implementation of the BED data structure as an array. Looking up a vertex $v = (\alpha, low, high)$ is done like this: The hash value of v, say 2, is used as an entry point in the array. The H field points to a vertex, here 4. If this vertex is not v then we follow the pointer in the *next* field of vertex 4, which in this case leads to entry 7. This is repeated until either the vertex is found or we reach a null-pointer.

vertices. Once a vertex is created, it stays in the data structure. When the data structure is full, we perform a garbage collection by sweeping through the whole BED array and marking all the vertices which are still in use. The remaining vertices are removed and their corresponding entries in the array are freed. Unused vertices are placed on a free-list, which we implement as a linking through the *low* field of all entries not in use.

2.2 Properties

In the rest of this dissertation we often relate BEDs to Binary Decision Diagrams (BDDs) [Bry86, Bry92]. We therefore start by defining what a BDD is:

Definition 2.2.1 (Binary Decision Diagrams). A Binary Decision Diagram (BDD) is a BED with only terminal and variable vertices.

By defining BDDs in terms of BEDs, we already have the semantics (Definition 2.1.2) for BDDs. Like for BEDs, we assume all BDDs are both reduced and free.

BDDs are often restricted in some way. A common restriction is to require an ordering of the variables:

Name: MK(α, l, h)
1: **if** there exists a (α, l, h) vertex **then**
2: **return** that vertex
3: **else if** α is a variable and $l = h$ **then**
4: **return** l
5: **else if** α is a Boolean connective and either l and h are identical or one of them is a terminal **then**
6: **return** either **0**, **1**, l, h, MK$(\neg, l, \cdot)$ or MK$(\neg, h, \cdot)$
7: **else**
8: **return** new vertex (α, l, h)

Algorithm 2.3: The MK algorithm. The algorithm takes a variable or operator α and two BEDs l and h as arguments and returns a BED vertex with variable or operator α, *low* child l and *high* child h. The two dots $(\cdot)$ in line 6 indicate dummy second arguments as explained on page 14.

Definition 2.2.2 (Ordered Binary Decision Diagrams). A BDD is called *ordered* if on all paths the variables respect a given ordering $<$.

This restriction is so common that we assume all BDDs are ordered unless otherwise noted.

It is clear that since a BDD is a special case of a BED, the latter has the expressive power of at least that of the former. Any extra power of a BED must stem from the operator vertices. However, Bryant proposed in [Bry86, Bry92] an algorithm, APPLY, for connecting BDDs with Boolean connectives. Informally, any operator vertex in a BED corresponds to a call to APPLY in a BDD, and thus BEDs do not have any extra expressive power. We summarize this in Observation 2.2.3:

Observation 2.2.3. BEDs and BDDs have the same expressive powers.

We measure the size of a BED or a BDD in the number of vertices it has. Bryant [Bry86] has shown that there exists no BDD representation of a multiplier circuit, such that the BDD size is sub-exponential in the bit-width of the multiplier. However, we can build multiplier circuits using only a quadratic number of gates [CLR90]. Mapping each gate to an operator vertex, we can construct a BED of the same size. Thus, BEDs are more succinct than BDDs. We capture this in Observation 2.2.4:

Observation 2.2.4. BEDs are exponentially more succinct than BDDs.

One of the key properties of BDDs is their canonicity. Given a variable ordering, there exists one and only one BDD for a given Boolean function.

BEDs do not have this property. The BEDs for $x \wedge y$ and $y \wedge x$ are clearly semantically equivalent, but they are syntactically different.

Observation 2.2.5. BDDs are canonical; BEDs are not.

Since BEDs are not canonical, syntactical and semantical equivalence are not the same. There is the following relation between the two notions of equivalence: Syntactical equivalence implies semantical equivalence, but the reverse is not true. This is of importance in implementations of BEDs and BDDs where syntactical equivalence is the only available equivalence.

Solving the satisfiability problem for BDDs is easy. Because of the canonicity of BDDs, there is only *one* representation of an unsatisfiable function: the terminal vertex **0**. All other BDDs represent satisfiable functions. The satisfiability problem can be solved by comparing a given BDD to the BDD **0** – a constant time operation for any decent BDD implementation. Likewise, solving the tautology problem on BDDs is also a constant time operation.

In [AH, AH97], Andersen and Hulgaard note that the satisfiability problem for BEDs is NP-complete and the tautology problem is co-NP-complete. They use the close relationship between BEDs and circuits. We summarize in Observation 2.2.6:

Observation 2.2.6. For BEDs, the satisfiability problem is NP-complete and the tautology problem is co-NP-complete. For BDDs, both problems are solvable in constant time.

Assuming that it takes constant time to create a new vertex, we can construct a BED for a formula in time linear in the size of the formula. For BDDs this takes much longer. In the worst case it takes time exponential in the size of the formula.

Observation 2.2.7. BED size is linear in the size of the formula. BDD size is exponential.

2.3 Algorithms

One way to solve the satisfiability and the tautology problems using BEDs is to convert the BEDs into BDDs. As stated in Observation 2.2.6, for BDDs both problems are solvable in constant time. It might seem counterintuitive to first construct a BED and then transform it to a BDD when we could have constructed the BDD to begin with. BDDs are canonical, we might argue, and thus the end result is the same regardless of how we obtain it. The

solution to this dilemma is in the words *end result*. It is often not the size of the end result which is the limiting factor. It is the size of the intermediate results. Figure 2.4 shows a typical graph of the memory usage over time in BDD construction. The memory usage increases until it peaks, and then it

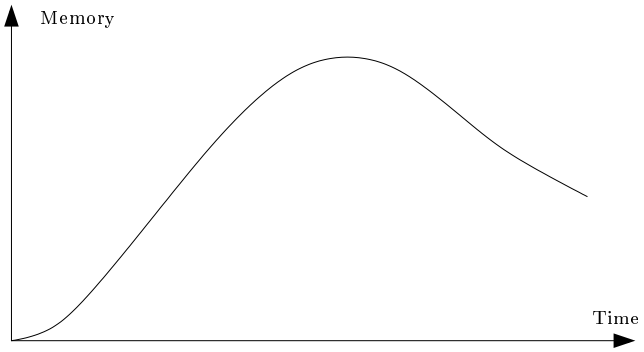

Figure 2.4: Typical memory usage over time in BDD construction.

drops. In some cases it drops to almost nothing. Consider constructing the BDD for $\phi = \theta \leftrightarrow \theta$, where the BDD for θ is large. The size of the end result is only one vertex as ϕ is a tautology. However, as an intermediate step we need to construct the BDD for θ, which is large.

The intermediate results needed to convert a BED to a BDD are not necessarily the same as the ones needed to construct the BDD to begin with. In some cases we can take shortcuts with BED to BDD conversion which we cannot do with standard BDD construction. These shortcuts, although simple, turn out to be quite effective.

Andersen and Hulgaard present in [AH, AH97] two algorithms for converting BEDs to BDDs. The algorithms are called Up_One and Up_All. The idea behind both algorithms is the following: If we remove all operator vertices from a BED, we are left with a BDD. The algorithms Up_One and Up_All also ensure that the resulting BDD is ordered[2].

The following equations form the basis of Up_One and Up_All. The

[2]Strictly speaking, the original BED must be ordered (as per Definition 2.2.2 but with "BED" instead of "BDD"), but this is not a problem since the BED for any Boolean formula (as defined in Definition 2.1.8) is always ordered. If the original BED is not ordered, it is possible to patch the algorithms such that the final BDD is indeed ordered – see [AH] for details.

equations allow local modifications to Boolean formulae.

$$\begin{aligned}(x \to f_1, f_2) \odot (x \to g_1, g_2) &= x \to (f_1 \odot g_1), (f_2 \odot g_2) \\ (x \to f_1, f_2) \odot g &= x \to (f_1 \odot g), (f_2 \odot g)\end{aligned} \tag{2.2}$$

The operator $\odot$ is any binary Boolean operator. It is straightforward to prove the validity of the equations. Split on variable x. If x is 1, all *if-then-else* operators may be replaced by their "then" argument. The left and right sides of the equations become syntactically equivalent. A similar argument holds for the case of x equal to 0. The equations also hold in the case where $\odot$ is an *if-then-else* operator, i.e., it holds for variable vertices [FMK91]. This is the basis for dynamic variable reordering of BDDs [Rud93]. Figure 2.5 shows the equations as BED transformations. Notice how the variable x is pulled one level up.

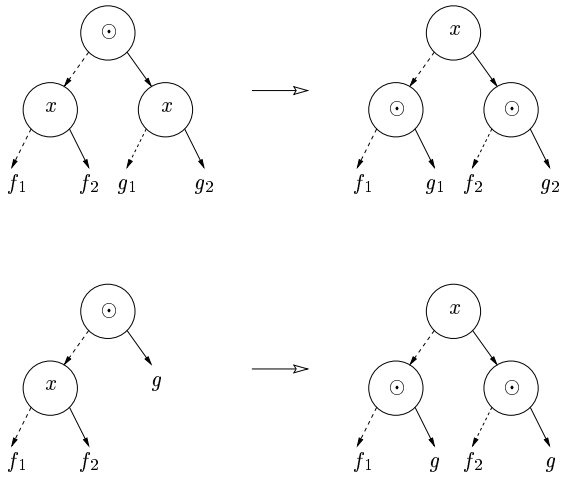

Figure 2.5: The up step from Equation 2.2 shown as BED operations. x is a variable and $\odot$ is an operator. The transformations are also valid if we replace $\odot$ with a variable.

UP_ONE works by sifting one variable to the root of the BED. Using UP_ONE repeatedly, we sift all variables to the top dividing the BED into three layers. The top layer consists of all the variable vertices. The middle layer consists of all the operator vertices. And the bottom layer consists of the terminals. BEDs, being always reduced (Definition 2.1.3), cannot contain operator vertices with terminal children. Such operator vertices are replaced with terminals according to the truth tables in Table 2.1. Thus

the resulting BED has only variable vertices and terminals; in other words, it is a BDD. Algorithm 2.6 shows the pseudo-code for UP_ONE. Repeated application of UP_ONE transforms a BED to a BDD from the top down. The table M is used to memorize previously computed results and ensures a linear expected runtime[3].

Observation 2.3.1 (Up_One). For a variable x and a BED u, let $v = \text{UP\_ONE}(x, u)$. Then UP_ONE has the following properties:

(*i*) $f^v = f^u$.

(*ii*) $|v| \leq 2|u| - 1$.

(*iii*) The running time of UP_ONE is $O(|u|)$.

Consider repeated application of UP_ONE to the variables $x_1, x_2, \ldots, x_n$, where each variable is pulled to the root. On the way to the root, variable x_i passes the variables $x_1, \ldots, x_{i-1}$. The final result is a BDD with the order $x_n, x_{n-1}, \ldots, x_1$. A lot of work goes into pulling variables up past the variables previously pulled up. The following "early stop" modification to UP_ONE would help: Pull a variable up until it is at the root of the BED or just below a previously UP_ONE'ed variable, whichever occurs first. This would result in a BDD with the order $x_1, x_2, \ldots, x_n$. And no time is wasted doing a reordering at the top of the BED.

As UP_ONE pulls a variable x to the root, it creates a lot of intermediate variable x vertices. These vertices are only used as a means to pass information between the recursive calls in UP_ONE. Algorithm 2.7 shows the pseudo-code for an optimized version of UP_ONE where intermediate vertices are not created. The algorithm is somewhat simpler than the original UP_ONE algorithm by Andersen and Hulgaard [AH97] (Algorithm 2.6). Instead of returning a vertex, UP_ONE$'$ returns a pair of vertices. The correspondence between UP_ONE and UP_ONE$'$ is:

Observation 2.3.2 (Up_One$'$). Let x be a variable and u be a vertex. Let $v = \text{UP\_ONE}(x, u)$ and $v' = \text{MK}(x, l, h)$, where $(l, h) = \text{UP\_ONE}'(x, u)$. Then v and v' are identical vertices.

For the experimental results we use a version of UP_ONE based on Algorithm 2.7 modified for early stop in repeated applications.

[3] In an implementation we use a cache instead of a table. This way we can bound the memory used for memorizing previously computed results. This comes at the expense of loosing some of the previously computed results.

Name: UP_ONE(x, u)
Require: The memorization table M is initialized to empty prior to the first call.

```
1: if (x, u) is in M then return M(x, u)
2: if u is a terminal or variable x vertex then return u
3: (l, h) ← (UP_ONE(x, low(u)), UP_ONE(x, high(u)))
4: if l and h are both variable x vertices then
5:    tmp_1 ← MK(α(u), low(l), low(h))
6:    tmp_2 ← MK(α(u), high(l), high(h))
7:    r ← MK(x, tmp_1, tmp_2)
8: else if l is a variable x vertex then
9:    tmp_1 ← MK(α(u), low(l), h)
10:   tmp_2 ← MK(α(u), high(l), h)
11:   r ← MK(x, tmp_1, tmp_2)
12: else if h is a variable x vertex then
13:   tmp_1 ← MK(α(u), l, low(h))
14:   tmp_2 ← MK(α(u), l, high(h))
15:   r ← MK(x, tmp_1, tmp_2)
16: else
17:   r ← MK(x, l, h)
18: insert ((x, u), r) in M
19: return r
```

Algorithm 2.6: The UP_ONE algorithm. UP_ONE takes a variable x and a BED u as arguments and returns a BED equivalent to u but with x pulled up to the root. $\alpha(u)$ is the "tag" $op(u)$ for operator vertices and $var(u)$ for variable vertices.

Name: UP_ONE$'(x, u)$
Require: The memorization table M is initialized to empty prior to the first call.
1: **if** (x, u) is in M **then return** M(x, u)
2: **if** u is a terminal vertex **then return** (u, u)
3: **if** u is a variable x vertex **then return** $(low(u), high(u))$
4: $(ll, lh) \leftarrow$ UP_ONE$'(x, low(u))$
5: $(hl, hh) \leftarrow$ UP_ONE$'(x, high(u))$
6: $rl \leftarrow$ MK$(\alpha(u), ll, hl)$
7: $rh \leftarrow$ MK$(\alpha(u), lh, hh)$
8: insert $((x, u), (rl, rh))$ in M
9: **return** (rl, rh)

Algorithm 2.7: The modified UP_ONE algorithm. The algorithm pulls a variable to the root just like the previous version (Algorithm 2.6). However, this version creates fewer intermediate vertices and thus uses less memory.

The UP_ALL algorithm works by sifting all variables to the root at the same time. Like for repeated use of UP_ONE, this eliminates operator vertices and the result is a BDD. UP_ALL is related to Bryant's APPLY-operator [Bry86, Bry92] on BDDs. Converting a BED to a BDD using UP_ALL corresponds to calling APPLY for each operator vertex in the BED in a bottom up fashion. Algorithms 2.8 and 2.9 show the pseudo-code for APPLY and UP_ALL. UP_ALL constructs BDDs in a bottom up way.

Observation 2.3.3 (Up_All). Let u be a vertex in a BED and let $v =$ UP_ALL(u). Then UP_ALL has the following properties:

(*i*) $f^v = f^u$.

(*ii*) v is a BDD.

(*iii*) If l and h are BDDs, then APPLY(op, l, h) is equivalent to using UP_ALL on MK(op, l, h), i.e., the two algorithms return identical BDDs and they make the same number of recursive calls.

(*iv*) If l and h are BDDs, the running time of UP_ALL(MK(op, l, h)) is $O(|l||h|)$.

UP_ONE pulls one variables up to the root. UP_ALL pulls all variables up. Both algorithms can be seen as special cases of a more general UP algorithm which pulls a set of variables up. UP has not been mentioned by

Name: APPLY(op, l, h)
Require: The memorization table M is initialized to empty prior to the first call.

```
1: if (l, h) is in M then return M(l, h)
2: if l and h are terminal vertices then
3:    r ← op(val(l), val(h))
4: else if var(l) = var(h) then
5:    tmp1 ← APPLY(op, low(l), low(h))
6:    tmp2 ← APPLY(op, high(l), high(h))
7:    r ← MK(var(l), tmp1, tmp2)
8: else if var(l) < var(h) then
9:    tmp1 ← APPLY(op, low(l), h)
10:   tmp2 ← APPLY(op, high(l), h)
11:   r ← MK(var(l), tmp1, tmp2)
12: else
13:   tmp1 ← APPLY(op, l, low(h))
14:   tmp2 ← APPLY(op, l, high(h))
15:   r ← MK(var(h), tmp1, tmp2)
16: insert ((l, h), r) in M
17: return r
```

Algorithm 2.8: The APPLY algorithm. It assumes l and h are BDDs. The imposed total order on the variable vertices is denoted $<$. In the code it is assumed that terminal vertices are included at the end of this order when comparing $var(l)$ and $var(h)$.

Name: UP_ALL(u)
Require: The memorization table M is initialized to empty prior to the first call.

```
1: if u is in M then return M(u)
2: if u is a terminal vertex then return u
3: (l, h) ← (UP_ALL(low(u)), UP_ALL(high(u)))
4: if l and h are terminal vertices then
5:    r ← MK(α(u), l, h)
6: else if u is a variable x vertex then
7:    r ← MK(x, l, h)
8: else if var(l) = var(h) then
9:    tmp1 ← UP_ALL(MK(α(u), low(l), low(h)))
10:   tmp2 ← UP_ALL(MK(α(u), high(l), high(h)))
11:   r ← MK(var(l), tmp1, tmp2)
12: else if var(l) < var(h) then
13:   tmp1 ← UP_ALL(MK(α(u), low(l), h))
14:   tmp2 ← UP_ALL(MK(α(u), high(l), h))
15:   r ← MK(var(l), tmp1, tmp2)
16: else
17:   tmp1 ← UP_ALL(MK(α(u), l, low(h)))
18:   tmp2 ← UP_ALL(MK(α(u), l, high(h)))
19:   r ← MK(var(h), tmp1, tmp2)
20: insert (u, r) in M
21: return r
```

Algorithm 2.9: The UP_ALL algorithm. The total order $<$ is defined as for APPLY (see Algorithm 2.8).

Andersen and Hulgaard. Algorithm 2.10 shows the pseudo-code for UP. In this dissertation we do not use UP, but rather stick to UP_ONE and UP_ALL.

2.4 Related Work

BDDs were introduced by Akers in 1978 [Ake78], but it was not until 1986 they became widely used. This is due to Bryant [Bry86] who made BDDs canonical by imposing an ordering of the variables and presenting efficient algorithms for BDD manipulations. Since then, lots of variations have emerged. In his paper [Bry95], Bryant gives an overview of some of the decision diagrams used in formal verification. Becker and Drechsler give in [BD97] an overview for decision diagrams in synthesis.

The family of decision diagrams is large. Each decision diagram has its own features and advantages. Here we discuss some of the decision diagrams. First we discuss the different features. Then we give an overview of the decision diagrams.

The syntax of all decision diagrams are directed acyclic graphs. Each vertex represents a function f over a set of n variables. The domain is D^n and the range is R: $f : D^n \mapsto R$. Many decision diagrams, for example BDDs, have $D = R = \mathbb{B}$.

The graph vertices are labeled. All decision diagrams have vertices labeled with variables. For decision diagrams with $D = \mathbb{B}$, a variable vertex typically has two extra attributes: a low and a high child. There are three main types of semantics (or decompositions) for variable vertices: Shannon (S), positive Davio (pD), and negative Davio (nD). Assume the range R and domain D are both $\mathbb{B}$. Consider a variable vertex v with variable x:

$$\begin{array}{lcll} f & = & (\neg x \wedge f^{low(v)}) \vee (x \wedge f^{high(v)}) & \textit{Shannon } (S) \\ f & = & f^{low(v)} \oplus (x \wedge f^{high(v)}) & \textit{positive Davio } (pD) \\ f & = & f^{low(v)} \oplus (\neg x \wedge f^{high(v)}) & \textit{negative Davio } (nD) \end{array}$$

The negative Davio decomposition is also referred to as the Reed-Muller decomposition. The decomposition is fixed for each variable in a decision diagram. However, it is possible to use two or more different decompositions within one decision diagram. The three composition types can be generalized to functions with non-Boolean ranges, see for example [DBR97a]. The moment decomposition used in BMDs and $*$BMDs is a generalized negative Davio decomposition [BC95].

Depending on the decision diagram, certain variable vertices are removed to make the representation reduced in size. The main reductions are:

Name: $\textsc{Up}(s, u)$
Require: The memorization table M is initialized to empty prior to the first call.

```
1: if (s, u) is in M then return M(s, u)
2: if u is a terminal vertex then return u
3: (l, h) ← (Up(s, low(u)), Up(s, high(u)))
4: if l and h are both terminal vertices then
5:    r ← Mk(α(u), l, h)
6: else if vl and (¬vu or var(l) < var(u)) then
7:    if vh and var(l) = var(h) then
8:       tmp1 ← Up(s, Mk(α(u), low(l), low(h)))
9:       tmp2 ← Up(s, Mk(α(u), high(l), high(h)))
10:      r ← Mk(var(l), tmp1, tmp2)
11:   else if vh and var(h) < var(l) then
12:      tmp1 ← Up(s, Mk(α(u), l, low(h)))
13:      tmp2 ← Up(s, Mk(α(u), l, high(h)))
14:      r ← Mk(var(h), tmp1, tmp2)
15:   else
16:      tmp1 ← Up(s, Mk(α(u), low(l), h))
17:      tmp2 ← Up(s, Mk(α(u), high(l), h))
18:      r ← Mk(var(l), tmp1, tmp2)
19: else if vh and (¬vu or var(h) < var(u)) then
20:   tmp1 ← Up(s, Mk(α(u), l, low(h)))
21:   tmp2 ← Up(s, Mk(α(u), l, high(h)))
22:   r ← Mk(var(h), tmp1, tmp2)
23: else
24:   r ← Mk(α(u), l, h)
25: insert ((s, u), r) in M
26: return r
```

Algorithm 2.10: The Up algorithm. It pulls a set of variables s to the root of a BED u. The total order $<$ is defined as for Apply (see Algorithm 2.8). For readability we use the abbreviation vl for "l is variable vertex and $var(l) \in s$". The terms vh and vu are defined in a similar way with h and u, respectively, instead of l.

Identical (I) : If two variable vertices have identical attributes (variable and low and high children), remove one of them and redirect all incoming edges to the other one.

Same (S) : If a variable vertex has identical low and high children it can be removed. Incoming edges are redirected to the common child.

Zero (Z) : If a vertex has high child **0**, it can be removed. Incoming edges are redirected to the low child.

DDDs [MLAH99, ML98] and LDDs [Gra00] both extend the concept of variable vertices. DDD vertices contain a difference between two real variables. The difference is compared to a constant. LDDs have a linear combination of a number of real variables in the vertices. The linear combination is also compared to a constant. For both diagrams, if a comparison is true (false), the function represented by the vertex is equal to the function of the high (low) child.

Some decision diagrams have vertices labeled with operators[4]. Such vertices typically have two children: low and high. The semantic of the vertex is the function obtained by applying the operator to the functions represented by the two children.

Edges in decision diagrams are always directed. Sometimes edges carry attributes. A typical attribute is a negation mark. A negation mark means that the function of the vertex pointed to by the edge should be negated. Most decision diagrams can have negations on the edges, and we do not mention it explicitly. Other possible edge attributes are weights ($*$BMDs [BC95] and EVBDD [VPL96, LPV94]) and existential and universal quantifiers (XBDDs [JPHS91]).

Many decision diagrams restrict how variables may occur. Typical restrictions are:

Order : The variables must obey a global ordering along all paths.

Free : The variable must obey an ordering, but the ordering may be different along different paths.

Index : The paths are segmented into "indexes". The variables must obey a global ordering within each segment.

[4]Strictly speaking, such data structures are not decision diagrams. However, they share many common traits with decision diagrams and therefore we choose to use the term decision diagrams for these structures as well.

A typical operator on decision diagrams is to compare two of them. It is straightforward to determine whether they are syntactically identical. However, we often want to know if they are semantically identical, i.e., we want to know if they represent the same functions. We call a decision diagram canonical if it has the property that syntactical equivalence implies semantical equivalence.

Tables 2.2 and 2.3 give an overview of the different decision diagrams. It is not a complete list. In the literature, BDDs are often called OBDDs to stress that they are ordered. We have opted to call them BDDs. However, there exists a non-ordered BDD data structure. We shall call it GBDD for general BDD. In the literature, GBDDs are sometimes refered to as BDDs.

Name	Functions	Decomp.	Reduct.	Operators	Restrict.	Canonical
BDD	$\mathbb{B}^n \mapsto \mathbb{B}$	S	I, S	no	order	yes
FBDD	$\mathbb{B}^n \mapsto \mathbb{B}$	S	I, S	no	free	no
GBDD	$\mathbb{B}^n \mapsto \mathbb{B}$	S	I, S	no	none	no
IBDD	$\mathbb{B}^n \mapsto \mathbb{B}$	S	I, S	no	index	no
FDD	$\mathbb{B}^n \mapsto \mathbb{B}$	nD	I, Z	no	order	yes
OKFDD	$\mathbb{B}^n \mapsto \mathbb{B}$	S, pD, nD	I, S, Z	no	order	yes
XBDD	$\mathbb{B}^n \mapsto \mathbb{B}$	S	I, S	on edges	order	semi
ZBDD	$\mathbb{B}^n \mapsto \mathbb{B}$	S	I, Z	no	order	yes
MTBDD	$\mathbb{B}^n \mapsto \mathbb{Z}$	S	I, S	no	order	yes
BMD	$\mathbb{B}^n \mapsto \mathbb{Z}$	nD	I, S	no	order	yes
$*$BMD	$\mathbb{B}^n \mapsto \mathbb{Z}$	nD	I, S	no	order	yes
K$*$BMD	$\mathbb{B}^n \mapsto \mathbb{Z}$	S, pD, nD	I, S, Z	no	order	yes
EVBDD	$\mathbb{B}^n \mapsto \mathbb{Z}$	S	I, S	no	order	yes
HDD	$\mathbb{B}^n \mapsto \mathbb{Z}$	multiple	I, S	no	order	yes
FBD	$\mathbb{B}^n \mapsto \mathbb{B}$	S	I, S	$\wedge, \oplus$	free	no
BED	$\mathbb{B}^n \mapsto \mathbb{B}$	S	I, S	yes	none	no
DDD	$\mathbb{R}^n \mapsto \mathbb{B}$	S	I, S	no	order	semi
LDD	$\mathbb{R}^n \mapsto \mathbb{B}$	S	I, S	no	order	semi

Table 2.2: Overview of decision diagrams. MTBDDs are also called ADDs for Algebraic Decision Diagrams. $*$BMDs and EVBDDs both have weights on the edges.

In 1996 and 1997, Hett, Drechsler, and Becker presented a new method for BDD construction [HDB96, HDB97]. They called it MORE for *Multi-operand synthesis OR-operations based on Existential quantification.* Their idea is to introduce extra variables, so called coding variables, in the BDD. The coding variables are implicitly existentially quantified. Vertices containing a coding variable are in effect OR-vertices since $(\exists s : s \to f, g) \leftrightarrow (f \vee g)$ assuming that f and g do not depend on s. MORE constructs the BDD

Name	Full Name	References
BDD	(Ordered) Binary Decision Diagrams	[Bry86, Bry92]
FBDD	Free BDDs	[GM94, SW95]
GBDD	General BDDs	[AGD91]
IBDD	Indexed BDDs	[JBA+97]
FDD	Functional Decision Diagrams	[KSR92]
OKFDD	Ordered Kronecker FDD	[DST+94]
XBDD	Extended BDDs	[JPHS91]
ZBDD	Zero-suppressed BDDs	[Min93, Min96]
MTBDD	Multi-terminal BDDs	[CMZ+93, BFG+93]
BMD	Binary Moment Diagrams	[BC95]
*BMD	Multiplicative BMDs	[BC95]
K*BMD	Kronecker Multiplicative BMDs	[DBR95, DBR97a, DBR97b]
EVBDD	Edge-valued BDDs	[VPL96, LPV94]
HDD	Hybrid Decision Diagrams	[CFZ95]
FBD	Free Boolean Diagrams	[SDG95]
BED	Boolean Expression Diagrams	[AH97, AH]
DDD	Difference Decision Diagrams	[MLAH99, ML98]
LDD	Linear Decision Diagrams	[Gra00]

Table 2.3: References to decision diagrams.

by moved coding variables toward the terminals using the level exchange operation [FMK91] which is similar to Equation 2.2, but for variables and not operators. Furthermore they use negation marks on the edges. MORE can be extended to all binary Boolean connectives since disjunction and negation are functionally complete. MORE can be seen as a first step in the direction of BEDs.

Chapter 3

Equivalence Checking of Flat Combinational Circuits

In this chapter we discuss how to use the BED data structure in equivalence checking of flat combinational circuits. First we define the problem and show how to convert it to the tautology problem for BEDs. Then we discuss different methods of solving the tautology problem for BEDs including heuristics and tuning of the algorithms and data structure. Finally, we survey related work. Part of this chapter is based on the papers [HWA97, HWA99].

3.1 Introduction

Designing complex combinational circuits is a multi-step process. Typically the designer starts with a high-level description of the circuit working his way toward a low level description. Each step is a refinement or a modification of the previous one. The refinements and modifications are done either manually by the designer or by a computer program. To ensure that no errors are introduced, each version of the circuit is compared with the previous version. This is called the equivalence checking problem because it is the problem of proving that the two versions of the circuit are functionally equivalent.

The functionality of a combinational circuit can be described in propositional logic. For a gate-level circuits, the following correspondences apply:

circuit		logic
inputs	$\sim$	variables
gates	$\sim$	connectives
outputs	$\sim$	formulae

We only consider combinational circuits without cycles. A cycle potentially introduces a memory element into the circuit. The outputs of the circuit are thus no longer only functions of the inputs. Now they are also functions of the previous inputs. Consider, for example, the circuit in Figure 3.1. The circuit has a cycle which acts like a memory element. Assuming that everything is 0 to begin with, the input sequence $\langle 00, 01, 10, 11\rangle$ produces the output sequence $\langle 1, 1, 0, 0\rangle$ while the input sequence $\langle 00, 10, 01, 11\rangle$ produces the output sequence $\langle 1, 0, 1, 1\rangle$. In the two cases, the outputs for input 11 differ. By disallowing cycles, we avoid such behavior.

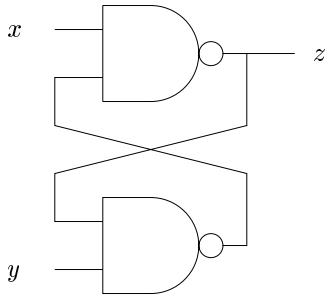

Figure 3.1: A circuit with two NAND gates. There is a hidden memory element in the circuit. The output value z for input $xy = 11$ depends on the previous inputs.

A circuit is described by the formulae for its outputs. For example, the output formulae for the two combinational circuits in Figure 1.1 are:

$$\begin{aligned}
o_1' &= i_1 \mathbin{\bar{\wedge}} \big((i_2 \mathbin{\bar{\wedge}} i_3) \mathbin{\bar{\wedge}} (i_3 \mathbin{\bar{\wedge}} i_4)\big) \\
o_2' &= i_3 \mathbin{\bar{\wedge}} i_4 \\
o_1'' &= i_1 \mathbin{\bar{\wedge}} \big((i_2 \wedge i_3) \vee (i_3 \wedge i_4)\big) \\
o_2'' &= \big((i_2 \wedge i_3) \vee (i_3 \wedge i_4)\big) \mathbin{\bar{\wedge}} i_4
\end{aligned}$$

Combinational circuits described on other levels of abstraction can also be transformed to formulae in propositional logic. As we have just seen, the transformation from gate-level circuit descriptions to Boolean formulae is straightforward. However, transformations from other description levels, e.g., the transistor level, to Boolean formulae may be a difficult problem.

It is worth noting that we actually transform *descriptions* of hardware circuits into propositional logic. This has some implications:

- The propositional logic formulae only capture the functional behavior of the circuit mentioned in the description. For example, glitches and hazards are often not modeled.

- Errors introduced at a later stage (for example, defect components in the final hardware) are not taking into account.

For simplicity we say that we are comparing two circuits when in fact we are comparing the propositional logic formulae obtained from the descriptions of two circuits. Since we always deal with descriptions of circuits, we can never skip a test of the final hardware circuits. Formal verification proves that the design works, but testing examines the physical implementation of the design. Formal verification and testing complement each other.

We consider two circuits equivalent if they have pairwise identical outputs given pairwise identical inputs. In an industrial setting, this might not be the case. For example, when optimizing for power consumption, it may be advantageous to duplicate an input. Or when mapping a circuit to a component library, only parts of some components are used. In both cases, there are more inputs and/or outputs in the resulting circuit than in the original one. Equivalence checking of two such circuits can be done by first finding a possible mapping between the inputs and between the outputs, and then proving the equivalence between the two new circuits under this mapping. Typically this is done iteratively. However, for the purpose of this dissertation, we shall only consider proving that the circuits have pairwise identical outputs given pairwise identical inputs.

Consider a pair of outputs; one from each of the two circuits. Let f and g be the logic formulae representing these outputs, where f and g are formulae over the same variables. The two outputs are equivalent if, for all variable assignments, f and g evaluate to the same value. In other words, we need to solve the tautology problem for $f \leftrightarrow g$.

The functionality of the two circuits may not be 100% specified. Or the two circuits may be at different levels of abstraction. To take care of such situations, it is convenient to specify a set of variable assignments called the care-set. The care-set is the set of variable assignments for which f and g should evaluate alike. Outside the care-set, f and g may take different values. Let c be the characteristic function for the care-set. Then this problem can be cast as the tautology problem for $c \rightarrow (f \leftrightarrow g)$.

We deal with the case where c is 1, that is, we consider f and g to be equal only if they are equal for all variable assignments.

The equivalence checking problem also occurs as a subproblem of other verification problems. For example, when verifying arithmetic circuits by checking that they satisfy a given recurrence equation [Fuj96] or when verifying the equivalence of two state machines without performing a state traversal [vE98].

The equivalence checking problem can be solved using BDDs. However, consider the case of comparing two identical circuits. This corresponds to constructing the BDD for the function $f \leftrightarrow f$. To do this we would first construct the BDDs for the two f-functions. Then we would realize that they are identical. If f has a large BDD representation, this method may not be feasible. Using BEDs instead of BDDs, we can construct the BED for $f \leftrightarrow f$. The BED for f is of size linear in the circuit description, i.e., we can construct the BED for f without using much memory. It is now trivial to realize that $f \leftrightarrow f$ is a tautology and convert $f \leftrightarrow f$ to a BDD without converting f to a BDD first. The following section describes ways of simplifying BEDs based on similar observations.

3.2 Simplifications

We measure the size of a BED as the number of vertices it has. Almost all BED algorithms have runtimes depending on the size of the involved BEDs. By keeping the BED size down, we get faster algorithms and less memory usage.

We want each BED to have a small number of vertices. We can obtain this if the vertices within a BED are used often, i.e., they have multiple edges leading to them, or by rewriting the BED to a smaller, but equivalent BED. At the same time we want the combined size (total number of different vertices) of all the BEDs to be small. In the following sections we describe different ways of simplifying BEDs.

3.2.1 Operator Sets

Recall from Section 1.2 that not all 16 binary Boolean connectives are needed in propositional logic. Only a small number of them are necessary to express all the others. A set of connectives able to express all other connectives is called functionally complete. For example, the sets $\{\wedge, \vee, \neg\}$ and $\{\bar{\wedge}\}$ are both functionally complete.

The same idea can be applied to operator vertices in BEDs. Instead of allowing all binary Boolean connectives as *op* attributes, we can restrict ourselves to any functionally complete set. The remaining operators can be obtained using multiple ones from the set. Should we use a small or a large set of operators?

- With fewer different connectives, the chance of reuse of vertices is greater thus reducing the BED size.
- With only a few different connectives available, some connectives require more than one operator vertex thus increasing the BED size.

It is not clear what set to choose. The following argumentation leads to a set which works well in practice.

Of the 16 binary Boolean connectives, five can easily be eliminated: $K0$, $K1$, π_1, π_2, and $\bar{\pi}_2$. They can be replaced by terminals, left or right argument, or negation. The remaining 11 connectives fall in four categories:

Negation	:	$\{\neg\}$
Positive	:	$\{\vee, \rightarrow, \leftarrow, \bar{\wedge}\}$
Negative	:	$\{\bar{\vee}, \not\rightarrow, \not\leftarrow, \wedge\}$
Neutral	:	$\{\leftrightarrow, \oplus\}$

The positive connectives have three out of four 1s in their truth tables. The negative connectives have three out of four 0s. The neutral connectives have two 1s and two 0s.

Each positive connective has a corresponding negative connective such that the truth table for one connective is the negated of the truth table for the other one. The same is true for the two neutral connectives. Let us consider the set of negation, the positive connectives, and one of the neutral ones:

$$\{\neg, \vee, \rightarrow, \leftarrow, \bar{\wedge}, \leftrightarrow\}\,. \tag{3.1}$$

Each of the remaining connectives can be expressed using at most one extra negation. For example, $a \wedge b$ can be expressed as $\neg(a\ \bar{\wedge}\ b)$. This set is small enough to allow sharing and large enough not to increase the BED size by adding too many extra operator vertices. The set also has other nice properties which we discuss in the next section.

3.2.2 Rewriting

Keeping the BEDs reduced, as mentioned in Definition 2.1.3, already gives us size reductions due to, for example, constant propagation. But we can

reduce the size of the BEDs even more. This can be achieved by increasing the sharing of vertices and by removing local redundancies.

The BEDs for $f \vee g$ and $g \vee f$ are syntactically different, but semantically equivalent. We can increase sharing by always choosing one over the other. In general, we fix an ordering $<$ of the vertices and only create operator vertices with $low < high$. The set (3.1) is closed under symmetry. This means that if op_1 and op_2 are operators such that $f\ op_1\ g$ is semantically equivalent to $g\ op_2\ f$, then either both operators or none of them are in the set (3.1).

The price we pay for using the set (3.1) is at most one extra negation vertex per connective outside the set. But we can eliminate many of these extra negation vertices. All negations below binary operators can be removed since for each binary operator op there exists another operator op' such that $f\ op'\ g$ is equivalent to $\neg f\ op\ g$. In this way we only need negation at the root of a BED or just below variable vertices. Since negation vertices are only needed there, we can skip them and replace them with one of the remaining binary Boolean operators. For example, $\neg(f \leftrightarrow g)$ would be written $f \oplus g$ even though the $\oplus$ operator is not part of the set (3.1).

We can exploit equivalences like the absorption laws, for example $f \vee (f \wedge g) = f$, and distributive laws, for example $(f \wedge g) \vee (f \wedge h) = f \wedge (g \vee h)$, to reduce the size of the BEDs. In both cases we eliminate one or more connectives.

Figure 3.2 shows the rewriting rules for the BEDs. The rules are presented as a filter and a transformation. Every time we create a new operator vertex, we match it against the filters. In case of a match, the corresponding transformation gives an equivalent, but locally smaller, BED. Since there are only $16^3 = 4096$ combinations of three operators (and even less if we do not allow all 16 different operators) it is feasible to tabulate all possible rewriting rules.

To incorporate rewriting rules in BEDs, we just have to alter the MK algorithm slightly. Algorithm 3.3 shows the pseudo-code. The only difference between this version of MK and the one in Algorithm 2.3 are lines 5 and 6 where we apply the rewriting rules by table lookup. The functionality of the lines 5 and 6 in the old MK algorithm is covered by the rewriting rules.

With every set of rewriting rules, one needs to worry about termination. We apply our rules recursively. Rule (1) normalizes an operator vertex. Since the set of connectives we use is closed under symmetry, this does not introduce extra vertices. Rule (3) and (4) may replace an operator with a negation. In all other cases the rewriting rules locally reduce the number of connectives (and thus the number of operator vertices) in the BED. Thus,

Filter	$\Rightarrow$	Transformation
(1) normalization $f\ op_1\ g$	$\Rightarrow$	$g\ op_2\ f$ where $g < f$ in the vertex order
(2) negation absorption $\neg f\ op_1\ g$	$\Rightarrow$	$f\ op_2\ g$ where op_2 is found using identities
(3) constant propagation $f\ op_1\ T$	$\Rightarrow$	$\mathbf{0}, \mathbf{1}, f, \neg f$ depending on op_1 and terminal T
(4) repeated children $f\ op_1\ f$	$\Rightarrow$	$\mathbf{0}, \mathbf{1}, f, \neg f$ depending on op_1
(5) absorption $f\ op_1\ (\ f\ op_2\ g\ )$	$\Rightarrow$	$f\ op_3\ g$
(6) distributivity 1 $(\ f\ op_2\ g\ )\ op_1\ (\ f\ op_3\ g\ )$	$\Rightarrow$	$f\ op_4\ g$
(7) distributivity 2 $(\ f\ op_2\ g\ )\ op_1\ (\ h\ op_3\ f\ )$	$\Rightarrow$	$(\ f\ op_5\ g\ )\ op_4\ h$ if possible
(8) distributivity 3 $(\ f\ op_2\ g\ )\ op_1\ (\ h\ op_3\ f\ )$	$\Rightarrow$	$f\ op_4\ (\ g\ op_5\ h\ )$ if possible

Figure 3.2: Rewrite rules for BEDs. Most of the rules have one or more symmetric cases which are not shown. For example, in rule (3), f and T may be swapped, and in (7) and (8) the three arguments f, g and h may be swapped. T is a terminal. The "if possible" in rule (7) and (8) indicates that the rewriting is not possible for all combinations of op_1, op_2 and op_3.

Name: MK(α, l, h)
1: **if** there exists a (α, l, h) vertex **then**
2: **return** that vertex
3: **else if** α is a variable and $l = h$ **then**
4: **return** l
5: **else if** α is operator and (α, l, h) is in rewriting table **then**
6: **return** MK$(lookup(\alpha, l, h))$
7: **else**
8: **return** new vertex (α, l, h)

Algorithm 3.3: The modified MK algorithm which includes rewriting rules. The algorithm works just like the old MK algorithm; see Algorithm 2.3. The only difference is that before creating new operator vertices, MK now looks in a rewriting table to find a smaller representation.

rewriting terminates.

However, the rewriting rules only reduce the BED size locally. The vertices discarded by the rewriting rules may be in use elsewhere in this or other BEDs. Thus the total number of vertices may not go down. In fact, the total number of vertices may increase. The increase arises from lost sharing. Figure 3.4 gives an example where the use of rewrite rule (8) increases the total number of vertices. Let γu be an expanded graph for BED u such that γu is identical to u except that shared vertices are duplicated so no vertex has in-degree larger than one. Think of γ as an operator which turns a DAG into a tree. Consider a BED before (u) and after (u') the application of a rewriting rule. Then the size of $\gamma u'$ is less than γu, where size is measured in number of vertices. The only exceptions are rule (1) and part of rules (3) and (4) where the size is unchanged.

We state, without a formal proof, that our rewriting system makes the BED simpler by either decreasing the BED sizes (in the γ sense), reducing a binary operator to negation, or normalizing an operator.

One could go a step further and extend the rewriting rules to even greater depth. This poses the question of which rewriting rules to include? With a greater depth it is no longer feasible to tabulate all possibilities. There is also no need to include rules which are combinations of a number of simpler rules. Another question is how do we match a BED against the rules? Hoffmann and O'Donnell [HO82] show different methods for pattern matching in trees. Similar techniques must be developed for the BED structure.

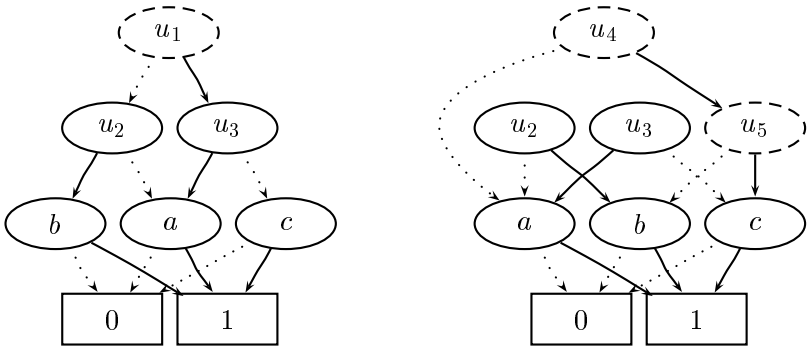

Figure 3.4: Lost sharing of vertices. The u vertices are operator vertices. Assume vertices u_2 and u_3 are shared with another BED. If we create vertex u_1, the BED matches rewrite rule (8). The resulting BED contains two new operator vertices u_4 and u_5 instead of u_1, u_2 and u_3. But since u_2 and u_3 are shared, they cannot be removed.

3.2.3 Other Simplification Methods

In the previous sections we have described two simplification methods (operator set restriction and rewriting). But there are other methods available. Here we just mention a few of them.

The first method is balancing. Consider the deep and thin BED for $a \vee (b \vee (c \vee a))$. The rewriting rules do not work well for such BEDs. Had we placed the parentheses differently, the rewriting rules would be more effective: $(a \vee b) \vee (c \vee a)$ transforms to $(a \vee b) \vee c$ and thus saves a vertex. Figure 3.5 shows the three BEDs. Balancing can be applied to BEDs with other operators and to BEDs with mixed operators. In one example (`barrel6.dimacs` converted to BED format) from the BMC distribution [BCCZ99, BCC$^+$99, BCRZ99], balancing reduced the longest path from 8933 to 21.

A second simplification method, based on a greedy strategy, is Up_One-minimizing. Each variable is in turn pulled to the root of the BED using Up_One. If it shrinks the BED in size, then the new BED is kept, otherwise it is discarded and we continue to use the old BED. The process continues until no variable shrinks the BED further. The result depends on the order in which the variables are tried. Up_One-minimizing can be generalized to n variables instead of just one variable. In the general version, n variables are Up_One'ed, and the result is kept if it is smaller than the original BED. We continue until Up_One of no combination of n variables shrinks the

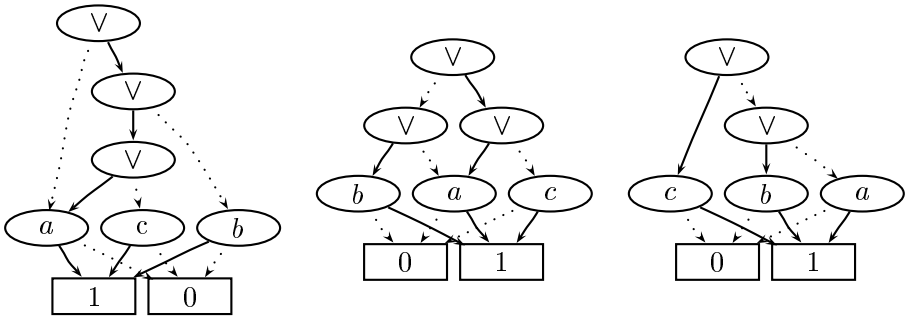

Figure 3.5: Balancing BEDs. All three BEDs represent the same Boolean function. The rewriting rules are not able to do anything with the leftmost BED as it is too deep. In the middle BED, the operators are placed as to minimize the depth. The rewriting rules now recognize the shared a vertex. The rightmost BED shows the result of applying the rewriting rules.

BED further. However, for complexity reasons n equal to 1 or 2 is the limit in practice.

A third method is the use of Stålmarck's method. We discuss Stålmarck's method in detail later, but here it suffices to say that the algorithm works on Boolean formulae by extracting knowledge from them under some assumption and then extracting knowledge under the negated assumption. The intersection of the two knowledge sets can be inferred without any assumptions. In BED terms, the knowledge we extract is in the form of equivalence between vertices, and we can use it to replace large sub-BEDs with small but equivalent sub-BEDs. The power of Stålmarck's method is divided into saturation levels, where a higher level indicates that more knowledge can be extracted but at the cost of a higher complexity. Bjesse [Bje99] proposed this minimization idea for formulae. However, Stålmarck's method is not able to simplify BEDs by much. The low saturation levels mainly identify identical sub-formulae – something which BEDs handle with sharing. The higher saturation levels have too high a complexity to be of interest.

Pruning is a fourth simplification method. It exploits the uneven distribution of 0s and 1s in the positive and the negative connectives. For example, consider the formula $f \vee g$. If g is true, then the whole formula is true independently of f. However, if g is false, then the value of the formula depends on f. Since f only matters when g is false, we can use this fact to simplify f. If g is the formula x, where x is a variable, then $f \vee x$ is equivalent to $f[0/x] \vee x$, where $f[0/x]$ is a substitution of 0 for x in f. In general $f \vee g$ is equivalent to $f[0/g] \vee g$, where $f[0/g]$ is a simplification of f given that g is

0. Coudert, Berthet and Madre [CBM89] presented a method for doing such simplifications. Their method simplifies a formula f to a smaller formula f' such that f and f' are equivalent for variable assignments in a care-set. The method can easily be applied to BEDs. However, the nature of the method is such that the BDDs for the formulae are implicitly constructed. Since we try to avoid BDD construction for intermediate results, this method is not interesting for us. The idea of pruning is still very interesting. In Chapter 5 we use a special version of pruning as a key ingredient in model checking.

3.2.4 Experimental Results for Simplifications

In this section we give some experimental results showing the effects of operator and rewriting simplifications. As test examples we use a set of combinational circuits from the ISCAS'85 benchmark suite [BF85]. The combinational circuits are present in two versions: one with and one without redundancies removed. We verify that each pair of circuits has equivalent outputs given identical inputs.

For these experiments we use a Pentium III 450 MHz computer running Linux. We report the runtime in seconds and the number of vertices used. We use Up_One and Up_All to convert the BEDs to BDDs. The input variables are ordered as they appear in the original netlists of the circuits[1].

First, we run the BED package with no simplifications (except to ensure that all BEDs are reduced as per Definition 2.1.3), i.e., we use the full set of operators and no rewriting rules. The results are in Table 3.1. Up_One is not able to handle any of the circuits within 3 million vertices of memory (almost 64 MB of memory).

Second, we run the same examples again, but this time we let the BED package use all rewriting rules and the restricted set of operators. The results are in Table 3.2. The Up_One algorithm now completes the verification. The Up_All algorithm is both faster and uses significantly less memory in four out of five cases. In the last case rewriting rules and the operator set does slightly worse. Note the case of `c499` versus `c1355`. The simplifications alone are enough to verify the two circuits without the use of Up_One or Up_All.

Based on these experiments we conclude that the simplifications are critical for the performance of Up_One. For Up_All, the simplifications generally work well and give an improvement in speed and memory.

[1] While this is not an optimal ordering, it is enough in these experiments. Later in this chapter we address the variable ordering question.

	Up_One		Up_All	
Circuit	Runtime	Size	Runtime	Size
`c432`, `c432nr`	-	-	0.1	12,722
`c499`, `c499nr`	-	-	2.0	80,913
`c499`, `c1355`	-	-	3.4	191,882
`c1355`, `c1355nr`	-	-	4.4	205,600
`c1908`, `c1908nr`	-	-	1.7	209,660

Table 3.1: Results for some of the ISCAS'85 circuits without simplifications. Runtimes in seconds and sizes in number of vertices for verifying circuits. No rewriting rules are used. The full set of operators are used. A dash indicates that the computation required more than 3 million vertices.

	Up_One		Up_All	
Circuit	Runtime	Size	Runtime	Size
`c432`, `c432nr`	0.4	88,421	0.1	11,808
`c499`, `c499nr`	2.3	580,136	0.3	14,596
`c499`, `c1355`	0.1	701	0.1	701
`c1355`, `c1355nr`	2.7	580,544	0.3	15,004
`c1908`, `c1908nr`	1,826	1,255,665	1.9	256,541

Table 3.2: Results for some of the ISCAS'85 circuits with simplifications. Runtimes in seconds and sizes in number of vertices for verifying circuits. All rewriting rules are used and the operators are restricted to the set (3.1).

To explain why the simplifications have different effects on Up_One and Up_All, we need to look more closely at the algorithms. During the initial BED construction the rewriting rules are applied. This reduces the initial size of the BED before we apply either algorithm. However, only Up_One constructs new operator vertices. Up_All does not. This means that Up_All is unable to take advantage of the rewriting rules. Up_One constructs both variable and operator vertices and we are thus able to apply the rewriting rules during the BED to BDD conversion. The improvement we see in Table 3.2 over Table 3.1 for Up_All is because of simplification of the initial BED. This leads to an interesting use for BEDs: A preprocessing tool for Boolean formulae used before applying standard techniques like BDDs.

The ISCAS'85 suite also contains two 16-bit multipliers, `c6288` and `c6288nr`. Figure 3.6 shows the BED for verifying that the 17th output of the two multipliers are equivalent. We have not applied any rewriting rules. The red vertices are from `c6288` and the blue ones are from `c6288nr`. The purple vertices are shared between the two multipliers. The single black vertex on top is the biimplication between the two output functions. The figure shows that there is not much sharing between the two multipliers as most vertices are either red or blue. Figure 3.7 shows the same picture, but this time with the rewriting rules turned on. The result is a mere 11 vertices compared to 2448 vertices before. The rewriting rules have eliminated most of the vertices.

In the rest of this dissertation, all experimental results are performed with the rewriting rules in Figure 3.2 and BEDs restricted to the set of operators in (3.1).

3.3 Variable Ordering

The efficiency of Up_One and Up_All depends on the variable order. Although the initial and final size of the BEDs are independent of the variable order, the intermediate BEDs may grow exponentially. The initial BED for a circuit has only variable vertices with terminal children, and thus it respects all possible orderings. If the verification succeeds, the result is the BED **1**, which is independent of all variables.

A large number of variable ordering heuristics have been developed for BDDs based on the topology of a circuit [BRRM91, CHP93, FOH93, FFK88, JPHS91, MWBSV88, Min96]. The ordering heuristics attempt to statically determine a variable order such that the BDD representation of the circuit

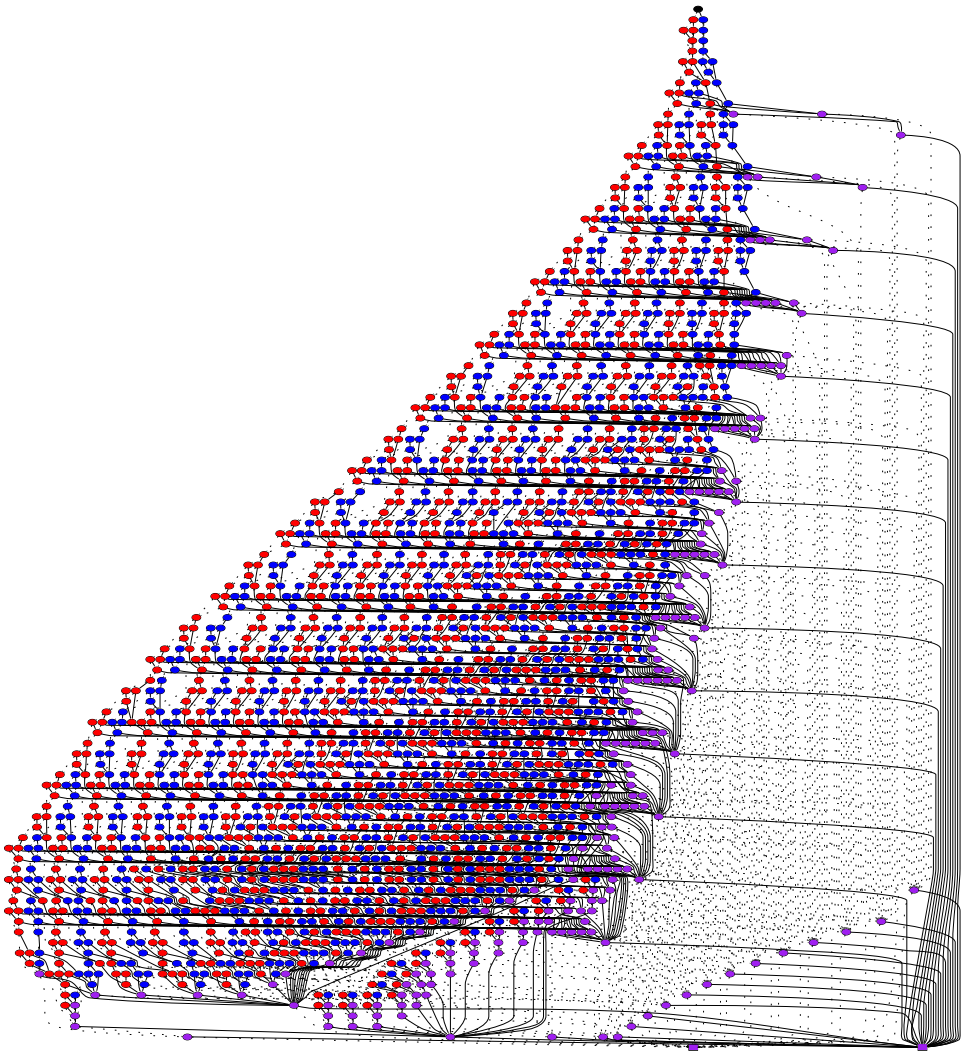

Figure 3.6: Verification of the 17th output bit of two 16-bit multipliers *without* using rewriting rules. The red vertices are from one multiplier, the blue vertices are from the other one. Purple vertices are shared between the two multipliers. The black vertex on top contains a biimplication.

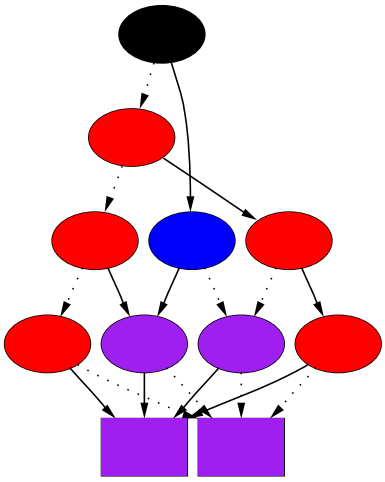

Figure 3.7: Verification of the 17th output bit of two 16-bit multipliers *with* rewriting rules. The red vertices are from one multiplier, the blue vertex is from the other one. Purple vertices are shared between the two multipliers. The black vertex on top contains a biimplication.

is small. Typically, these heuristics consist of three steps to obtain a single global variable order: first, an order of the primary outputs is constructed. Second, for each of the primary outputs in this order, the variables in the support of the output are ordered. Third, the orders for the different outputs are merged into a single global variable ordering. We only consider the step of finding a variable ordering for a *given* primary output, since different variable orders can be used for different roots of a BED. This allows a greater flexibility in finding good variable orders since the orders of the primary outputs are independent. However, the cost is that there is no or little reuse between verifying different primary outputs. In most BDD packages, all BDDs *must* respect the same ordering of the variables.

Since Up_All essentially works as Apply (property (*iii*) of Observation 2.3.3), the variable orders that are good for BDDs are also good orders to use with Up_All. Thus, when using Up_All we can immediately use the variable ordering heuristics developed for BDDs.

Since Up_One works quite differently than Up_All, the variable ordering heuristics developed for BDDs may not be effective when using Up_One. However, our experiments show that this is not so; a good BDD variable order also keeps the intermediate BEDs small when constructing a BDD with Up_One. The reason for this is that a good variable order for BDDs

has dependent variables close in the order. This allows UP_ONE to collapse sub-circuits early in the verification process. Also, a good variable order has the variables that affect the output the most early in the order. UP_ONE then pull these variables to the root first which allow the most reductions.

In the following we present two variable ordering heuristics, originally developed for BDDs, but which have proven to be effective for BEDs. The inputs to the heuristics were meant to be circuit descriptions. However, we use BEDs instead. This has some implications. For example, a circuit may contain a 4-input AND gate. Using BEDs, this translates to three AND operator vertices. A heuristic like DEPTH_FANOUT distributes a weight evenly among the fanout / children; see Figure 3.8. This causes the heuristic to give different results when run on circuits and on BEDs.

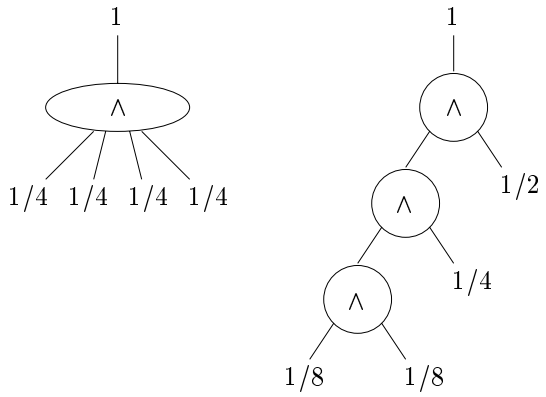

Figure 3.8: A four-input AND gate (left) and a corresponding BED (right). A weight of 1 is distributed from the top down. Each vertex distributes the weight evenly among the children.

3.3.1 The Fanin Heuristic

A number of variable ordering heuristics are based on a depth-first traversal of the circuit [CHP93, FFK88, MWBSV88]. A depth-first traversal is a simple and fast heuristic that has shown to be practical for most combinational circuits [CHP93, JPHS91] since inputs that are close together in the circuit are also placed together in the ordering. The depth-first based heuristics differ in how they decide in what order the inputs of a gate are visited. The FANIN heuristic by Malik *et. al.* [MWBSV88] uses the *depth* of the inputs to a gate to determine in what order to consider the inputs:

Definition 3.3.1. The *depth* of a vertex v is defined recursively as:

$$depth(v) = \begin{cases} 0 & : \quad v \text{ is a terminal vertex} \\ m & : \quad \text{otherwise} \end{cases}$$

where m is

$$\max\Big(depth\big(low(v)\big), depth\big(high(v)\big)\Big) + 1\,.$$

The FANIN heuristic for a BED is shown in Algorithm 3.9. Calling FANIN(u) determines a variable order σ for the variables in $sup(u)$.

In one traversal of the BED u, we can mark all vertices with their depth. Then the runtime of FANIN(u) is linear in the size of the BED u, i.e., $O(|u|)$.

Name: FANIN(u)
Require: The *mark* array must be initialized to **false** before calling FANIN.

```
1:  if mark[u] then
2:    return ⟨⟩
3:  else
4:    mark[u] ← true
5:    if u is variable x then
6:      σ ← ⟨x⟩
7:    else if depth(low(u)) > depth(high(u)) then
8:      σ ← FANIN(low(u))^FANIN(high(u))
9:    else
10:     σ ← FANIN(high(u))^FANIN(low(u))
11:   return σ
```

Algorithm 3.9: The FANIN heuristic for determining a variable ordering σ for BED vertex u. Concatenation of sequences is denoted by the operator $\hat{}$. The symbol $\langle\rangle$ denotes the empty sequence. The algorithm assumes that all variable vertices in the BED u have *low* child **0** and *high* child **1**.

3.3.2 The Depth_Fanout Heuristic

The FANIN heuristic does not capture that variables that affect the output the most should be ordered first, something which is particularly important for UP_ONE. The DEPTH_FANOUT heuristic [Min96], shown for BEDs instead of circuits in Algorithm 3.10, attempts to determine the variables that affect an output the most by propagating a value from the output backward toward the primary inputs. The value is distributed evenly among the input signals to a gate: if a value of c is assigned to the output of a gate with

n input signals, the value assigned to each of the n fanin signals is incremented by c/n (the signal may be input to several gates and thus obtains a contribution from each gate). After propagating the value throughout the circuit to the primary inputs, the DEPTH_FANOUT heuristic adds the primary input with the highest value to the variable order. This input is then removed from the circuit and the process is repeated until all variables in the support have been included in the variable order.

The runtime of DEPTH_FANOUT(u) is $O(|sup(u)| \cdot |u|)$ since the loop (line 3 to 8) is repeated $|sup(u)|$ times and propagation of values can be performed in time linear in the number of reachable nodes. Thus, this heuristic takes longer to compute than FANIN.

Name: DEPTH_FANOUT(u)
1: $\sigma \leftarrow \langle\rangle$
2: $S \leftarrow sup(u)$
3: **while** $S \neq \emptyset$ **do**
4: propagate value 1.0 downwards from u.
5: $x \leftarrow$ the variable with the highest value.
6: $\sigma \leftarrow \sigma^\frown\langle x\rangle$
7: remove x from the BED.
8: $S \leftarrow S \setminus \{x\}$
9: **return** σ

Algorithm 3.10: The DEPTH_FANOUT heuristic for determining a variable ordering σ for BED vertex u.

3.4 Stålmarck's Method

BED to BDD conversion using UP_ONE and UP_ALL is not the only way to solve the satisfiability and tautology problems for BEDs. Stålmarck's method [SS98] is a patented algorithm for tautology checking of formulae in propositional logic. In this section we show how to adapt Stålmarck's method to BEDs.

The advantage of Stålmarck's method is its small memory usage. BED to BDD conversion using either UP_ONE or UP_ALL requires in the worst case memory and time of size exponential in the original BED. However, Stålmarck's method is often slower than UP_ONE and UP_ALL.

Stålmarck's method determines tautology of formulae in propositional logic. We allow a propositional logic formula f to have the following form:

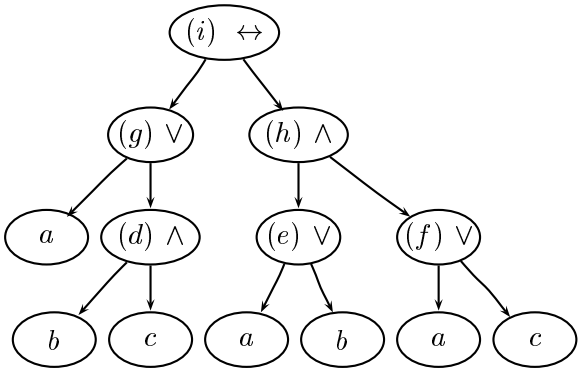

Figure 3.11: Parse-tree for $a \vee (b \wedge c) \leftrightarrow (a \vee b) \wedge (a \vee c)$. Each vertex is labeled with a Boolean variable.

$$f \ ::= \ a \mid \neg f \mid f_1 \odot f_2 \,,$$

where $\odot$ denotes a binary operator from the set:

$$\{\vee, \wedge, \rightarrow, \leftarrow, \leftrightarrow, \bar{\vee}, \bar{\wedge}, \not\rightarrow, \not\leftarrow, \oplus\} \,.$$

The idea behind Stålmarck's method is that to prove that f is a tautology, we negate f, and then show that $\neg f$ is unsatisfiable. More technically we proceed as described below.

From a formula f we construct a parse-tree. Each leaf vertex contains a Boolean variable and we label such vertices with the variable they contain. Each internal vertex contains a Boolean connective. We label each internal vertex with a new unique Boolean variable. From such a parse-tree we construct a set of triplets — one triplet for each internal vertex. A triplet is a constraint on the variables of the internal vertices expressed in terms of the operator of the vertex and the variable labels of the children.

Consider the formula $a \vee (b \wedge c) \leftrightarrow (a \vee b) \wedge (a \vee c)$. Figure 3.11 shows the parse-tree. It gives rise to the following triplets:

$$\begin{array}{lcllcl} i & \leftrightarrow & g \leftrightarrow h & \qquad d & \leftrightarrow & b \wedge c \\ g & \leftrightarrow & a \vee d & \qquad e & \leftrightarrow & a \vee b \\ h & \leftrightarrow & e \wedge f & \qquad f & \leftrightarrow & a \vee c \end{array}$$

Stålmarck's method works by placing variables in equivalences classes. Two variables a and b are in the same equivalence class, $a \sim b$, if they are always assigned the same Boolean value. We also want to keep track of

variables which have opposite values. For example, if c and d are variables with opposite values we write $\neg c \sim d$. If a variable e is assigned the value true (false) we write $e \sim 1$ ($e \sim 0$). We use 0 as an abbreviation of $\neg 1$. In the beginning all variables are placed in separate classes except the variable of the top vertex in the parse-tree. That variable is put in the equivalence class of 0 to denote the assumption that the formula is unsatisfiable. Stålmarck's method merges equivalence classes. If at any point a variable and its negation are placed in the same equivalence class, i.e. $a \sim \neg a$, then we have found a contradiction and thus proven the original formula to be a tautology.

The equivalence relation $\sim$ is, by definition, transitive, symmetric, and reflexive. It handles negations in the expected way, for example, $\neg a \sim \neg b$ implies $a \sim b$ and $\neg a \sim b$ implies $a \sim \neg b$. Moreover, if the relation $\sim$ contains a contradiction then everything relates: $a \sim \neg a$ implies $b \sim c$ and $b \sim \neg c$.

An equivalence relation $\sim$ gives rise to a set of axioms $\mathcal{C}_\sim$:

$$\mathcal{C}_\sim = \{\ \frac{}{\vdash x \sim y} \mid x \sim y\ \}, \tag{3.2}$$

where x and y are either variables, negated variables, 0 or 1.

Each triplet gives rise to a set of rules. The rules $\mathcal{A}_u$ for triplet $u \leftrightarrow l \odot h$ relate the variables u, l, and h (with and without negations) and 0 and 1 depending on the connective $\odot$. For example, in the case of $\wedge$, $\mathcal{A}_u$ are the rules in Figure 3.12. We denote the union of the rules for all triplets by $\mathcal{A}$.

$$\frac{\vdash u \sim \neg h}{\vdash h \sim \mathbf{1}} \qquad \frac{\vdash u \sim \neg l}{\vdash l \sim \mathbf{1}} \qquad \frac{\vdash u \sim \mathbf{1}}{\vdash h \sim \mathbf{1}} \qquad \frac{\vdash h \sim \mathbf{0}}{\vdash u \sim \mathbf{0}}$$

$$\frac{\vdash u \sim \neg h}{\vdash l \sim \mathbf{0}} \qquad \frac{\vdash l \sim h}{\vdash u \sim h} \qquad \frac{\vdash u \sim \mathbf{1}}{\vdash l \sim \mathbf{1}} \qquad \frac{\vdash l \sim \mathbf{1}}{\vdash u \sim h}$$

$$\frac{\vdash u \sim \neg l}{\vdash h \sim \mathbf{0}} \qquad \frac{\vdash l \sim \neg h}{\vdash u \sim \mathbf{0}} \qquad \frac{\vdash h \sim \mathbf{1}}{\vdash u \sim l} \qquad \frac{\vdash l \sim \mathbf{0}}{\vdash u \sim \mathbf{0}}$$

Figure 3.12: Rules $\mathcal{A}_u$ for the triplet $u \leftrightarrow l \wedge h$.

The zero-saturation, 0-SAT, of an equivalence relation $\sim$ is a new equivalence relation:

$$Sat(\sim, 0) = \{\ (x, y) \mid \mathcal{A} \cup \mathcal{C}_\sim \vdash x \sim y\ \}.$$

The zero-saturation of $\sim$ deduces from $\sim$ all possible relations between triplet variables based on the rules in $\mathcal{A}$. Algorithm 3.13 shows the pseudo-code for 0-SAT. The sign $\vdash_1$ in line 4 indicates application of exactly one rule in $\mathcal{A}_u$ to one axiom in $\mathcal{C}_\sim$. In line 6, we update U with the parents of x and y as well as x and y themselves (discarding any negations as well as 0 and 1).

Name: 0-SAT($\sim$)
1: $U \leftarrow$ set of all variables
2: **while** U is not empty **do**
3: remove some u from U
4: **while** $\mathcal{A}_u \cup \mathcal{C}_\sim \vdash_1 x \sim y$ and not $x \sim y$ **do**
5: update $\sim$ with $x \sim y$
6: $U \leftarrow U \cup Parents(x) \cup Parents(y) \cup \{x, y\} \backslash u$
7: **return** $\sim$

Algorithm 3.13: The 0-SAT algorithm for zero-saturation.

0-SAT is seldom enough to prove a tautology. Stålmarck adds the dilemma rule to his proof system. The dilemma rule states that if we can prove $x \sim y$ under a given assumption $u \sim 0$, and we can prove the same thing under the negated assumption $u \sim 1$, then we can prove $x \sim y$ without an assumption on u. In other words, if $\frac{\vdash u \sim 0}{\vdash x \sim y}$ and $\frac{\vdash u \sim 1}{\vdash x \sim y}$ then $\frac{}{\vdash x \sim y}$.

This leads to the $(k+1)$-SAT algorithm for $(k+1)$-saturation, where the $k+1$ indicates that the dilemma rules is used to a depth of $k+1$. Algorithm 3.14 shows the pseudo-code for $(k+1)$-SAT. For readability we use R to indicate equivalence relations.

Name: $(k+1)$-SAT(R)
1: **repeat**
2: $U \leftarrow$ set of all variables
3: $R' \leftarrow R$
4: **for all** u in U **do**
5: $R_1 \leftarrow (k)$-SAT$(R \cup (u \sim \mathbf{0}))$
6: $R_2 \leftarrow (k)$-SAT$(R \cup (u \sim \mathbf{1}))$
7: $R \leftarrow R_1 \cap R_2$
8: **until** $R = R'$
9: **return** R

Algorithm 3.14: The $(k+1)$-SAT algorithm for $(k+1)$-saturation.

When performing a $(k+1)$-saturation, we first perform a k-saturation and the resulting equivalence relation is passed to the $(k+1)$-saturation algorithm. This way we prove a formula to be a tautology using as low a saturation depth as possible, and we kick-start the higher levels of saturation instead of starting from scratch.

Definition 3.4.1 (Hardness). A formula for a tautology is said to be *n-easy* if it can be proven to be a tautology using n-saturation. The same formula is said to be *n-hard* if it cannot be proven by $(n-1)$-saturation.

Consider the distributive law in Figure 3.11. Proving the implication in the left direction is 0-easy:

$$a \vee (b \wedge c) \leftarrow (a \vee b) \wedge (a \vee c)\,.$$

The right implication gives a 1-hard formula:

$$a \vee (b \wedge c) \rightarrow (a \vee b) \wedge (a \vee c)\,.$$

In the following, we use the terminology "vertex u is assigned the valued 0 (1)" to mean that the variable for vertex u is placed in the same equivalence class as $\mathbf{0}$ ($\mathbf{1}$).

As an example of Stålmarck's method, we now prove the left-implication of the distributive law: $a \vee (b \wedge c) \leftrightarrow (a \vee b) \wedge (a \vee c)$. Figure 3.15 shows the parse-tree. We start by assigning the top vertex (i) the value 0. This is indicated on the graph by a dark-gray color. The goal is now to derive a contradiction. Since (i) is 0, (g) and (h) *must* be 0 and 1, respectively. The light-gray color of (h) indicates that is has the value 1. The 0 value of (g) implies a 0 value on (a) and (d). The value of 1 assigned to (h) implies that both (e) and (f) are 1. Since a disjunction is only 1 if at least one of the arguments is 1, then (b) has to be 1 as (e) is 1 and (a) is 0. The same argument holds for (c). However, look at vertex (d). It is a conjunction vertex assigned the value 0. But both children (b) and (c) are assigned the value 1. This leads to a contradiction, and thus we have proven the formula to be a tautology.

Lemma 3.4.2. *The rewriting rules may decrease the hardness of a formula.*

Proof. (by example) Consider the two BEDs in Figure 3.16. They represent equivalent Boolean formulae. The right BED is obtained from the left one by twice applying the rewriting rule

$$(x \wedge y) \vee (x \wedge z) = x \wedge (y \vee z)\,.$$

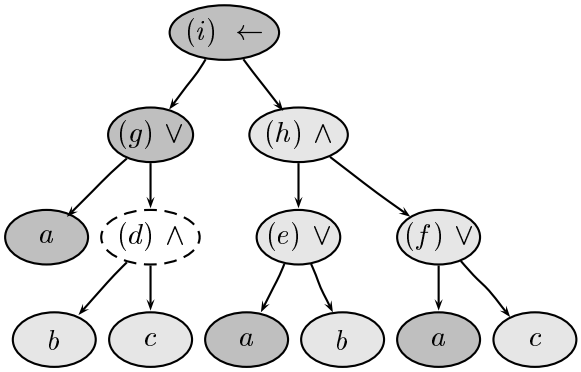

Figure 3.15: Parse-tree for $a \vee (b \wedge c) \leftarrow (a \vee b) \wedge (a \vee c)$. Each vertex is labeled with a Boolean variable. During zero-saturation, the light-gray vertices become equivalent to **1**, dark-gray vertices become equivalent to **0**, and the dashed vertex becomes equivalent to *both* **0** and **1** indicating a conflict.

Both BEDs represent tautologies, however, the left BED is 1-hard while the right BED is 0-easy. We set the top vertex in both BEDs in the same equivalence class as **0**.

A zero-saturation of the left BED results in the value 1 being assigned to the two disjunction vertices. This is as much as zero-saturation can do.

A zero-saturation of the right BED results in the value 1 being assigned to the two conjunction vertices. The 1 on the leftmost conjunction vertex propagates to 1 on both the disjunction and the negation vertex. The 1 on the rightmost conjunction vertex propagates to 1 on the b vertex. This implies a value of 0 on the negation vertex. However, the negation vertex was assumed to have the value 1. We have a contradiction and the BED must represent a tautology. □

3.4.1 Implementation of Stålmarck's Method using BEDs

Stålmarck's method works on triplets constructed from the parse-tree for a formula. BEDs offer a representation of the parse-tree for a formula where identical subexpression have been merged. A triplet in Stålmarck's method contains a Boolean connective and references to two other triplets. Substitute "vertex" for "triplet" and we have a description of operator vertices in BEDs. This similarity leads us to implement Stålmarck's method using BEDs as the underlying data structure.

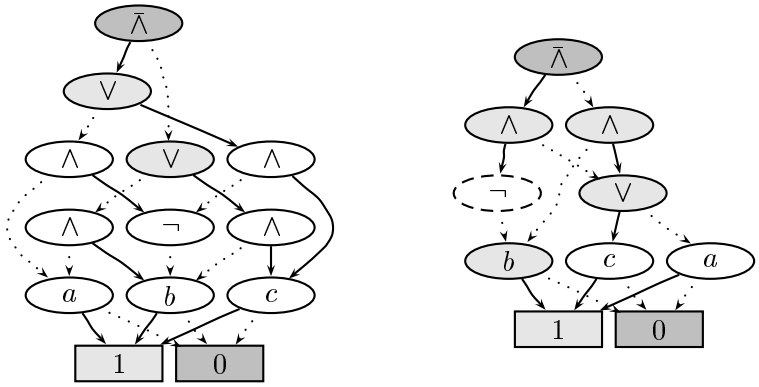

Figure 3.16: Two equivalent BEDs. The right one is obtained from the left one by twice applying the rule $(x \wedge y) \vee (x \wedge z) = x \wedge (y \vee z)$. During zero-saturation, the light-gray vertices become equivalent to **1**, dark-gray vertices become equivalent to **0**, and the dashed vertex becomes equivalent to *both* **0** and **1** indicating a conflict.

Looking over Stålmarck's algorithms for saturation of formulae, we identify the key concepts to be: application of rules for a given triplet/vertex, obtaining parents of triplets/vertices, representation of equivalence relations, and union, intersection, and equality testing of equivalence relations.

The rules can be handled by hard-coding skeletons for each of the Boolean connectives. Application of rules for a triplet is then just a function call passing the actual triplet variables as arguments.

The parents of a triplet/vertex are not readily available in BEDs. All edges are directed toward the children and not the parents. However, in a single pass through the BED structure we can label all vertices with the set of their parents. This is linear in the size of the BED and we only have to do it once.

This leaves handling equivalence relations. We use a modified version of the *disjoint sets* in [CLR90]. In our case we have a disjoint set of vertices. Each vertex has a pointer to another vertex in the same set. By avoiding cycles, we can determine whether two vertices are in the same set by repeatedly following their pointers. If we end up with two identical pointers, then the two original vertices are in the same set. We extend this notion of disjoint sets by allowing for negation marks on the pointers. If vertex a points to vertex b, then $a \sim b$. If the pointer has a negation mark, then $a \sim \neg b$. Disjoint sets gives us transitivity, symmetry, and reflexivity of our equivalence relation. The addition of negation marks gives us the correct handling of negation in the equivalence relations.

Union on disjoint sets is very efficient. It is just a matter of moving a few pointers. Handling the negations correctly in the union algorithm is rather straightforward.

Intersection between two equivalence relations is more complicated. We do it by sorting the vertices in each equivalence class in the two relations and then use a technique similar to merge-sort to construct a new equivalence relation where elements relate if and only if they relate in both of the original relations. This way of performing intersection requires a fast way to access all members of a given equivalence class. We add an extra pointer to each vertex pointing to another member of the equivalence class. Using this pointer we connect all members of an equivalence class in a cycle. It is straightforward to update this cycle-pointer when performing unions.

The order in which we iterate through the variables in the set U in Algorithm 3.14 has no influence on the results of the algorithm. However, it does have an effect on how fast we obtain the results. We have experimentally found that a most-popular-vertex-first strategy (highest fan-in and fan-out) gives good results. The reason for this is that information is spread to many neighbors fast. On our examples, other methods like top-down-breadth-first and top-down-depth-first run slower than most-popular-vertex-first.

3.5 Experimental Results

In this section we give experimental results for combinational circuit verification using BEDs. We use the two public benchmark suites ISCAS'85 and LGSynth'91[2].

All the BED experiments are performed on a 450 MHz Pentium III PC running Linux. We allocate 32 MB of memory for the BED data structure and 4 MB of memory for caches. Most of the examples can be verified in less memory but at the expense of more frequent garbage collections and thus longer runtimes.

3.5.1 The ISCAS'85 Benchmark Suite

The ISCAS'85 benchmark suite [BF85] contains a series of combinational circuits each available in two versions: the original and one with redundancies removed. The verification task is to prove that the pairwise biimplication of the outputs are tautologies. This benchmark suite has been extensively

[2]The benchmark suites are available from The Collaborative Benchmarking Laboratory (`http://www.cbl.ncsu.edu`).

used to test different methods of tautology checking and thus it makes comparisons between methods possible.

Some researchers consider the ISCAS'85 circuits too simple. This may be true for some application areas but these circuits have several properties that make them suitable as benchmark circuits for combinational verification techniques. First, despite their small size, the circuits are not easy to verify due to their functionality. For example, one of the circuits is a multiplier for which BDD techniques fail. Second, the circuits have a large logic-depth; up to 125 logic levels. This makes them nontrivial for many verification techniques. Third, the circuits are available in two functionally equivalent versions.

Table 3.3 shows the number of inputs, outputs and gates for the ten pairs of circuits. Five of the original circuits contain errors [KP94]. The software used to remove the redundancies introduced the errors. This has been corrected, but both the erroneous circuits and the correct ones are present in the benchmark suite today. This makes the ISCAS'85 suited for verifying both correct circuits and finding bugs. Table 3.4 shows how many outputs were erroneous.

	Number of			
Circuit	Inputs	Outputs	Gates	Function
`c432`, `c432nr`	36	7	433	Priority Decoder
`c499`, `c499nr`	41	32	516	ECAT
`c499`, `c1355`	41	32	868	ECAT
`c1355`, `c1355nr`	41	32	1204	ECAT
`c1908`, `c1908nr`	33	25	2134	ECAT
`c2670`, `c2670nr`	157	63	2603	ALU and Control
`c3540`, `c3540nr`	50	22	3901	ALU and Control
`c5315`, `c5315nr`	178	123	6018	ALU and Selector
`c6288`, `c6288nr`	32	32	4847	16-bit Multiplier
`c7552`, `c7552nr`	207	107	8067	ALU and Control

Table 3.3: Size and functionality of the ISCAS'85 benchmark circuits.

Table 3.5 shows the results for UP_ALL. The left column shows the runtimes in seconds using the FANIN variable ordering heuristic; the right column shows the runtimes for the DEPTH_FANOUT variable ordering heuristic. The runtimes include the time needed to calculate the ordering. The results for the FANIN ordering heuristic are in all cases the better ones. How-

Circuit	Number of Outputs Total	Erroneous
c1908, c1908nr-err	25	1
c2670, c2670nr-err	63	6
c3540, c3540nr-err	22	5
c5315, c5315nr-err	123	33
c7552, c7552nr-err	107	28

Table 3.4: Erroneous circuits from the ISCAS'85 benchmark suite.

ever, the differences are not large. Even though the FANIN heuristic gives an order of magnitude better results in some cases, there are only 10 to 15 seconds of time difference. Most of this difference stems from the fact that calculating the variable ordering using DEPTH_FANOUT is more expensive than using FANIN. Using UP_ALL it was not possible to verify the c6288 case. This is because the c6288 circuit is a 16-bit multiplier. Bryant [Bry86] has shown that there exists no good BDD variable ordering for multipliers. In [YCBO98], the BDD size of a 16-bit multiplier is given to around 40 million vertices[3]. With UP_ALL we try to construct the BDD for each of the two multipliers. With only 32 MB of memory we fail.

UP_ALL does almost as well for the erroneous circuits. Only in one case (DEPTH_FANOUT on c2670 versus c2670nr-err) is UP_ALL unable to perform the verification. In the other cases, UP_ALL takes slightly longer for the erroneous circuits than for the correct ones.

Table 3.6 shows the results for UP_ONE. Again the left column shows the runtimes in seconds using the FANIN variable ordering heuristic; the right column shows the runtimes for the DEPTH_FANOUT variable ordering heuristic. The runtimes include the time needed to calculate the ordering. The two heuristics perform just about equally well. However, using the FANIN heuristic, we are able to verify the 16-bit multiplier (c6288).

Like for UP_ALL, UP_ONE does also well for the erroneous circuits. Again the case of DEPTH_FANOUT on c2670 versus c2670nr-err is problematic. In most of the other cases, UP_ONE takes slightly longer for the erroneous circuits than for the correct ones.

Tables 3.7 and 3.8 compare the BED results with the results obtained by other methods on the same ISCAS'85 circuits. The CUDD [Som98] results are obtained on the same 450 MHz computer as the BED results. The other

[3] 40 million BDD vertices correspond to around 750 MB of memory.

Circuit	FANIN Runtime	DEPTH_FANOUT Runtime
c432, c432nr	1.7	2.2
c499, c499nr	1.8	4.0
c499, c1355	0.4	1.1
c1355, c1355nr	1.8	4.1
c1908, c1908nr	0.6	1.4
c2670, c2670nr	0.6	4.6
c3540, c3540nr	39.2	43.5
c5315, c5315nr	1.9	11.7
c6288, c6288nr	-	-
c7552, c7552nr	1.1	16.3
c1908, c1908nr-err	0.6	1.4
c2670, c2670nr-err	0.7	-
c3540, c3540nr-err	40.2	43.9
c5315, c5315nr-err	2.4	17.4
c7552, c7552nr-err	1.8	27.7

Table 3.5: ISCAS'85 results for UP_ALL using both the FANIN and the DEPTH_FANOUT ordering heuristics. The runtimes are in seconds. A dash indicates that the computation could not be done in 32 MB of memory.

Circuit	FANIN Runtime	DEPTH_FANOUT Runtime
c432, c432nr	2.1	2.5
c499, c499nr	4.3	4.1
c499, c1355	0.4	1.1
c1355, c1355nr	4.3	4.1
c1908, c1908nr	0.7	1.3
c2670, c2670nr	1.2	4.7
c3540, c3540nr	32.3	25.9
c5315, c5315nr	16.2	12.4
c6288, c6288nr	2.7	-
c7552, c7552nr	3.6	16.9
c1908, c1908nr-err	0.7	1.2
c2670, c2670nr-err	2.9	-
c3540, c3540nr-err	42.8	31.1
c5315, c5315nr-err	32.7	18.8
c7552, c7552nr-err	8.1	34.1

Table 3.6: ISCAS'85 results for UP_ONE using both the FANIN and the DEPTH_FANOUT ordering heuristics. The runtimes are in seconds. A dash indicates that the computation could not be done in 32 MB of memory.

runtimes are taken directly from the papers cited in the table. Thus those experiments are not performed on the same computer and are therefore not directly comparable. However, the runtimes still give an indication of the relative strength of the methods.

Brand [Bra93] reports the results of a slightly different verification problem. He does not compare the redundant and the non-redundant versions. Instead the circuits are synthesized and optimized, and he verifies these circuits against the originals. This may well be a more difficult verification problem.

Kunz *et. al.* [KPR96] and Pradhan *et. al.* [PPC96] use a technique based on recursive learning combined with BDDs. The reported runtimes are comparable to the BED runtimes for the smaller circuits, but for the larger circuits their method shows a degradation in performance. The BED approach is in some cases two or three orders of magnitude faster.

Matsunaga [Mat96] uses a BDD based approach to exploit the structural information on the circuits. His results are comparable to the BED results. For some examples like the `c7552` circuits, the BED approach is faster. On other examples like the `c3540` circuits, his methods is faster.

In [vE97], van Eijk uses BDDs to determine whether one node is functionally equivalent to another one. If such a pair of nodes is found, they are replaced with a free variable. His results are comparable to ours. In some cases his method is faster while in other cases our method is faster. Van Eijk also reports results for the erroneous circuits. His method performs well on these circuits except for the case of `c2670` versus `c2670nr-err` which takes much longer than the correct version.

The results of Mukherjee *et. al.* [MTJ+97] are shown in the last column of Table 3.7. They use a filtering technique to verify the circuits. Each circuit passes through a series of filters. Each filter solves part of the verification problem. In this way they combine many different methods. They obtain results comparable to the BED method. However, for the largest example (`c7552`) the BED method is much faster.

CUDD is a state-of-the-art BDD package from University of Colorado. We run version 2.3.0 in two modes: without variable reordering and with the "sift" heuristic for variable reordering. Table 3.8 shows the results. Without variable reordering, CUDD cannot complete seven of the 15 verification problems. With the "sift" heuristic, CUDD completes all but one problem: the 16-bit multiplier which is notoriously difficult for BDDs. Variable reordering has its costs. It now takes longer to complete some verification problems. We have performed an out-of-the-box test of CUDD. This means that we have not attempted to find a good initial variable ordering. We are

sure that an attempt would increase the performance of CUDD.

Circuit	BEDs	[Bra93]	[KPR96]	[PPC96]	[Mat96]	[vE97]	[MTJ+97]
c432/nr	1.7	(4.0)	1.0	2.0	0.8	0.2	0.4
c499/nr	1.8	(38.0)	1.9	5.0	1.2	0.2	0.4
c1355/nr	1.8	(9.0)	6.6	20.0	3.4	0.5	1.0
c1908/nr	0.6	(22.0)	11.2	22.0	6.2	1.6	2.1
c2670/nr	0.6	(58.0)	159.3	61.0	3.9	0.8	3.4
c3540/nr	25.9	(39.0)	67.6	281.0	17.4	3.0	12.7
c5315/nr	1.9	(29.0)	372.8	190.0	14.0	2.7	8.3
c6288/nr	2.7	(193.0)	21.5	40.0	9.1	4.3	7.2
c7552/nr	1.1	(136.0)	5583.3	412.0	20.6	34.6	20.8
c1908/err	0.6	n/a	n/a	n/a	n/a	2.5	n/a
c2670/err	0.7	n/a	n/a	n/a	n/a	54.6	n/a
c3540/err	31.1	n/a	n/a	n/a	n/a	2.9	n/a
c5315/err	2.4	n/a	n/a	n/a	n/a	8.3	n/a
c7552/err	1.8	n/a	n/a	n/a	n/a	26.2	n/a

Table 3.7: Runtimes in seconds of other approaches for verifying the ISCAS'85 benchmarks. Notice that the results of Brand [Bra93] are not directly comparable since a different verification problem is solved. "n/a" denotes that the runtime has not been reported.

3.5.2 The LGSynth'91 Benchmark Suite

The LGSynth'91 benchmark suite contains 77 multilevel combinational circuits and 40 sequential circuits. These circuits do not come in two versions, so instead we map each of the circuits to a gate library (`msu-genlib`) using SIS [S+92] and then optimize the circuits with respect to area using the SIS script `script.algebraic`. This poses two combinational verification problems, `map` and `opt`:

`map`: Verification of the correctness of the mapping by comparing the original descriptions to the mapped circuits.

`opt`: Verification of the correctness of the optimization by comparing the mapped circuits to the optimized ones.

The circuits differ considerably more in structure than the redundant and non-redundant circuits in the ISCAS'85 benchmark. The ISCAS'85 circuits are also in the LGSynth'91 benchmark suite, and while they are not the largest, they are among the more difficult of the circuits to verify using the BED techniques.

Circuit	BEDs	CUDD	CUDD (sift)
c432/nr	1.7	0.1	0.5
c499/nr	1.8	0.6	9.7
c499, c1355	0.4	0.6	37.0
c1355/nr	1.8	0.7	46.2
c1908/nr	0.6	0.7	3.2
c2670/nr	0.6	-	6.2
c3540/nr	25.9	18.9	20.7
c5315/nr	1.9	-	2.8
c6288/nr	2.7	-	-
c7552/nr	1.1	-	21.5
c1908/err	0.6	0.6	3.3
c2670/err	0.7	-	27.3
c3540/err	31.1	19.5	69.2
c5315/err	2.4	-	2.3
c7552/err	1.8	-	30.9

Table 3.8: Runtimes in seconds for CUDD for verifying the ISCAS'85 benchmarks. Runtimes are reported for no variable reordering and for the "sift" heuristic. A dash indicates that the verification could not be completed in 150 MB of memory.

Tables 3.9 and 3.10 show the results for the combinational LGSynth'91 circuits. There are 154 verification problems — two for each of the 77 circuits. We try all four combinations of UP_ONE and UP_ALL, and FANIN and DEPTH_FANOUT, and we compare with CUDD using the "sift" variable reordering heuristic. 112 of the problems are solved in less than 5 seconds by all five methods; see Table 3.9. The remaining 42 problems are the ones shown in Table 3.10.

Tables 3.11 and 3.12 show the results for the sequential LGSynth'91 circuits. There are 80 verification problems — two for each of the 40 circuits. Again we try all four combinations of UP_ONE and UP_ALL, and FANIN and DEPTH_FANOUT. The 50 problems are solved in less than 5 seconds by all five methods; see Table 3.11. The remaining 30 problems are the ones shown in Table 3.12.

Only 13 of the 234 LGSynth'91 problems are not verified in 32 MB of memory and 15 minutes by all four BED methods. Of these 13 problems, only five are not verified by any BED method. Two of the five are verified by CUDD. The remaining 221 problems are verified by all four methods; 165 of these are verified in less than 5 seconds for all methods. CUDD solves all but four problems. The following tabulars summarize the results:

Solvable by	Number out of 234
All BED methods	221
At least one BED method	229
No BED method	5
CUDD (with sift)	230

Number solved out of the 72 difficult problems	
BED faster than CUDD	39
CUDD faster than BED	27
Solved by BED, but not by CUDD	1
Solved by CUDD, but not by BED	2

Based on the results, we make a number of observations:

- BEDs and BDDs seem to agree on which problems are the difficult ones. A problem which is difficult for CUDD, is also difficult for BEDs, and vice versa. There are exceptions, for example the `C6288-map` which is easy for all BED methods but hard for CUDD.
- The FANIN heuristic performs very well with UP_ALL. The method

only takes longer than 5 seconds for six of the combinational problems (not counting the single one it could not complete). For all six problems, the FANIN / UP_ALL combination was the fastest of the five methods.

- The two methods using the DEPTH_FANOUT heuristic have generally longer runtimes than the other methods. We attribute some of it to the DEPTH_FANOUT algorithm, which is more complex and time consuming than FANIN. For example, the DEPTH_FANOUT heuristic spends around 640 seconds on computing the variable orderings for the 684 outputs of `s15850.1-map` problem. For comparison, the FANIN heuristic uses just a second or two. For problems with many variables (`s15850.1-map` has 611 input variables), the DEPTH_FANOUT heuristic takes a long time.

- For a circuit, either both the `map` and `opt` problems are easy or they are both hard. This seems to indicate that the hardness of a problem depends on the circuit and not the mapping / optimization.

- Except for `mm9b-map` and `mm9b-opt`, when CUDD is faster, the best BED method is at most 5 seconds slower. There are 17 problems for which CUDD is more than 5 seconds slower than the best BED method. (This is not counting the three cases which could only be solved by either BED or CUDD, but not both.)

3.5.3 Stålmarck's Method

To test the effectiveness of Stålmarck's method based on BEDs, we compare it to Harrison's implementation based on HOL [Har96]. Table 3.13 shows the runtimes for 14 Boolean satisfiability test cases from the Second DIMACS Challenge[4]. These examples do not fall in the category of combinational circuits. However, since Harrison has reported results for his implementation of Stålmarck's method on these examples, they are well-suited for comparison. Harrison rewrote the formulae so they only contained the following Boolean connectives: $\wedge, \leftrightarrow, \neg$. We also rewrote the formulae but to the set: $\bar{\wedge}, \vee, \rightarrow, \leftarrow, \leftrightarrow$. At the same time we performed the rewriting rules described in Section 3.2.2. These differences explain why our degree of hardness differs for a couple of the test cases. Our implementation of Stålmarck's method is the fastest by up to an order or two of magnitude. This is also expected as

[4] See `http://mat.gsia.cmu.edu/challenge.html`.

9symml-map	9symml-opt	alu2-map	alu2-opt
alu4-map	alu4-opt	apex6-map	apex7-map
apex7-opt	b1-map	b1-opt	b9-map
b9-opt	C17-map	C17-opt	C880-map
c8-map	c8-opt	cc-map	cc-opt
cht-map	cht-opt	cm138a-map	cm138a-opt
cm150a-map	cm150a-opt	cm151a-map	cm151a-opt
cm152a-map	cm152a-opt	cm162a-map	cm162a-opt
cm163a-map	cm163a-opt	cm42a-map	cm42a-opt
cm82a-map	cm82a-opt	cm85a-map	cm85a-opt
cmb-map	cmb-opt	comp-map	comp-opt
cordic-map	cordic-opt	count-map	count-opt
cu-map	cu-opt	decod-map	decod-opt
example2-map	example2-opt	f51m-map	f51m-opt
frg1-map	frg1-opt	i1-map	i1-opt
i2-map	i2-opt	i3-map	i3-opt
i4-map	i4-opt	i5-map	i5-opt
i6-map	i6-opt	lal-map	lal-opt
majority-map	majority-opt	mux-map	mux-opt
my_adder-map	my_adder-opt	parity-map	parity-opt
pcle-map	pcler8-map	pcler8-opt	pcle-opt
pm1-map	pm1-opt	sct-map	sct-opt
t481-map	t481-opt	tcon-map	tcon-opt
term1-map	term1-opt	t-map	t-opt
ttt2-map	ttt2-opt	unreg-map	unreg-opt
vda-map	vda-opt	x1-map	x1-opt
x2-map	x2-opt	x3-map	x3-opt
x4-map	x4-opt	z4ml-map	z4ml-opt

Table 3.9: The 112 combinational LGSynth'91 circuit problems which are verified in less than five seconds by all four BED methods and by CUDD (with the "sift" variable reordering heurstic).

Circuit	Up_One Fanin	Up_One Depth_Fanout	Up_All Fanin	Up_All Depth_Fanout	CUDD (sift)
apex6-opt	2.4	5.8	2.4	5.6	0.5
C1355-map	8.7	9.3	2.0	4.1	20.9
C1355-opt	11.9	12.0	3.1	6.4	34.2
C1908-map	5.0	3.6	3.1	3.7	7.0
C1908-opt	4.9	4.2	2.5	3.7	5.0
C2670-map	2.6	17.3	1.3	14.2	6.9
C2670-opt	2.7	68.2	1.2	13.6	9.5
C3540-map	46.7	53.0	30.5	66.2	89.3
C3540-opt	28.8	62.8	13.8	46.9	75.6
C432-map	9.6	10.2	4.6	5.5	0.5
C432-opt	8.4	9.0	4.8	5.3	0.6
C499-map	8.7	9.4	2.0	4.1	17.2
C499-opt	13.3	12.7	2.9	6.4	18.8
C5315-map	50.7	32.5	3.1	29.9	3.0
C5315-opt	119.0	33.8	3.2	31.4	3.5
C6288-map	0.6	1.2	0.5	1.3	-
C6288-opt	-	-	-	-	-
C7552-map	8.7	41.4	2.2	36.6	30.0
C7552-opt	9.3	43.6	2.2	37.9	30.8
C880-opt	2.1	3.3	1.0	2.3	11.9
dalu-map	1.7	13.3	1.1	13.1	2.6
dalu-opt	1.8	10.7	1.1	10.2	2.8
des-map	6.8	62.0	5.4	58.8	7.3
des-opt	6.6	64.3	5.4	61.3	6.9
frg2-map	3.4	14.1	2.9	13.3	0.7
frg2-opt	3.2	13.2	2.8	12.3	0.8
i10-map	35.4	151.9	8.3	112.1	56.2
i10-opt	37.8	154.2	8.3	106.0	41.7
i7-map	1.9	5.6	1.8	5.4	0.1
i7-opt	2.0	5.9	1.8	5.7	0.1
i8-map	3.0	19.3	2.5	19.3	4.0
i8-opt	2.8	16.8	2.4	16.8	2.5
i9-map	2.2	9.1	1.8	9.1	0.5
i9-opt	2.4	9.1	1.9	9.0	0.3
k2-map	1.7	7.1	1.6	7.1	1.1
k2-opt	1.8	7.2	1.8	7.2	1.0
pair-map	4.1	19.9	3.1	19.7	8.1
pair-opt	4.4	21.7	3.2	20.6	8.1
rot-map	4.0	13.2	2.2	11.5	3.7
rot-opt	3.0	9.7	2.0	8.8	2.9
too_large-map	19.1	4.6	0.9	3.6	3.6
too_large-opt	1.4	1.4	0.5	1.2	1.1

Table 3.10: Runtimes in seconds for the combinational LGSynth'91 circuits using all four BED methods and CUDD (with the "sift" variable reordering heuristic). Both the results for the `map` and the `opt` verification problems are shown. Of the 154 verification problems (77 circuits with two problems each), only the 42 ones with runtimes longer than 5 seconds are shown. A dash indicates that the computation could not be completed in 32 MB of memory and 15 minutes.

mm4a-map	mm4a-opt	mult16a-map	mult16a-opt
mult16b-map	mult16b-opt	mult32b-map	mult32b-opt
s1196-map	s1196-opt	s1488-map	s1488-opt
s1494-map	s1494-opt	s208.1-map	s208.1-opt
s27-map	s27-opt	s298-map	s298-opt
s344-map	s344-opt	s349-map	s349-opt
s382-map	s382-opt	s386-map	s386-opt
s400-map	s400-opt	s420.1-map	s420.1-opt
s444-map	s444-opt	s510-map	s510-opt
s526-map	s526-opt	s641-map	s641-opt
s713-map	s713-opt	s820-map	s820-opt
s832-map	s832-opt	s838.1-map	s838.1-opt
sbc-map	sbc-opt		

Table 3.11: The 50 sequential LGSynth'91 circuit problems which are verified in less than five seconds by all four BED methods and by CUDD (with the "sift" variable reordering heuristic).

we use a faster computer and a compiled language, whereas Harrison uses an interpreted language. In his paper, Harrison estimates the speedup for a compiled implementation to be between several times and 50 times faster.

Table 3.14 shows the runtimes for the ISCAS'85 benchmark suite. Most outputs are zero- or one-easy. Some are two-hard. For c3540, 17 outputs are two-hard. For c2670 and c7552, only one output is two-hard. For c5315, 13 outputs are two-hard. The two-hard outputs take up the vast majority of the verification time.

The Up_One algorithm allows for a gradual transformation of a BED into a BDD. By breaking off this transformation and applying Stålmarck's method to the intermediate form, we can combine the two methods. We use the minimization strategy described in Section 3.2.3. We perform an Up_One with each variable and keep the result if it is reduces the number of vertices. We repeat this until we have reached a local minimum. The order in which we iterate through the variables is found by a depth-first traversal of the circuit. Table 3.15 shows runtimes for the ISCAS'85 circuits with and without minimizing.

It is clear that our implementation of Stålmarck's method does not compare favorably in terms of runtime with the other methods. However, Stålmarck's method uses only a few megabytes of memory, and the amount of memory is linear in the size of the circuit (given a fixed saturation level).

Circuit	Up_One Fanin	Up_One Depth_Fanout	Up_All Fanin	Up_All Depth_Fanout	CUDD (sift)
bigkey-map	10.0	114.6	6.0	115.0	3.0
bigkey-opt	10.3	107.3	5.9	119.3	2.7
clma-map	3.9	11.4	2.9	10.7	4.5
clma-opt	9.2	179.9	5.2	181.3	8.1
clmb-map	2.1	5.8	1.7	5.6	4.1
clmb-opt	7.1	176.8	4.0	177.5	8.3
dsip-map	9.3	88.9	5.5	84.5	3.5
dsip-opt	9.3	90.1	5.6	87.0	3.5
mm30a-map	-	-	-	-	146.9
mm30a-opt	-	-	-	-	21.2
mm9a-map	-	5.5	143.4	3.9	2.1
mm9a-opt	-	7.5	123.1	5.5	6.2
mm9b-map	-	37.8	-	331.3	2.7
mm9b-opt	-	33.5	-	28.7	2.8
mult32a-map	3.00	5.8	1.4	5.1	0.6
mult32a-opt	2.9	6.0	1.3	5.1	0.5
s13207.1-map	19.1	161.6	5.4	145.5	21.3
s13207.1-opt	20.5	176.5	6.3	159.4	23.3
s1423-map	6.5	7.2	1.9	7.0	1.2
s1423-opt	7.3	11.8	2.5	10.5	1.3
s15850.1-map	46.6	698.8	15.8	695.2	22.7
s15850.1-opt	88.0	696.9	19.2	680.0	42.8
s38417-map	-	-	-	-	-
s38417-opt	-	-	-	-	-
s38584.1-map	175.0	-	22.4	-	62.6
s38584.1-opt	181.4	-	29.3	-	77.1
s5378-map	5.2	26.1	3.5	24.3	1.3
s5378-opt	4.2	25.1	3.4	24.1	0.7
s9234.1-map	15.6	-	4.5	127.2	3.6
s9234.1-opt	18.9	-	5.7	127.3	5.0

Table 3.12: Runtimes in seconds for the sequential LGSynth'91 circuits using both BED methods and CUDD (with the "sift" variable reordering heuristic). Both the results for the `map` and the `opt` verification problems are shown. Of the 80 verification problems (40 circuits with two problems each), only the 30 ones with runtimes longer than 5 seconds are shown. A dash indicates that the computation could not be completed in 32 MB of memory and 15 minutes.

DIMACS	BED+Stålmarck [sec]	Harrison [sec]	Degree of hardness
aim-50-1_6-no-1	3.1	8.2	2 / 2
aim-50-1_6-no-2	0.4	14.1	1 / 2
aim-50-1_6-no-3	1.2	15.3	1 / 1
aim-50-1_6-no-4	0.4	5.9	1 / 1
aim-50-2_0-no-1	0.9	20.5	1 / 1
aim-50-2_0-no-2	0.5	28.7	1 / 1
aim-50-2_0-no-3	0.6	13.9	1 / 1
aim-50-2_0-no-4	0.6	29.1	1 / 1
aim-100-1_6-no-3	2.4	135.3	1 / 2
aim-100-2_0-no-1	2.4	12.9	1 / 1
aim-100-2_0-no-2	2.5	14.6	1 / 1
dubois20	66.9	377.0	2 / 2
jnh211	15.0	1762.0	1 / 1
ssa0432-003	8.5	490.1	1 / 1

Table 3.13: Boolean satisfiability test cases from the Second DIMACS Challenge. The test cases, all being not satisfiable, have been negated to form tautologies. The table shows the runtimes for our C implementation run on a 450 MHz Pentium III and the runtimes for Harrison's CAML Light implementation run on a Sparc 10 [Har96]. The last column shows the degree of hardness for our implementation (first number) and for Harrison's (second number).

Circuit	Stålmarck Hardness	Stålmarck Runtime	Up_One / Up_All Runtime
`c432`, `c432nr`	1	0.3	1.7
`c499`, `c499nr`	1	0.2	1.8
`c1355`, `c1355nr`	1	0.2	1.8
`c1908`, `c1908nr`	1	0.2	0.6
`c2670`, `c2670nr`	2	25.7	0.6
`c3540`, `c3540nr`	2	63529	25.9
`c5315`, `c5315nr`	2	1328	1.9
`c6288`, `c6288nr`	1	1427	2.7
`c7552`, `c7552nr`	2	57.6	1.1

Table 3.14: ISCAS'85 results for Stålmarck's method on BEDs compared to the Up_One / Up_All method. The degree of hardness is for the hardest output. The runtimes are in seconds.

ISCAS'85	No Minimize [sec]	With Minimize [sec]
c432, c432nr	0.3	0.3
c499, c499nr	0.2	0.3
c1355, c1355nr	0.2	0.3
c1908, c1908nr	0.2	0.1
c2670, c2670nr	25.7	96.0
c3540, c3540nr	63529	60932
c5315, c5315nr	1328	95.1
c6288, c6288nr	1427	0.5
c7552, c7552nr	57.6	66.0

Table 3.15: Verification results for Stålmarck's method on the ISCAS'85 benchmark suite with and without minimizing the formulae first. The runtimes in the second column includes the runtimes for the minimization algorithm.

The other methods (including BEDs with UP_ONE and UP_ALL) rely on BDDs or BDD-like representations where the worst-case size is exponential in the size of the circuit.

The main advantage of implementing Stålmarck's method in a BED framework is the additional availability of BED and BDD methods for tautology checking. Standard BED and BDD methods typically run out of memory before they run out of time. Stålmarck's method has it the other way around. Combining Stålmarck's method with BEDs gives the user the choice between time and memory.

3.6 Related Work

Current approaches for equivalence checking of combinational circuits can be classified into two categories: functional and structural.

The functional methods consist of representing a circuit as a canonical decision diagram. Two circuits are equivalent if and only if their decision diagrams are equal (isomorphic). To overcome some of the limitations of BDDs, a number of more expressive, yet still canonical, decision diagrams have been proposed. One can use other types of decomposition rules [DST+94, KSR92], relax the variable ordering restriction [GM94, GKB97, JBA+97, SW95], or extend the domains and/or ranges to integers instead of Booleans [BFG+93, BC95, CMZ+93]. These extensions are typically targeted to solving a par-

ticular class of problems, e.g., being able to represent the multiplication function. The canonical representations all have worst case exponential size (since the tautology problem can be solved in constant time), thus they are all exponentially less compact than BEDs.

The structural methods exploit similarities between the two circuits that are compared by identifying related nodes in the circuits and using this information to simplify the verification problem. These techniques rely on the observation that if two circuits are structurally similar, they have a large number of internal nodes that are functionally equivalent (typically, for more than 80% of the nodes in one circuit, there exists a node in the other circuit which is functionally equivalent [KK97]). This observation is used in several ways. Brand [Bra93] uses a test generator for determining whether one node can be replaced by another in a given context (the nodes need not necessarily be functionally equivalent as long as the difference cannot be observed at the primary output). If so, the replacement is carried out. In this way, one circuit is gradually transformed into the other. The key problem is to find a sufficiently large number of pairs, yet avoid having to spend time testing all possible pairs of nodes. Several heuristics are used to select candidate pairs of nodes to check, e.g., using the labeling of nodes and the results of simulation.

Test generation techniques are also the basis for the recursive learning technique for finding logical implications between nodes in the circuits by Kunz *et. al.* [Kun93, KP94]. To enable the verification of larger circuits, the recursive learning techniques can be combined with BDDs [KPR96, PPC96]. The learning technique is further extended by Jain *et. al.* [JMF95] and by Matsunaga [Mat96], introducing more general learning methods based on BDDs and better heuristics for finding cuts in the circuits to split the verification problem into more manageable sizes. Recursive learning is closely related to Stålmarck's method [SS98]. Both methods work by performing a number of 0/1 splits and then combine the knowledge learned in the two cases. For a conjunctive normal form (CNF) formula, the difference between Stålmarck's method and recursive learning is that for the former you split on the variables while in the latter you split on the clauses.

Van Eijk and Janssen [vEJ94, vE97] use the canonicity of BDDs to determined whether one node is functionally equivalent to another. If two nodes are found to be identical, they are replaced with a new, free variable. Heuristics are used to select candidate pairs of nodes to check for equivalence. The main problem with this technique is to manage the BDD sizes when eliminating false negatives (when re-substituting BDDs for the introduced free variables).

Cerny and Mauras [CM90] present another technique for comparing two circuits without representing their full functionality. A relation that represents the possible combinations of logic values at a given cut is propagated through the two circuits. A key problem with this and the other cut-based techniques [KPR96, Mat96, PPC96, vE97, vEJ94, JMF95] is that the performance is very sensitive to how the cuts are chosen and there is no generally applicable method to chose appropriate cuts.

The technique by Kuehlmann and Krohm [KK97] and later by Ganai and Kuehlmann [GK00] represents a recent development of the structural methods, combining several of the above techniques and developing better heuristics for determining cuts. Kuehlmann, Krohm and Ganai represent the combinational circuits using a non-canonical data structure which is similar to BEDs except that only conjunction and negation operators are used. This data structure is only used to identify isomorphic sub-circuits since no operator reductions are performed. We believe that the structural technique by Kuehlmann, Krohm and Ganai would benefit significantly from replacing the used circuit representation with BEDs.

BEDs can be seen as an intermediate representation between the compact circuits and the canonical BDDs. Compared to the functional techniques, BEDs are capable of exploiting equivalences of the two circuits and the performance is provably no worse than when using BDDs. Compared to the structural techniques, BEDs only have a limited capability to find equivalences between pairs of nodes (since only local operator reduction rules are included). Combining BEDs with structural techniques would be beneficial since information about equivalent nodes immediately reduce the size of the BED and make even further identifications of nodes possible.

Richards [Ric98] has some interesting notes on a tautology checker loosely related to Stålmarck's algorithm. Whereas Stålmarck's method uses relations over two variables, Richards proposes relations over more than two variables. His approach reduces the recursive depth (the "hardness") of problems and it increases the number of inference rules. The inference rules become quite simple to implement; typically simple masking operations or table lookups.

During the last three decades the AI community has worked on developing efficient satisfiability checkers. They could in principle be used to solve the equivalence problem for combinational circuits. However, comparisons between algorithms based on the prominent Davis-Putnam algorithm and BDDs show that although efficient for typical AI problems, they are quite inferior to BDDs on circuits [US94]. Our own experiments with Stålmarck's method support this. However, satisfiability checkers are very memory effi-

cient so one should not completely discard them.

3.7 Conclusion

In this chapter we have shown how to use BEDs in equivalence checking of flat combinational circuits. First, we introduced a set of methods for reducing the size of BEDs. The methods include:

- Restricting the allowed operators in BEDs to a small set of operators.
- Rewriting rules for local reductions of the BED data structure.
- A number of methods for restructuring BEDs.

Our experiments show that the use of such size-reducing methods is important for the performance of BEDs. Especially the UP_ONE algorithm depends on them.

Second, we discussed the variable ordering problem. Just like BDDs, BEDs are sensitive to the order in which the variables are pulled up using UP_ONE. We examined two ordering heuristics developed for BDDs. Both gave good results when used on BEDs.

Third, we examined Stålmarck's method for solving the tautology problem. We discussed the method and illustrated how to implement it on top of the BED data structure. The method works for combinational circuit verification, but it is much slower than UP_ONE and UP_ALL. However, Stålmarck's method uses little memory.

Finally, we experimentally tested the BEDs on a number of combinational circuit verification problems. The BEDs compare favorably with other methods both in terms of time and memory usage. Of 234 equivalence checking problems from the LGSynth'91 suite, only five could not be verified when using 32 MB of memory. The verification technique is almost push-button for the user: two methods (UP_ONE and UP_ALL) and two heuristics (FANIN and DEPTH_FANOUT), and more than half of the problems are solvable in less than 5 seconds by any one of the four combinations. Especially the combination of DEPTH_FANOUT and UP_ALL performs well on combinational circuits.

Chapter 4

Equivalence Checking of Hierarchical Combinational Circuits

In this chapter we present a method for verifying that two hierarchical combinational circuits implement the same Boolean functions. The key new feature of the method is its ability to exploit the modularity of the circuits to reuse results obtained from one part of the circuits in other parts. We demonstrate the method on large adder and multiplier circuits. This chapter is based on the paper [WHA99].

4.1 Introduction

Due to the increase in the complexity of design automation tools and the circuits they manipulate, such tools cannot in general be assumed to be correct. Instead of attempting to formally verify the design automation tools, a more practical approach is to formally check that a circuit generated by a design automation tool functionally corresponds to the original input. This chapter presents a technique for formally verifying that two hierarchical combinational circuits implement the same Boolean functions. The presented technique can also be used to check manual modifications of a circuit to ensure that the designer has not introduced errors.

We use a *hierarchical* model of combinational circuits as opposed to the *flat* model used in Chapter 3. Based on this hierarchical model, we show how to propagate a cut through two circuits from the inputs to the outputs. The key new feature of the method is its ability to reuse previously

cell fa(**in** x_0, x_1, x_2; **out** u_0, u_1) {
    $u_0 := x_0 \oplus x_1 \oplus x_2$
    $u_1 := (\ x_0 \wedge x_1) \vee (x_2 \wedge (x_0 \vee x_1\ )\ )$
}
cell *4bitadder*(**in** $s_0, \ldots, s_8$;
                out $s_9, s_{11}, s_{13}, s_{15}, s_{16}$) {
    var s_{10}, s_{12}, s_{14}
    instance $fa(s_1, s_2, s_0; s_9, s_{10})$
    instance $fa(s_3, s_4, s_{10}; s_{11}, s_{12})$
    instance $fa(s_5, s_6, s_{12}; s_{13}, s_{14})$
    instance $fa(s_7, s_8, s_{14}; s_{15}, s_{16})$
}

Figure 4.1: A 4-bit adder consisting of a full-adder cell and a 4-bit adder cell with four instantiations of the full-adder cell.

calculated results in the verification. Consider the 4-bit adder in Figure 4.1. The description consists of two cells; a full-adder cell, *fa*, and a 4-bit adder cell, *4bitadder*, containing four instantiations of the full-adder cell and a description of how they are interconnected. The traditional way of verifying hierarchical combinational circuits is to flatten them into a single block of combinational logic on which the verification is performed. In case of complex circuits, this method is not feasible. Our method attempts to work on one cell, and then reuse information about this cell whenever possible.

The 4-bit adder circuit described above corresponds to the top circuit in Figure 4.2. The bottom circuit in the figure is also a 4-bit adder, but with two instantiations of two different full-adder cells which negate either the inputs or the outputs.

Our method compares the full-adder from the top circuit with each of the two different full-adders in the bottom circuit and combines the results to prove that the two circuits are indeed identical (except for some negated inputs and outputs). The method is automatic as it requires no human interaction during the verification process. If the adders in Figure 4.2 were larger, our method would still only consider two comparisons between full-adders. The rest of the verification would reuse the comparisons to prove the equivalence.

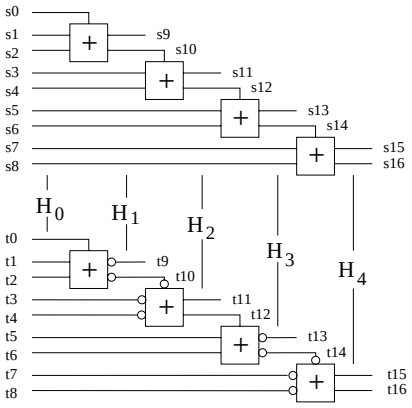

Figure 4.2: Two 4-bit adders.

4.2 Hierarchical Combinational Circuits

Most circuit description languages, for example Berkeley Logic Interchange Format (BLIF), contain language constructs for modular circuit descriptions. Modules may contain other modules yielding a hierarchical description. This leads to a model of *hierarchical combinational circuits* based on cells, instantiations of cells, and connecting wires. There are two types of cells: those that contain instantiations of other cells, *container cells*, and those that contain logic gates, *logic cells*. Definition 4.2.1, 4.2.2, and 4.2.3 define a mathematical model for hierarchical combinational circuits based on these observations.

Definition 4.2.1 (HCC). A *hierarchical combinational circuit* (HCC) is a pair (C, c), where C is a set of *cells* and $c \in C$ is the top cell.

For example, Figure 4.1 describes an HCC (C, c), where $C = \{fa, 4bitadder\}$ and $c = 4bitadder$.

Definition 4.2.2 (Cell). A *cell* c has the following attributes:

$Vars(c)$, a set of Boolean variables,

$In(c) \subseteq Vars(c)$, a list of input variables[1],

$Out(c) \subseteq Vars(c)$, a list of output variables,

and either

[1]We shall use set notation on lists in situations where we refer to the set of elements from a list.

$Inst(c)$, a list of *instantiations*, or

$Fct(c)$, a list of Boolean functions $In(c) \to \mathbb{B}$, one function for each output.

Container cells have the *Inst* attribute while logic cells have the *Fct* attribute. In the 4-bit adder circuit in Figure 4.1 there are two cells; the full-adder cell, *fa*, and the 4-bit adder cell, *4bitadder*. $Vars(fa)$ is the set $\{x_0, x_1, x_2, u_0, u_1\}$. $In(fa)$ is the list $[x_0, x_1, x_2]$, and $Out(fa)$ is the list $[u_0, u_1]$. The full-adder cell has the *Fct* attribute; a list with two elements where the first element is the function for the u_0 output: $x_0 \oplus x_1 \oplus x_2$. The *4bitadder* cell has the *Inst* attribute; a list of four instantiations of *fa*.

Definition 4.2.3 (Instantiation). An *instantiation* i of a cell c has the following attributes:

$cell(i)$, the cell c of which i is an instantiation,

$par(i)$, the cell in which i is located ($i \in Inst(par(i))$),

$in(i) \subseteq Vars(par(i))$, a list of input variables,

$out(i) \subseteq Vars(par(i))$, a list of output variables.

The instantiation i must further fulfill the requirements:

$$|in(i)| = |In(cell(i))| \quad \text{and} \quad |out(i)| = |Out(cell(i)|\,.$$

The topmost instantiation of a full-adder cell in the 4-bit adder example has $cell(i) = fa$, $par(i) = 4bitadder$, $in(i) = [s_1, s_2, s_0]$, and $out(i) = [s_9, s_{10}]$.

The outputs of a hierarchical combinational circuit are determined by the inputs. In the case of the 4-bit adder, the outputs are the sum of two 4-bit numbers on the inputs. We use a relation $Rel(c)$ to capture this relation between the inputs and the outputs of a cell c. For a logic cell, *Rel* is determined by the logic of the gates (the *Fct* attribute):

$$Rel(c) = \bigwedge_{k=1,\ldots,|Out(c)|} (Out(c)_k \leftrightarrow Fct(c)_k)\,.$$

(We use characteristic functions to represent relations.) The subscript k indicates the k'th element in a list. For example, $Rel(fa)$ is the relation:

$$\begin{aligned}&(u_0 \leftrightarrow x_0 \oplus x_1 \oplus x_2) \wedge \\ &\quad (u_1 \leftrightarrow (x_0 \wedge x_1) \vee (x_2 \wedge (x_0 \vee x_1)))\,.\end{aligned}$$

For container cells, Rel is determined as:

$$Rel(c) = \exists v_{\in V} : \bigwedge_{i \in Inst(c)} Rel(cell(i))[Map]\,,$$

where V is the set variables which are neither inputs nor outputs, i.e., $V = Vars(c) \backslash (In(c) \cup Out(c))$, and $[Map]$ is a renaming of $In(cell(i))$ and $Out(cell(i))$ variables to $in(i)$ and $out(i)$ variables, respectively. The notation $\exists v_{\in V}$ for $V = \{v_1, v_2, \ldots, v_k\}$ is shorthand for $\exists v_1 : \exists v_2 : \cdots : \exists v_k$.

For an HCC (C, c), the relation over the primary inputs and the primary outputs is $Rel(c)$.

We now define a *path* and a *cut* in a cell.

Definition 4.2.4 (Path). For a container cell c, a *path* $p = \langle p_1, \ldots, p_n \rangle$ is a sequence of variables from $Vars(c)$ such that for all k, $1 \leq k < n$, there exists an instantiation $i \in Inst(c)$, such that $p_k \in in(i)$ and $p_{k+1} \in out(i)$.

Definition 4.2.5 (Cut). A *cut* K in a container cell c is a set of variables from $Vars(c)$ such that any path $p = \langle p_1, \ldots, p_n \rangle$ through c with $p_1 \in In(c)$ and $p_n \in Out(c)$ contains exactly one variable from the cut K.

For both logic and container cells, the *input cut* is the set $In(i)$ and the *output cut* is the set $Out(i)$.

The set $\{s_3, s_4, \ldots, s_{10}\}$ is a cut in the *4bitadder* container cell. For two cuts in different HCCs we define a *cut-relation* H as a relation over the values of the variables in the cuts.

Definition 4.2.6 (Cut-relation). A *cut-relation* H between two cuts K_1 and K_2 in two cells is a relation over the values of the variables in K_1 and K_2, i.e., $H \subseteq \mathbb{B}^{K_1 \cup K_2}$.

A cut-relation over the input cuts of the two circuits in Figure 4.2 could be:

$$\bigwedge_{i=0,1,2,5,6} (s_i \leftrightarrow t_i) \;\wedge\; \bigwedge_{i=3,4,7,8} (s_i \leftrightarrow \neg t_i)\,, \tag{4.1}$$

stating that s_i and t_i have identical values for $i = 0, 1, 2, 5, 6$, and that s_i and t_i have opposite values for $i = 3, 4, 7, 8$.

We call a cut-relation between input cuts in two cells for an *input relation*. Likewise, we call a cut-relation between output cuts for an *output relation*.

Given a cut-relation H for two cuts K_1 and K_2, and two instantiations i_1 and i_2 such that the input variables of i_1 and i_2 are subsets of K_1 and K_2, respectively, we can determine a relation R_{in} between the input variables of i_1 and i_2:

$$R_{in} = (\exists v_{\in (K_1 \cup K_2) \setminus (in(i_1) \cup in(i_2))} : H)[Map]\,, \tag{4.2}$$

where $[Map]$ is a renaming of $in(i_1)$, $in(i_2)$, $out(i_1)$, and $out(i_2)$ variables to $In(cell(i_1))$, $In(cell(i_2))$, $Out(cell(i_1))$, and $Out(cell(i_2))$ variables, respectively.

4.3 Cut Propagation

Given two hierarchical combinational circuits HCC_1 (C_1, c_1) and HCC_2 (C_2, c_2), and an input relation H_{in}, the verification problem we consider is to determine whether the outputs satisfy a desired relation H_{out}. Typically, H_{in} and H_{out} would represent "the circuits have identical inputs and outputs."

The verification algorithm works by propagating a cut-relation from the inputs to the outputs. Let H_0 be the input relation H_{in}, a cut-relation between the input cuts of c_1 and c_2. We move their cut-relation past instantiations of cells in c_1 and c_2 (assuming that c_1 and c_2 are container cells). In each step we calculate a new cut-relation, H_{k+1}, based on the previous one, H_k. When the cut-relation has reached the outputs, the resulting cut-relation, H_n, relates the outputs of c_1 to the outputs of c_2. If H_n is a subset of H_{out}, $H_n \subseteq H_{out}$, the circuits have the desired output relation.

4.3.1 Example

Before describing the algorithm in detail, we give an example to illustrate the basic ideas. Consider again the two different implementations of 4-bit adders in Figure 4.2. The 4-bit adders are described using s and t variables, respectively. The full-adders in the top circuit are described using x (input) and u (output) variables, while the full-adders in the bottom circuit use y and v variables. The H's represent the cut-relations and the vertical lines indicate the cuts.

Assume H_0 in Figure 4.2 is given by (4.1). We decide to move the cuts from the inputs to the outputs one full-adder at a time and to move the cuts in the two circuits simultaneously. First we calculate the input relation $R_{in,1}$ between the leftmost full-adder cell in each of the two circuits using (4.2):

$$R_{in,1} = (x_0 \leftrightarrow y_0) \ \wedge \ (x_1 \leftrightarrow y_1) \ \wedge \ (x_2 \leftrightarrow y_2)\,. \tag{4.3}$$

Notice the use of cell variables x and y and not instantiation variables s and t, which is important in order to recognize this situation in the future.

Given $R_{in,1}$ and the input/output relation Rel for the two full-adders, we can determine the relation between the outputs (we will show later how to do this):

$$R_{out,1} = (u_0 \leftrightarrow \neg v_0) \ \wedge \ (u_1 \leftrightarrow \neg v_1)\,. \tag{4.4}$$

We move the cuts and determine the new cut-relation H_1 based on $R_{out,1}$ (again, we will show later how to do this):

$$\begin{aligned} H_1 = \quad & \textstyle\bigwedge_{i=5,6}(s_i \leftrightarrow t_i) \ \wedge \\ & \textstyle\bigwedge_{i=3,4,7,8,9,10}(s_i \leftrightarrow \neg t_i)\,. \end{aligned} \tag{4.5}$$

In a similar way we propagate the cuts one step further getting H_2:

$$H_2 = \textstyle\bigwedge_{i=5,6,11,12}(s_i \leftrightarrow t_i) \ \wedge \ \bigwedge_{i=7,8,9}(s_i \leftrightarrow \neg t_i)\,.$$

In the third step, we start by finding the input relation $R_{in,3}$:

$$R_{in,3} = (x_0 \leftrightarrow y_0) \ \wedge \ (x_1 \leftrightarrow y_1) \ \wedge \ (x_2 \leftrightarrow y_2)\,.$$

This is identical to the relation $R_{in,1}$ from the first step. The full-adders in the third step are also identical to the full-adders in the first step, and thus we can immediately reuse the output relation $R_{out,1}$ instead of calculating $R_{out,3}$. We update the H_2 relation using $R_{out,1}$ and obtain H_3:

$$\begin{aligned} H_3 = \quad & (s_9 \leftrightarrow \neg t_9) \ \wedge \ (s_{11} \leftrightarrow t_{11}) \ \wedge \\ & (s_{13} \leftrightarrow \neg t_{13}) \ \wedge \ (s_{14} \leftrightarrow \neg t_{14})\,. \end{aligned}$$

Similarly, in the fourth step the input relation is found to be identical to that of the second step, and the full-adders in the second and the fourth step are identical. We update the relation H_3, and get the final output relation H_4:

$$\begin{aligned} H_4 = \quad & (s_9 \leftrightarrow \neg t_9) \ \wedge \ (s_{11} \leftrightarrow t_{11}) \ \wedge \ (s_{13} \leftrightarrow \neg t_{13}) \\ & \wedge \ (s_{15} \leftrightarrow t_{15}) \ \wedge \ (s_{16} \leftrightarrow t_{16})\,. \end{aligned}$$

We observe that the sum-bits of the first and third pair of adders have opposite values while the sum-bits of the second and fourth pair of adders are pairwise equivalent.

4.3.2 Moving Cuts

We distinguish between two ways of moving cuts: BUILD and PROPAGATE. BUILD determines the input/output relation *Rel* for a cell c and uses it to calculate the new cut-relation by moving the cut past an instantiation of cell c. PROPAGATE moves cuts past two cells simultaneously by calculating the input relation R_{in} between the inputs of the two cells and from that calculate the output relation R_{out} for the same pair of cells. In the example above we only used PROPAGATE.

Algorithm 4.3 shows the pseudo-code for the overall algorithm PROP. The algorithm moves a cut through two container cells by for each step selecting either the BUILD or the PROPAGATE algorithm. In the example,

Name: PROP(H, c_1, c_2)
Require: c_1 and c_2 are container cells
1: $(K_1, K_2) \leftarrow$ input cut for (c_1,c_2)
2: **while** K_1, K_2 are not output cuts **do**
3: Select method
4: **if** method is build instantiation $i \in Inst(c_1)$ **then**
5: $(H, K_1) \leftarrow$ BUILD(i, H, K_1)
6: **else if** method is build instantiation $i \in Inst(c_2)$ **then**
7: $(H, K_2) \leftarrow$ BUILD(i, H, K_2)
8: **else if** method is propagate instantiations $i_1 \in Inst(c_1)$ and $i_2 \in Inst(c_2)$ **then**
9: $(H, K_1, K_2) \leftarrow$ PROPAGATE(i_1, i_2, H, K_1, K_2)
10: **return** H

Algorithm 4.3: The PROP(H, c_1, c_2) algorithm. H is a input relation between container cells c_1 and c_2. PROP returns the output relation for c_1 and c_2.

PROP$(H_0, 4bitadder_1, 4bitadder_2)$ calculates the output relation H_4, where $4bitadder_1$ and $4bitadder_2$ are the two different descriptions of 4-bit adders.

Algorithm 4.4 shows the pseudo-code for BUILD. It takes three inputs: an instantiation i, a cut-relation H, and a cut K. It is assumed that all input variables for i are in the cut. The lines 1 and 2 calculate the input/output relation for $cell(i)$ using instantiation variables, line 3 calculates the new cut-relation, and line 4 calculates the new cut.

The PROPAGATE algorithm shown in Algorithm 4.5 considers two cell instantiations at a time; one in each circuit. PROPAGATE takes five arguments; two instantiations i_1 and i_2, two cuts K_1 and K_2, and a cut-relation H over the cuts. The result is a new cut-relation and two new cuts. It is

Name: BUILD(i, H, K)
Require: $in(i) \subseteq K$
1: $Map \leftarrow$ map $In(cell(i))$ to $in(i)$ and $Out(cell(i))$ to $out(i)$
2: $R \leftarrow Rel(cell(i))[Map]$
3: $H' \leftarrow \exists v_{\in in(i)} : H \wedge R$
4: $K' \leftarrow K \cup out(i) \backslash in(i)$
5: **return** (H', K')

Algorithm 4.4: The BUILD(i, H, K) algorithm where i is an instantiation, H is a cut-relation, and K is a cut. The output of the algorithm is a pair: a new cut-relation and a new cut. The algorithm moves the cut K past instantiation i and updates the cut-relation H accordingly.

Name: PROPAGATE(i_1, i_2, H, K_1, K_2)
Require: $in(i_1) \subseteq K_1$ and $in(i_2) \subseteq K_2$
1: $R_{in} \leftarrow$ input relation between i_1 and i_2 based on H using (4.2)
2: **if** memorized $(R_{in}, cell(i_1), cell(i_2))$ **then**
3: $\quad R_{out} \leftarrow$ memorized result
4: **else**
5: $\quad$ **either** $R_{out} \leftarrow$ PROP$(R_{in}, cell(i_1), cell(i_2))$
6: $\quad$ **or** $R_{out} \leftarrow \exists v_{\in In(cell(i_1)) \cup In(cell(i_2))} : R_{in} \wedge Rel(cell(i_1)) \wedge Rel(cell(i_2))$
7: $\quad$ memorize $(R_{in}, cell(i_1), cell(i_2), R_{out})$
8: $Map \leftarrow$ map $Out(cell(i_1))$ to $out(i_1)$ and $Out(cell(i_2))$ to $out(i_2)$
9: $H' \leftarrow (\exists v_{\in In(cell(i_1)) \cup In(cell(i_2))} : H) \ \wedge \ R_{out}[Map]$
10: $K_1' \leftarrow K_1 \cup out(i_1) \backslash in(i_1)$
11: $K_2' \leftarrow K_2 \cup out(i_2) \backslash in(i_2)$
12: **return** (H', K_1', K_2')

Algorithm 4.5: The PROPAGATE(i_1, i_2, H, K_1, K_2) algorithm. Both i_1 and i_2 are instances of container cell. H is a cut-relation over the cuts K_1 and K_2. The algorithm moves the cuts past the cell instances and updates the cut-relation accordingly.

assumed that the input variables of the cell instantiations i_1 and i_2 belong to the cuts K_1 and K_2, respectively.

In line 1 PROPAGATE calculates, using (4.2), the input relation R_{in} between i_1 and i_2 based on the cut-relation H. The input relation R_{in} is described in cell variables, not in instantiation variables. Next, we calculate the output relation R_{out} for i_1 and i_2. If we have previously propagated a similar cut past the same cells, we reuse the previous result (line 3). Otherwise we have two ways of calculating R_{out}. If both i_1 and i_2 are instantiations of container cells, we can propagate the cut through these instantiations (line 5) by calling PROP, which allows us to use PROPAGATE on the container cells. Alternatively, we compute R_{out} from the input/output relation Rel for each of the instantiations i_1 and i_2 (line 6). This resembles calling BUILD twice. The rest of the algorithm updates the cuts and calculates the new cut-relation.

In the example, we calculated (4.4) in line 6 and we calculated the updated cut-relation H_1 (4.5) in line 10.

4.3.3 Build vs. Propagate

We use BEDs to represent the characteristic functions of relations. We use UP_ALL to convert BEDs to BDDs. The canonicity of BDDs allows us to recognize memorized results (line 2 in algorithm 4.5) in constant time.

BUILD works by constructing a representation of the input/output relation for a cell which is used to update the cut-relation H. Such a relation captures the functionality of the cell. Using BUILD on the top cell corresponds to the standard verification method of building the BDD for the entire circuit. While this works well for smaller circuits, the BDDs tend to become quite large for more complex circuits.

PROPAGATE works by moving a relation between input variables of two cells to a relation between output variables of the same two cells. In case of container cells, PROPAGATE moves the cuts one step at a time past instantiations of cells in the container cells. It avoids constructing a BDD for the functionality of a cell as long as possible. For logic cells it is necessary to construct such a BDD. However, this BDD represents only the functionality of a part of the circuit, not the whole circuit, and it is therefore more manageable.

The use of PROPAGATE may cause loss of information since it requires construction of the input relation R_{in} between the cell inputs. Consider the two equivalent circuits in Figure 4.6. The input cut for the top circuit is $K_1 = \{s_0, s_1\}$ and for the bottom circuit it is $K_2 = \{t_0, t_1\}$. Let H_0 be the

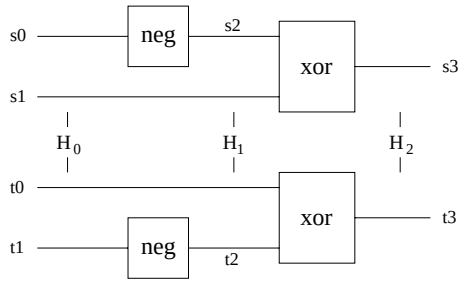

Figure 4.6: Two combinational circuits. The relation H_0 between the wires in the input cuts is $(s_0 \leftrightarrow t_0) \wedge (s_1 \leftrightarrow t_1)$.

cut-relation $(s_0 \leftrightarrow t_0) \wedge (s_1 \leftrightarrow t_1)$ when calling PROPAGATE. We want to move the cuts past the negation cells. The new cuts contain the variables s_1 and s_2, and t_0 and t_2. We build the input relation R_{in} for the two negation cells:

$$\begin{aligned} R_{in} &= (\exists v_{\in (K_1 \cup K_2) \setminus (in(i_1) \cup in(i_2))} : H_0)[Map] \\ &= (\exists v_{\in \{s_1, t_0\}} : (s_0 \leftrightarrow t_0) \wedge (s_1 \leftrightarrow t_1))[Map] \\ &= 1\,, \end{aligned}$$

where i_1 and i_2 are instantiations of the two negation cells. In this case R_{in} evaluates to 1, meaning that the inputs are unrelated; knowing the value of s_0 does not imply a particular value of t_1, i.e., they are unrelated. $R_{in} = 1$ results in H_1 also being 1 and not the expected $(s_1 \leftrightarrow \neg t_2) \wedge (\neg s_2 \leftrightarrow t_0)$.

The problem is that H_0 does not relate the cut variables s_0 and t_1. In general we can state that PROPAGATE(i_1, i_2, H, K_1, K_2) works without loss of information if the cut-relation H can be split in two parts: one part containing the variables in $M = in(i_1) \cup in(i_2)$ and one part containing the remaining variables:

$$H \iff (\exists v_{\in M} : H) \wedge (\exists v_{\in (K_1 \cup K_2) \setminus M} : H) \tag{4.6}$$

If H can be written as in (4.6), PROPAGATE determines the exact cut-relation H'. Otherwise, it gives a conservative approximation to the output relation: $H_{exact} \subseteq H_{approx}$.

4.4 Experimental Results

We have built hierarchical adder and multiplier circuits of different sizes. Each n-bit adder consists of two $n/2$-bit adders. We built one series of adders

Adder [bits]	Runtime [sec]	Multiplier [bits]	Runtime [sec]
64	0.2	32	2.9
128	0.3	64	14.6
256	0.5	128	87.4
512	0.8	256	649
1024	1.6		

Table 4.1: Runtimes in seconds on a 500 MHz Digital Alpha to verify pairs of hierarchical adders (left) and pairs of hierarchical multipliers (right) against each other.

using the full-adder cells from Figure 4.1, and another series of adders using two different types of full-adder cells: one full-adder outputting a negated carry-out, and one receiving a negated carry-in signal. The verification task is to verify that given identical inputs, the adders from the two series have identical outputs. The left part of Table 4.1 shows the runtimes for this experiment. The strategy for moving cuts was to use PROPAGATE whenever H can be written as in (4.6), otherwise we use BUILD. Because of the reuse of previously calculated results, we only apply PROPAGATE a number of times proportional to $\log_2(n)$ for n-bit adders.

Using the standard technique of flattening the two circuits and constructing a BDD for each of them, it is possible to get results comparable to those in Table 4.1 for the verification of adders since the addition function has a small BDD representation (when using an appropriate variable order). However, BDDs are very sensitive to the chosen variable ordering, and using a bad variable order results in BDDs of size exponential in n making it infeasible to build the BDDs for the adders. Our proposed method is not sensitive to the variable ordering of the adders as we never build BDDs representing the functionality of the circuits.

We tested the sensitivity to errors of the cut-propagation method by introducing errors into the adders by switching wires around close to the leaves and close to the root in the hierarchy — errors typically arising if wrong parameter lists are given in the circuit descriptions. None of the modifications cause the runtimes to increase significantly.

While adders are easy to handle using BDDs, multipliers are notoriously difficult. We construct multipliers as series of adders and shifters. From the two different types of adders in the previous experiment, we create two different types of multipliers. The verification task is to verify the pairwise

equivalence of the outputs given the pairwise equivalence of the inputs. One complication is that the outputs of a multiplier are not unrelated. For example, it is not possible for all outputs to be 1 simultaneously[2]. When calculating the cut-relations, such restrictions are included in the relations. This means that the cut-relations contain more information than we need. Repeated use of PROPAGATE, even when the cut-relation cannot be written as (4.6) and thus PROPAGATE causes loss of information, turns out to be exactly what is needed to "forget" this extra information. The right part of Table 4.1 shows the results from running the multiplier experiments.

4.5 Related Work

A traditional way of verifying two hierarchical combinational circuits is to first expand them to flat circuits and then use standard equivalence checking methods for flat circuits. In Chapter 3 we have studied the combinational logic-level verification problem for flat circuits using Boolean Expression Diagrams. This approach works well if the two circuits are similar in structure. However, if the two circuits are very dissimilar in structure, the BED method has the same performance as a standard BDD method, where we build the BDDs for each pair of outputs. Other approaches for flat equivalence checking are discusses in Section 3.6.

Cerny *et. al.* [CM90] split circuits into cells and each cell is described by a relation between the inputs and the outputs of the cell. Using a sweep strategy, they move either forward or backward through the circuits calculating the relations between the circuits along a cut. The cells are lower level logic primitives and thus Cerny *et. al.* have a modular model, but not a hierarchical one.

4.6 Conclusion

We have presented a method based on cut-propagation for obtaining a relation between the outputs of two hierarchically specified combinational circuits. The key new feature of the method is its ability to exploit the hierarchy in the circuit description to reuse previously calculated results in the verification. We have demonstrated the power of the method by verifying large adders and multipliers.

[2]For an n-bit unsigned multiplier, the greatest result is $(2^n - 1)^2$, which is less than $2^{2n} - 1$, where $2^{2n} - 1$ corresponds to 1 on all outputs.

The performance of the method depends on the order in which we pick the subcells when propagating cuts from the inputs to the outputs of container cells. Some orders may result in cut-relations which have large BDD representations. It helps if the hierarchical structure of the two circuits are similar. That way it is easier to pick good candidate subcells for propagating the cuts.

Often the cut-relation states that the variables along the two cuts are pairwise equivalent: $\bigwedge_i s_i \leftrightarrow t_i$. Using BDDs, such a cut-relation can be represented very efficiently. An interleaved variable ordering with s_i and t_i next to each other gives a very compact BDD representation. However, if s_i and t_i are far from one another, then the BDD representation becomes huge. This is a drawback of the presented method.

Chapter 5

Symbolic Model Checking

In this chapter we show how Boolean Expression Diagrams can be used in symbolic model checking. We present a method based on standard fixed-point algorithms, and we use both BDDs and SAT-solvers to perform satisfiability checking. As a result we are able to model check systems for which standard BDD-based methods fail. This chapter is partly based on the paper [WBCG00].

5.1 Introduction

Symbolic model checking has been performed using fixed-point iterations for a number of years [BCM+92, McM93]. The key to the success is the canonical Binary Decision Diagram (BDD) [Bry86] data structure for representing Boolean functions. However, such a representation explodes in size for certain functions. Biere *et. al.* [BCC+99, BCCZ99, BCRZ99] introduced Bounded Model Checking as a way of avoiding BDDs. Instead of performing a fixed-point iteration, they construct formulae for possible counterexamples and use SAT-solvers to prove or disprove the existence of such counterexamples. Abdulla *et. al.* [ABE00] also use SAT-solvers but keep the fixed-point iterations.

In this chapter we combine BDDs and SAT-solvers in symbolic model checking based on fixed-points. We use Boolean Expression Diagrams as the underlying data structure. The method is theoretically complete as we only change the representation and not the algorithms. Going from a BDD to a BED representation, we have to give up canonicity. That has both advantages and disadvantages: Non-canonical data structures are more succinct than canonical ones – sometimes exponentially more succinct.

Determining satisfiability of Boolean functions is easy with canonical data structures, but with non-canonical data structures it is hard. We show how to overcome the disadvantages and exploit some of the advantages in symbolic model checking.

We use two different methods for satisfiability checking: (1) SAT-solvers like GRASP[1] [MSS99] and SATO[2] [Zha97], and (2) conversion of BEDs to BDDs. BDDs are canonical and thus we can check for satisfiability in constant time. We perform symbolic model checking the classical way with fixed-point iterations. One of the key elements of our method is the *quantification by substitution* rule: $\big(\exists y : g \wedge (y \leftrightarrow f)\big) \leftrightarrow g[f/y]$. The rule is used (1) during fixed-point iterations, (2) while deciding whether an initial set of states is a subset of another set of states, and finally (3) while doing iterative squaring.

While complete in the sense that it handles full CTL model checking, our method performs best if the system has few inputs. The reason is that this allows us to fully exploit the quantification by substitution rule.

Using our method, we can model check a liveness property of a 256-bit shift-and-add multiplier, which requires 256 iterations to reach the fixed-point. This should be compared with the 23-bit multipliers that standard BDD methods can handle. In fact, we are able to detect a previously unknown bug in the specification of a 16-bit multiplier. It was generally thought that iterative squaring was of no use in model checking. However, we show that iterative squaring enables us to calculate the reachable set of states for *all* 32 outputs of a 16-bit multiplier faster than without iterative squaring.

The quantification by substitution rule helps remove some of the quantifications. However, the remaining quantifications still cause trouble in the form of a size explosion when we eliminate them. In the last part of the chapter we investigate the possibility of skipping the quantification of the remaining variables by simply leaving the variables in the formula.

We shall only consider model checking of *finite* state system. *Infinite* state systems are beyond the scope of this dissertation.

[1] GRASP version September 1999. See `http://algos.inesc.pt/~jpms/grasp` for more information.

[2] SATO version 3.2. See `http://www.cs.uiowa.edu/~hzhang/sato.html` for more information.

5.2 Theory of Model Checking

In this section we describe model checking. We first use set theory as the underlying theory. Then we redefine model checking in terms of Boolean functions[3]. The systems we consider are finite state machines (FSMs) represented by finite Kripke structures [HC74, CGP99].

Definition 5.2.1 (Finite Kripke Structure). A *finite Kripke structure* M is a tuple (S^M, I^M, T^M, ℓ^M) with a finite set of states S^M, a set of initial states $I^M \subseteq S^M$, a transition relation $T^M \subseteq S^M \times S^M$, and a labeling of the states $\ell^M : S^M \mapsto \mathcal{P}(\mathcal{A})$ with atomic propositions $\mathcal{A}$.

Consider the SMV [McM93] program and the accompanying Kripke structure in Figure 5.1. There are four states in the Kripke structure represented by two state variables s_1 and x_1. Only the s_1 variable is restricted through the `init` and `next` statements. The x_1 variable is unrestricted. We can interpret the system as follows: x_1 is a variable which signals the presents (x_1 is true) or absence (x_1 is false) of an input which toggles s_1.

```
MODULE main
VAR
  s1 : boolean;
  x1 : boolean;

ASSIGN
  init(s1) := 1;
  next(s1) := !(s1 <-> x1);
```

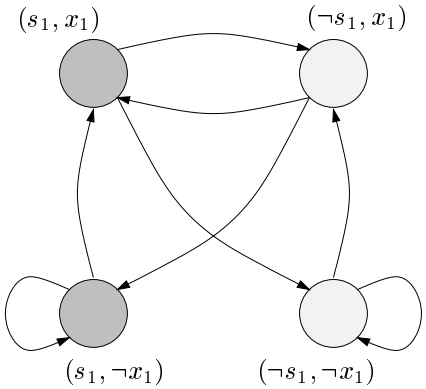

Figure 5.1: An SMV program and the associated Kripke structure. The initial states are colored dark-gray.

This example seems to indicate that there are two kinds of state variables: Those that are restricted and those that are unrestricted. The restricted variables carry the information on which state we are in. The unrestricted variables carry the information of the inputs to the system. We use a modified definition of Kripke structures that allows us to distinguish between the two kinds of variables.

[3]This causes some overloading of symbols. We have chosen overloading instead of using a more cumbersome notation.

Definition 5.2.2 (Modified Kripke Structure). A modified *Kripke structure* M is a tuple (S, X, I, T, ℓ) with a finite set of states S, a finite set of inputs X, a set of initial states $I \subseteq S$, a transition function $T : S \times X \mapsto S$, and a labeling of the states $\ell : S \mapsto \mathcal{P}(\mathcal{A})$ with atomic propositions $\mathcal{A}$.

The transition relation now becomes a transition function which we assume is defined for all $(s, x) \in S \times X$. This has the effect that a modified Kripke structure with an empty set of inputs X is deterministic. We call a modified Kripke structure with a non-empty set of inputs for a reactive system. By having a transition function and not a transition relation, we isolate the non-determinism to inputs. One can always add more inputs to obtain the required non-determinism.

Figure 5.2 shows the modified Kripke structure for the same SMV program as in Figure 5.1.

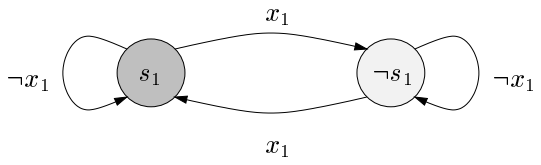

Figure 5.2: A modified Kripke structure. The initial state is dark-gray.

Given a modified Kripke structure $M = (S, X, I, T, \ell)$, we can obtain an equivalent Kripke structure $M' = (S^M, I^M, T^M, \ell^M)$ in the following way:

$$
\begin{aligned}
S^M &= S \times X \\
I^M &= I \times X \\
T^M &= \{\ \big((s^1, x^1), (s^2, x^2)\big) \mid \\
&\qquad s^1, s^2 \in S \text{ and } x^1, x^2 \in X \text{ and } s^2 = T(s^1, x^1)\ \} \\
\ell^M(s, x) &= \ell(s) \quad \text{where } (s, x) \in S^M
\end{aligned}
$$

In the following, we use the term "Kripke structure" to refer to the modified version.

The transition function T specifies all the possible behaviors of a system. An FSM can only move from state s^i to state s^j if there is an input x such that $s^j = T(s^i, x)$. We say that the FSM takes a transition from s^i to s^j on input x.

Definition 5.2.3 (Transition). Let $M = (S, X, I, T, \ell)$ be a Kripke structure. There is a *transition* from state $s^i \in S$ to state $s^j \in S$ if:

$$\exists x \in X : s^j = T(s^i, x)\,.$$

We write $s^i \overset{x}{\rightsquigarrow} s^j$ to indicate a transition from s^i to s^j on input x. If the input does not matter or is implicitly understood, then we just write $s^i \rightsquigarrow s^j$.

We say there is a path from state s^i to state s^j if there is a series of successive transitions leading from s^i to s^j.

Definition 5.2.4 (Path). Let $M = (S, X, I, T, \ell)$ be a Kripke structure. A *path* from a state s is an infinite sequence $\langle s^1, s^2, \ldots \rangle$ of states in S such that:

$$s = s^1\,, \text{ and}$$
$$\forall i \geq 1 : \exists x \in X : s^{i+1} = T(s^i, x)\,.$$

We write $s^i \rightsquigarrow_k s^j$ to indicate that there is a path of length k between states s^i and s^j, i.e., there is an infinite path $\langle s^1, s^2, \ldots \rangle$ such that $s^i = s^1$ and $s^j = s^{k+1}$. The length indicates the number of transitions required of the FSM to move from state s^i to state s^j.

Since the transition function is defined for all $(s, x) \in S \times X$, it means that it is always possible to take a transition. Self-loops, i.e., transitions leading from one state to itself, are allowed. There are no dead-end states and thus all finite paths can be thought of as prefixes of infinite paths.

An infinite path is sometimes called a *computation*.

5.2.1 Computation Tree Logic

Computation Tree Logic (CTL) [CES86] is a temporal logic used to describe the specification of a finite state machine. There are a number of different temporal logics, but we shall only consider CTL.

Given a state in a Kripke structure M and a CTL specification ϕ for M, then ϕ either holds or does not hold for that state. Thus a CTL formula represents a set of states, namely the states for which the CTL formula holds. We denote that set of states $[\![\phi]\!]$.

Model checking is the process of determining whether a Kripke structure $M = (S, X, I, T, \ell)$ is a model of a CTL formula ϕ. We write $M \models \phi$ to indicate that M models ϕ. In the following, when we write CTL formulae, we also assume a given system M.

Definition 5.2.5 (Model Checking). Let $M = (S, X, I, T, \ell)$ be a Kripke structure and let ϕ be a CTL specification for M. We say that M models ϕ, and write $M \models \phi$, if $I \subseteq [\![\phi]\!]$.

A CTL formula contains a propositional logic part with constants, negation, conjunction and disjunction. Furthermore, it contains a number of temporal operators each consisting of a path quantifier and a path operator. There are two path quantifiers: **E** ("there exists an infinite path") and **A** ("for all infinite paths"). There are five path operators:

X	:	next state
G	:	globally
F	:	future
U	:	until
R	:	release

The intuition behind the path operators is the following (here expressed with the **E** path quantifier and all paths mentioned originate from the current state):

EX ϕ : ϕ holds in the next state along some path.

EG ϕ : ϕ holds on all states along some path.

EF ϕ : ϕ holds at some point in the future along some path.

EU(ϕ_1, ϕ_2) : On some path, ϕ_1 holds on all states until ϕ_2 holds[4].

ER(ϕ_1, ϕ_2) : On some path, ϕ_2 holds until ϕ_1 releases ϕ_2 (i.e., ϕ_2 holds on the first state in which ϕ_1 holds).

Definition 5.2.6 (CTL Syntax). A CTL formula can be generated from the following grammar:

$$f \quad ::= \quad 0 \mid 1 \mid a \mid \neg f \mid f \vee f \mid f \wedge f \mid \mathbf{EX}\, f \mid \mathbf{EG}\, f \mid \mathbf{EU}(f, f)\,,$$

where a is an atomic proposition.

[4]We use the notion of *strong until* where ϕ_2 has to hold eventually. There is also a *weak until* in which ϕ_2 does not have to hold if ϕ_1 holds on all states along the infinite path.

The remaining temporal operators are defined in terms of the three basic operators **EX**, **EG**, and **EU**:

$$\begin{aligned}
\mathbf{AX}\ \phi &\equiv \neg\mathbf{EX}\ \neg\phi \\
\mathbf{EF}\ \phi &\equiv \mathbf{EU}(1, \phi) \\
\mathbf{AF}\ \phi &\equiv \neg\mathbf{EG}\ \neg\phi \\
\mathbf{AG}\ \phi &\equiv \neg\mathbf{EF}\ \neg\phi \\
\mathbf{AU}(\phi_1, \phi_2) &\equiv \neg\mathbf{EU}(\neg\phi_2, \neg\phi_1 \wedge \neg\phi_2) \wedge \neg\mathbf{EG}\ \neg\phi_2 \\
\mathbf{ER}(\phi_1, \phi_2) &\equiv \neg\mathbf{AU}(\neg\phi_1, \neg\phi_2) \\
\mathbf{AR}(\phi_1, \phi_2) &\equiv \neg\mathbf{EU}(\neg\phi_1, \neg\phi_2)
\end{aligned}$$

Before giving the semantics of a CTL formula, we define two auxiliary operators: the least and greatest fixed-point operators. Both operators take a monotonic set transformer as argument and return the fixed-point (least or greatest) for that set transformer.

Definition 5.2.7 (Set Transformer). A set transformer τ is a function $\tau : \Phi \mapsto \Phi$, where Φ is some set.

τ is *monotonic* if $P_1 \subseteq P_2$ implies $\tau(P_1) \subseteq \tau(P_2)$.

In model checking of a Kripke structure $M = (S, X, I, T, \ell)$, we use set transformers of the type $\tau : \mathcal{P}(S) \mapsto \mathcal{P}(S)$. Algorithm 5.3 shows the pseudo-code for the least fixed-point operator LFP and the greatest fixed-point operator GFP, respectively.

Name: LFP $[\tau]$
Require: τ must be monotonic.
1: $Q \leftarrow \emptyset$
2: $Q' \leftarrow \tau(Q)$
3: **while** $Q \neq Q'$ **do**
4: $\quad Q \leftarrow Q'$
5: $\quad Q' \leftarrow \tau(Q')$
6: **return** Q

Name: GFP $[\tau]$
Require: τ must be monotonic.
1: $Q \leftarrow S$
2: $Q' \leftarrow \tau(Q)$
3: **while** $Q \neq Q'$ **do**
4: $\quad Q \leftarrow Q'$
5: $\quad Q' \leftarrow \tau(Q')$
6: **return** Q

Algorithm 5.3: The LFP operator (left) and the GFP operator (right). They take a set transformer as argument and returns the least fixed-point and greatest fixed-point, respectively, for it.

Lemma 5.2.8 (Increasing). *The sequence of Q's $(Q_0, Q_1, Q_2, \ldots)$ obtained from the* LFP *algorithm is increasing. That is, $Q_i \subseteq Q_{i+1}$ for all $i \geq 0$.*

Lemma 5.2.9 (Decreasing). *The sequence of Q's $(Q_0, Q_1, Q_2, \ldots)$ obtained from the* GFP *algorithm is decreasing. That is, $Q_{i+1} \subseteq Q_i$ for all $i \geq 0$.*

Lemmas 5.2.8 and 5.2.9 follow directly from the definition of a monotonic set transformer in Definition 5.2.7.

The semantics $[\![\phi]\!]$ of a CTL formula ϕ is a set of states. For example, the CTL formula $\mathbf{EG}\ \phi$ denotes the set of states $[\![\mathbf{EG}\ \phi]\!]$ such that from each state in $[\![\mathbf{EG}\ \phi]\!]$ there exists an infinite path on which ϕ holds globally. Four of the most common CTL operators are **EF**, **AF**, **EG**, and **AG**. One may think of them as follows:

$$\begin{aligned} \mathbf{EF}\ \phi &: \phi \text{ is potential} \\ \mathbf{AF}\ \phi &: \phi \text{ is inevitable} \\ \mathbf{EG}\ \phi &: \phi \text{ is potentially invariant} \\ \mathbf{AG}\ \phi &: \phi \text{ is invariant} \end{aligned}$$

Definition 5.2.10 (CTL Semantics). Given a Kripke structure $M = (S, X, I, T, \ell)$, the semantics of a CTL formula is a set of states $[\![\cdot]\!] \subseteq \mathcal{P}(S)$.

$$\begin{aligned} [\![0]\!] &= \emptyset \\ [\![1]\!] &= S \\ [\![a]\!] &= \{\ s \in S \mid a \in \ell(s)\ \} \\ [\![\neg\phi]\!] &= S \setminus [\![\phi]\!] \\ [\![\phi_1 \vee \phi_2]\!] &= [\![\phi_1]\!] \cup [\![\phi_2]\!] \\ [\![\phi_1 \wedge \phi_2]\!] &= [\![\phi_1]\!] \cap [\![\phi_2]\!] \\ [\![\mathbf{EX}\ \phi]\!] &= \{\ s \in S \mid \exists s' \in [\![\phi]\!]\ :\ s \rightsquigarrow s'\ \} \\ [\![\mathbf{EG}\ \phi]\!] &= \textsc{Gfp}\ Z\ \Big[\ [\![\phi]\!] \cap \mathbf{EX}\ Z\ \Big] \\ [\![\mathbf{EU}(\phi_1, \phi_2)]\!] &= \textsc{Lfp}\ Z\ \Big[\ [\![\phi_2]\!] \cup \big([\![\phi_1]\!] \cap \mathbf{EX}\ Z\big)\ \Big] \end{aligned}$$

where a is an atomic proposition.

The set transformer arguments to LFP and GFP in Definition 5.2.10 are monotonic. Lemma 5.2.11 proves it for $[\![\phi]\!] \cap \mathbf{EX}\ Z$.

Lemma 5.2.11. *The set transformer $\tau(Z) = [\![\phi]\!] \cap \mathbf{EX}\ Z$ is monotonic.*

Proof. Let $P_1 \subseteq P_2$. The set transformer $\tau(Z)$ is monotonic if $\tau(P_1) \subseteq \tau(P_2)$. Let s be some state in $\tau(P_1)$. Then s is in $[\![\phi]\!]$, and there exists a state $s' \in P_1$ such that $s \rightsquigarrow s'$. Since $P_1 \subseteq P_2$, then $s' \in P_2$, and thus $s \in \tau(P_2)$. □

The following is a list of common CTL formulae:

EF ϕ : It is possible to reach a state where ϕ holds.

AG$(Req \rightarrow$ **AF** $Ack)$: All requests will eventually be acknowledged.

AG AF ϕ : On every path ϕ holds infinitely often.

AG EF ϕ : It is always possible to reach a state where ϕ holds.

5.2.2 Model Checking using Propositional Logic

The theory of model checking is based on set theory. McMillan [McM93] showed that using characteristic functions of sets instead of the sets themselves, one is able to get highly efficient model checking algorithms. The main idea is that characteristic functions allow for a symbolic handling of sets. Previously an explicit enumeration of the sets was required, and that was often impractical due to the size of the sets.

Consider a Kripke structure $M = (S, X, I, T, \ell)$. We redefine it in terms of characteristic functions instead of sets.

S : We encode a state as a vector of Boolean variables $s = (s_1, \ldots, s_n)$. We refer to these variables as state variables.

X : We encode an input as a vector of Boolean variables $x = (x_1, \ldots, x_m)$. We refer to these variables as input variables.

I : We use the characteristic function $\Xi_I(s)$.

T : T is already a function.

ℓ : As atomic propositions we use state variables. Each state is labeled with the state variables which occur as positive for that state.

$$s_i \in \ell(s) \quad \text{if the } i\text{'th bit of } s \text{ is } 1\,.$$

Note that in the Kripke structures in Figures 5.1 and 5.2 we have already used Boolean variables to encode the states.

For both state and input variables we use subscripts to indicate elements of the vector and superscripts to indicate different vectors. For state variables, we use the convention that unprimed variables encode the current state while primed variables encode the next state. For example, $s' = T(s, x)$ indicates a transition on input x from current state s to next state s'.

In the *set* version of model checking, each state was labeled with a set of atomic propositions. The atomic propositions were not specified. Now we assume that the atomic propositions are state variables. This means we can change the CTL grammar from Definition 5.2.6 to reflect this. Definition 5.2.12 shows the modified CTL grammar.

Definition 5.2.12 (Modified CTL Syntax). A CTL formula can be generated from the following grammar:

$$f \quad ::= \quad 0 \mid 1 \mid s_i \mid \neg f \mid f \vee f \mid f \wedge f \mid \mathbf{EX}\ f \mid \mathbf{EG}\ f \mid \mathbf{EU}(f, f)\,,$$

where s_i is any state variable.

Each set operator has a corresponding logic operator. Using the set operator on sets corresponds to using the logic operator on the characteristic functions for the sets. Table 5.1 shows a list of set notations and their corresponding logic notations.

Set notation	Logic notation
$\emptyset$	0
S	1
α, $\{\alpha\}$	α
$\Phi \cup \Theta$	$\Phi \vee \Theta$
$\Phi \cap \Theta$	$\Phi \wedge \Theta$
$\Phi \setminus \Theta$	$\Phi \wedge \neg\Theta$
$S \setminus \Phi$	$\neg\Phi$
$\Phi = \Theta$	$\mathrm{TAUT}(\Phi \leftrightarrow \Theta)$
$\Phi \neq \Theta$	$\mathrm{SAT}(\Phi \oplus \Theta)$
$\alpha \in \Phi$	$\mathrm{TAUT}(\alpha \rightarrow \Phi)$
$\alpha \notin \Phi$	$\mathrm{SAT}(\alpha \wedge \neg\Phi)$
$\Phi \subseteq \Theta$	$\mathrm{TAUT}(\Phi \rightarrow \Theta)$
$\Phi \not\subseteq \Theta$	$\mathrm{SAT}(\Phi \wedge \neg\Theta)$

Table 5.1: Correspondence between set and logic notations. On the set side, α is an element and Φ and Θ are sets. On the logic side, α, Φ and Θ are functions. These correspondences assume that $|S|$ is equal to 2^k for some integer k. Otherwise we have to consider "ghost-states" introduced by the encoding of states and inputs as vectors of Boolean variables.

Using the transformations in Table 5.1, it is straightforward to adapt LFP and GFP to characteristic functions: $\emptyset$ and S become 0 and 1, respectively,

$Q \neq Q'$ becomes $\mathrm{SAT}(Q \oplus Q')$, and the set transformer τ should work on characteristic functions for sets instead of the sets themselves.

We adapt the other CTL definitions to characteristic functions as well.

Definition 5.2.13 (Modified CTL Semantics). Given a Kripke structure $M = (S, X, I, T, \ell)$, the semantics of a CTL formula is a set of states $[\![\cdot]\!] \subseteq \mathcal{P}(S)$ as described in Definition 5.2.10. In terms of characteristic functions, $[\![\cdot]\!]$ is a Boolean function of the state vector s defined recursively as follows:

$$\begin{aligned}
[\![0]\!] &= 0 \\
[\![1]\!] &= 1 \\
[\![s_i]\!] &= s_i \\
[\![\neg\phi]\!] &= \neg[\![\phi]\!] \\
[\![\phi_1 \vee \phi_2]\!] &= [\![\phi_1]\!] \vee [\![\phi_2]\!] \\
[\![\phi_1 \wedge \phi_2]\!] &= [\![\phi_1]\!] \wedge [\![\phi_2]\!] \\
[\![\mathbf{EX}\ \phi]\!] &= \exists s', x : \big(s' \leftrightarrow T(s,x)\big) \wedge [\![\phi]\!][s'/s] \\
[\![\mathbf{EG}\ \phi]\!] &= \textsc{Gfp}\ Z \left[[\![\phi]\!] \wedge \mathbf{EX}\ Z \right] \\
[\![\mathbf{EU}(\phi_1, \phi_2)]\!] &= \textsc{Lfp}\ Z \left[[\![\phi_2]\!] \vee \big([\![\phi_1]\!] \wedge \mathbf{EX}\ Z\big) \right]
\end{aligned}$$

where $[\![\phi]\!][s'/s]$ is a substitution of s' for s in $[\![\phi]\!]$.

Definition 5.2.14 (Modified Model Checking). Let $M = (S, X, I, T, \ell)$ be a Kripke structure and let ϕ be a CTL specification for M. We say that M models ϕ, and write $M \models \phi$, if $\mathrm{TAUT}(I \rightarrow [\![\phi]\!])$.

5.3 Model Checking with BEDs

In this section we apply BEDs to model checking. We use BEDs to represent the characteristic functions for sets. In the following we assume that all sets are represented by characteristic functions.

Let us look at how we actually solve a model checking problem. Assume we have a Kripke structure $M = (S, X, I, T, \ell)$ and a CTL specification ϕ. The following steps are necessary to determine whether $M \models \phi$:

1. Construct BEDs for the transition function T and the set of initial states I.

2. Compute a BED for $[\![\phi]\!]$ using Definition 5.2.13.

3. Compute TAUT$(I \rightarrow [\![\phi]\!])$. The specification holds if the result is true.

We now examine each step to see how we can apply BEDs. The transition function T is a function $S \times X \mapsto S$. We use T_i to indicate the function for the i'th state variable:

$$s' \leftrightarrow T(s,x) \;=\; \bigwedge_{i=1}^{n} s'_i \leftrightarrow T_i(s,x)\,, \tag{5.1}$$

where n is the number of state variables. Likewise, we assume that the set of initial states I is on the form

$$I \;=\; \bigwedge_{i} s_i \leftrightarrow I_i(s_1,\ldots,s_{i-1},s_{i+1},\ldots,s_n)\,. \tag{5.2}$$

Not all state variables need to be in the conjunction for I. If a state variable is omitted it means that there are initially no restrictions on the state variable. The SMV language [McM93] for describing finite state machines lets the user specify the transition relation and the set of initial states in this functional form. More specifically, the `next` and `init` statements in the `ASSIGN` section in SMV programs correspond to the T_i and I_i of Equations 5.1 and 5.2[5].

Instead of building one BED for T and one BED for I, we construct a series of BEDs; one BED for each T_i and I_i. In Section 5.3.1 we show why this is advantageous.

The second step was to compute a BED for $[\![\phi]\!]$ using Definition 5.2.13. The first six lines of the definition are straightforward: The BED is constructed recursively by adding either a terminal, a variable, a negation, a disjunction or a conjunction vertex. The three temporal operators are more interesting.

The substitution $[s'/s]$ in $[\![\mathbf{EX}\ \phi]\!]$ is actually a variable renaming. It can be done in one traversal of the BED. The quantification needs to be handled with care. We deal with it in Section 5.3.1.

In $[\![\mathbf{EG}\ \phi]\!]$ and $[\![\mathbf{EU}(\phi_1,\phi_2)]\!]$ we need the two fixed-point operators. In line 3 of LFP (Algorithm 5.3), we compare Q and Q' to find out whether we

[5]Some versions of SMV allow the use of primed variables in the `next` statements. This would correspond to $T_i(s,x,s')$ instead of just $T_i(s,x)$ in Equation 5.1. We do not consider such cases.

have reached the fixed-point. Since the sequence of Q's in LFP is increasing (see Lemma 5.2.8), it always holds that $Q \subseteq Q'$. Thus we can change the **while** condition to $Q' \not\subseteq Q$. Similarity for GFP (Algorithm 5.3). Since the sequence of Q's in GFP is decreasing (see Lemma 5.2.9), we can change the **while** condition to $Q \not\subseteq Q'$. In terms of characteristic functions, these conditions become $\mathrm{SAT}(Q' \wedge \neg Q)$ and $\mathrm{SAT}(Q \wedge \neg Q')$, respectively. Section 5.3.2 deals with satisfiability checking.

5.3.1 Quantification

The basic step in our quantification algorithm is to eliminate *one* quantified variable by the following rules:

$$\exists y : f \;\equiv\; f[0/y] \vee f[1/y] \qquad\qquad \forall y : f \;\equiv\; f[0/y] \wedge f[1/y] \tag{5.3}$$

It is worth noting, that these basic steps can easily be computed by performing an UP_ONE(y, f) operation and then replacing the top level variable vertex by an appropriate operator vertex.

In the worst case, while removing a quantifier from a formula, we double the formula size. Since each **EX** computation involves existential quantification of n state variables and m input variables, we risk increasing the formula size by a factor of 2^{n+m}. We have tried this kind of quantification on some examples. The result is always the same: Quantification of the first handful or two of variables only increase the size of the BED slightly. However, each of the remaining quantifications nearly doubles the BED size. It is our experience that for any reasonably sized problem, this quantification method is not sufficient.

In some cases it is possible to replace an existential quantification by a substitution. We call this the *quantification by substitution* rule, and it is a cornerstone in our BED model checking method:

$$\exists y : g \wedge (y \leftrightarrow f) \quad\equiv\quad g[f/y]\,, \qquad \text{where } y \text{ does not occur in } f\,. \tag{5.4}$$

Abdulla *et. al.* [ABE00] also use quantification by substitution in their model checking algorithm. They call it *inlining*. One can think of quantification by substitution as a special case of *pruning* which we described in Section 3.2.3.

We use the quantification by substitution rule in three places: **EX** computation, set inclusion, and iterative squaring.

EX Computation

Consider the **EX** computation in Definition 5.2.13. If the transition function T is written as in Equation 5.1, then we can apply rule (5.4) directly for the

quantification of the state variables. This can be done in one traversal of the BED. Algorithm 5.4 shows the pseudo-code for the QBS algorithm for computing $\exists s' : \big(s' \leftrightarrow T(s,x)\big) \wedge [\![\phi]\!][s'/s]$:

$$\begin{aligned} & \exists s' : (s' \leftrightarrow T(s,x)) \wedge [\![\phi]\!][s'/s] \\ = \quad & [\![\phi]\!][s'/s][T/s'] \\ = \quad & [\![\phi]\!][T/s] \end{aligned}$$

The QBS algorithm assumes that T is given as the set of BEDs for T_i from Equation 5.1 and that $[\![\phi]\!]$ is also given as a BED. It works in a bottom-up way replacing all primed state variables in $[\![\phi]\!][s'/s]$ (corresponding to unprimed state variables in $[\![\phi]\!]$) with their next-state function T_i. Line 5 does the replacing by performing a Shannon expansion of the variable vertex and inserting the next-state function.

Name: QBS(u)
1: **if** u is a terminal vertex **then**
2: **return** u
3: $(l,h) \leftarrow \big(\text{QBS}(low(u)), \text{QBS}(high(u))\big)$
4: **if** u is an unprimed variable vertex with $var(u) = s_j$ **then**
5: **return** $(T_j \wedge h) \vee (\neg T_j \wedge l)$
6: **else**
7: **return** MK($\alpha(u), l, h$)

Algorithm 5.4: The QBS algorithm for quantification by substitution. It computes $\exists s' : \big(s' \leftrightarrow T(s,x)\big) \wedge [\![\phi]\!][s'/s]$, where T is given as BEDs for T_i in Equation 5.1 and $[\![\phi]\!]$ is given as a BED u.

Quantification by substitution works for the quantification of the state variables in **EX**. The reason is our assumption on the form of T, where each next-state variable has a corresponding next-state function. Unfortunately, we cannot use quantification by substitution for the quantification of input variables since their values are not bound to a function as is the case for the next-state variables.

Set Inclusion

We now describe a preprocessing step simplifying TAUT($I \rightarrow [\![\phi]\!]$), i.e., whether the initial set of states is a subset of the states characterized by the specification. We assume we have BEDs for the I_i functions (as per

Equation 5.2) and for $[\![\phi]\!]$. In many cases I_i is either a constant or a very simple function, and we can use this fact to simplify $\mathrm{TAUT}(I \to [\![\phi]\!])$.

Let I be written $I' \wedge (s_i \leftrightarrow I_i)$, where I_i is a function of all state variables $s_1, \ldots, s_n$, but not s_i. Recall that tautology checking corresponds to universal quantification of all variables. This means that $I \to [\![\phi]\!]$ is a tautology if and only if $\forall s_i : I \to [\![\phi]\!]$ is a tautology:

$$\begin{aligned} & \forall s_i : I \to [\![\phi]\!] \\ = \quad & \forall s_i : \neg\left(I' \wedge (s_i \leftrightarrow I_i) \wedge \neg[\![\phi]\!]\right) \\ = \quad & \neg\exists s_i : I' \wedge (s_i \leftrightarrow I_i) \wedge \neg[\![\phi]\!] \\ = \quad & \neg(I' \wedge \neg[\![\phi]\!])[I_i/s_i] \\ = \quad & (I' \to [\![\phi]\!])[I_i/s_i] \end{aligned}$$

The $[I_i/s_i]$ means a substitution of I_i for s_i. In the third step we use quantification by substitution to replace a quantification by a substitution. In many cases the I_i functions are quite simple, e.g., a constant. In such situations this method reduces the number of variables and simplifies the formula.

Consider the case where I is a singleton set, i.e., there is only one initial state. Assume without loss of generality that this initial state is $(0, \ldots, 0)$. Our preprocessing step would simply replace all state variables in $[\![\phi]\!]$ with zeros. There would be no variables left, and thus the whole expression would trivially reduce to either 0 or 1 making the check for tautology trivial.

Iterative Squaring

Iterative squaring is a technique for reducing the number of iterations needed to reach the fixed-point for both the least and greatest fixed-point operators [BCL$^+$94]. During reachability analysis we repeatedly square the transition function:

$$s' \leftrightarrow T^2\big(s, (x, y)\big) = \exists s'' : \big(s'' \leftrightarrow T(s, x)\big) \wedge \big(s' \leftrightarrow T(s'', y)\big)\,.$$

$T^2(s, (x, y))$ is a new transition function allowing a transition from s to a state s' on input (x, y) if there is a middle state s'' such that $s \overset{x}{\rightsquigarrow} s''$ and $s'' \overset{y}{\rightsquigarrow} s'$, where $\rightsquigarrow$ indicates a transition in the old system.

Assume that T is written as in Equation 5.1.

$$\begin{aligned} s' \leftrightarrow T^2\big(s,(x,y)\big) &= \exists s'' : \big(s'' \leftrightarrow T(s,x)\big) \wedge \big(s' \leftrightarrow T(s'',y)\big) \\ &= \exists s'' : \left(\bigwedge_i s_i'' \leftrightarrow T_i(s,x)\right) \wedge \left(\bigwedge_i s_i' \leftrightarrow T_i(s'',y)\right) \\ &= \bigwedge_i s_i' \leftrightarrow \big(T_i(s'',y)[T_j(s,x)/s_j'']_j\big) \ , \end{aligned}$$

where $[T_j(s,x)/s_j'']_j$ is a substitution of function $T_j(s,x)$ for variable s_j'' for all j. The algorithm is similar to QBS in Algorithm 5.4.

In this way we can compute $T^{(2^k)}$ in only k steps. $T^{(2^k)}$ is a new transition function representing all paths in T with a length of *exactly* 2^k, and $T^{(2^k)}$ is on the functional form of Equation 5.1. However, it is not possible to represent on functional form the transition function allowing paths of length *up to* 2^k as this would involve a disjunction of transition functions. As a consequence we cannot combine this form of iterative squaring with, for example, frontier set simplifications [CBM89, BCL+94].

Algorithm 5.5 shows how to compute $[\![\mathbf{EF}\ \phi]\!]$ using a least fixed-point method and iterative squaring. $\mathbf{EX}^{(2^n)}$ is the **EX** operator with $T^{(2^n)}$ as transition function. After each iteration in the **while** loop, Q' represents the set of states reachable in up to and including $2^n - 1$ steps.

Name: EF(ϕ)
1: $n \leftarrow 0$
2: $Q \leftarrow 0$
3: $Q' \leftarrow [\![\phi]\!]$
4: **while** $Q \neq Q'$ **do**
5: $\quad Q \leftarrow Q'$
6: $\quad Q' \leftarrow Q' \vee \mathbf{EX}^{(2^n)}\ Q'$
7: $\quad n \leftarrow n + 1$
8: **return** Q

Algorithm 5.5: The EF algorithm. It computes **EF** ϕ with a least fixed-point iteration and using iterative squaring. $\mathbf{EX}^{(2^n)}$ is the **EX** operator with $T^{(2^n)}$ as transition function.

Scope Reduction Rules

Our verification method performs best when we can exploit the quantification by substitution rule. Such cases include systems with few or no

inputs. After performing quantification by substitution, we quantify the inputs variables using the rules below.

By applying scope reduction rules to a formula, we can push quantifiers down and thus reduce the potential blowup. The scope reduction rules are the following (shown for negation, conjunction and disjunction):

$$\begin{aligned}
\exists x : \neg f &= \neg \forall x : f \\
\exists x : f \vee g &= (\exists x : f) \vee (\exists x : g) \\
\exists x : f(y) \wedge g(x) &= f(y) \wedge (\exists x : g(x)) \\
\forall x : \neg f &= \neg \exists x : f \\
\forall x : f \wedge g &= (\forall x : f) \wedge (\forall x : g) \\
\forall x : f(y) \vee g(x) &= f(y) \vee (\forall x : g(x))
\end{aligned}$$

There are two ways to implement scope reduction rules. One way is to think of the quantifiers as algorithms ("forall" and "exist") on BEDs. Calling one of the algorithms on a vertex either generates new calls for the children according to the scope reduction rules or the algorithm expands the vertex using the basic quantifier rules in Equation 5.3.

The other way of handling the scope reduction rules is to add a new vertex type to the BEDs. It is possible to add a quantifier vertex with three attributes: a quantifier $quant \in \{\exists, \forall\}$, a variable var and BED low. The semantics of a quantifier vertex v is the Boolean function $f^v = quant(v)\ var(v) : f^{low(v)}$. The rewriting rules in Figure 3.2 can be extended with the scope reduction rules. The algorithms Up_One and Up_All need also be modified to handle the new vertex type. This turns out to be relatively simple since the quantifiers distribute over the *if-then-else* operator (remember that BEDs are assumed to be free):

$$\begin{aligned}
\exists x : y \rightarrow f, g &= y \rightarrow \exists x : f, \exists x : g \\
\exists x : x \rightarrow f, g &= f \vee g \\
\forall x : y \rightarrow f, g &= y \rightarrow \forall x : f, \forall x : g \\
\forall x : x \rightarrow f, g &= f \wedge g
\end{aligned}$$

Chapter 7 details how to extend the BED data structure with new types of vertices.

5.3.2 Satisfiability Checking

There are two places where we need to determine whether a Boolean formula represented by a BED is satisfiable. First we need to detect that a fixed-point has been reached in the computation of the set of states satisfying a CTL formula. This corresponds to $\mathrm{SAT}(Q' \wedge \neg Q)$ (or $\mathrm{SAT}(Q \wedge \neg Q')$). The formula $Q' \wedge \neg Q$ (or $Q \wedge \neg Q'$) is satisfiable until we reach the fixed-point. Then it is unsatisfiable. In other words, in all but one case the formula is satisfiable. A variable assignment satisfying the formula corresponds to a state which has just been added (or removed) from the approximation to the fixed-point. It is our experience that SAT-solvers are good at finding a satisfying variable assignment so we suggest using a SAT-solver here. Of course, we do not need to detect the fixed-point as soon as we reach it. It is possible to take one or more extra iterations. For example, if we know the number of iterations to be at least n, we could skip the satisfiability check the first n times. Another strategy would be to only check for satisfiability every k steps, where k is a small constant.

Second we need to determine whether the initial set of states I is a subset of the set of states $[\![\phi]\!]$ represented by the CTL specification: $\mathrm{TAUT}(I \rightarrow [\![\phi]\!])$. There are two cases:

- The specification holds. This means that $I \rightarrow [\![\phi]\!]$ is a tautology. We could use a SAT-solver to prove that the negation of $I \rightarrow [\![\phi]\!]$ is not satisfiable. However, it is our experience that most SAT-solvers are not very good at proving a formula to be unsatisfiable. We can also use BDDs. By using the UP_ONE algorithm, we can convert the BED for $I \rightarrow [\![\phi]\!]$ to a BDD. This results in the BED **1**.

- The specification does not hold. A proof is a variable assignment falsifying $I \rightarrow [\![\phi]\!]$. Or equivalent, a variable assignment satisfying $\neg(I \rightarrow [\![\phi]\!])$. SAT-solvers are good at finding such variable assignments.

Of course, in general we do not know beforehand whether the specification holds. A possibility is to run a SAT-solver and a BED to BDD conversion in parallel. However, in some situations we do have a pretty good idea of what to expect:

Bug fixing : We have a faulty design. We make corrections to the design and verify it again. Most likely this will be an iterative process where we have a faulty design in all but the last model checking runs.

Optimizing : We have a correct design. We optimize part of it and verify the changes. Iteratively we optimize more and more of the design, each time verifying the changes. Most of the model checking runs will be successful.

As we do model checking more than once on almost identical designs, we can use the number of iterations to reach a fixed-point in the first run as an estimate of the number of iterations in the subsequent runs. Thus we may avoid some of the $\mathrm{SAT}(Q' \wedge Q)$ checks in the fixed-point algorithms.

Chapter 6 discusses in more detail how to solve the satisfiability problem using BEDs.

BED to CNF Conversion

SAT-solvers like GRASP [MSS99] and SATO [Zha97] expect their input to be a propositional formula in CNF. We must therefore convert our BEDs into CNF. For this conversion we use the technique of introducing new variables for every non-terminal vertex [BCC+99].

We convert a BED to CNF by introducing k extra variables – one for each non-terminal vertex in the BED. This avoids an exponential blowup of the size of the resulting CNF. However, we do increase the size of the state space by a factor of 2^k which is unfortunate.

Let V_u be a fresh, new variable for the non-terminal vertex u and the attribute $val(u)$ for the terminal vertex u. Then for each operator vertex v we create a clause

$$V_v \;\leftrightarrow\; V_{low(v)}\; op(v)\; V_{high(v)}\,,$$

and for each variable vertex v we create a clause

$$V_v \;\leftrightarrow\; \big(var(v) \wedge\; V_{high(v)}\big) \vee\; \big(\neg var(v) \wedge V_{low(v)}\big)\;.$$

Each of these clauses can easily be expanded into CNF. For example, an operator vertex v with $op(v) = NOR$, $low(v) = l$, and $high(v) = h$ translates to the following CNF:

$$V_v \;\leftrightarrow\; V_l \;\bar{\vee}\; V_h \;=\; (\neg V_v \vee \neg V_l) \wedge (\neg V_v \vee \neg V_h) \wedge (V_v \vee V_l \vee V_h)\,.$$

The CNF of the whole BED is the conjunction of all the clauses.

Note that the resulting CNF formula is not equivalent to the original BED formula as we have introduced extra variables. However, one formula is satisfiable if and only if the other formula is satisfiable.

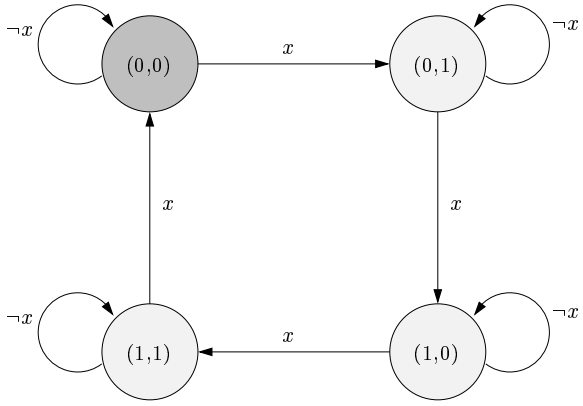

Figure 5.6: A modulo-4 counter which counts every time the input variable x is true. Each state is labeled with a pair (s_1, s_0). The initial state $(0, 0)$ is colored dark-gray.

5.3.3 Example

In this section we give an example of model checking using BEDs. We use a modulo-4 counter which only counts one type of events. The counter should start out being zero. Every time an event e happens, the counter should increment by one. The increments are done modulo 4. If an event other than e happens (we also consider *idle* an event, namely the event that no other event happens), then the counter keeps its value.

To implement such a modulo-4 counter, we need two Boolean state variables s_0 and s_1. The value of the counter is the binary number $s_1 s_0$. The input variable x models the presence and absence of event e: x is true if event e takes place[6].

Consider Figure 5.6. It shows the Kripke structure for the modulo-4 counter. The SMV program in Figure 5.7 implements the counter.

Now we construct the formulae for the set of initial states I and the transition function T. There is only one initial state, namely the state where both state variables are 0. We write I in the form of Equation 5.2:

$$I = (s_0 \leftrightarrow 0) \wedge (s_1 \leftrightarrow 0)\,.$$

We construct the BEDs corresponding to I_0 and I_1 in Equation 5.2. Both

[6]Normally x is the vector of input variables, but in this case we only need one input variable and we just call it x.

```
MODULE main
VAR
  s0 : boolean;
  s1 : boolean;
  x  : boolean;

ASSIGN
  init(s0) := 0;      -- I_0
  init(s1) := 0;      -- I_1

  next(s0) :=         -- T_0
    case
      !x : s0;
      !s1 : s0;
      1 : !s0;
    esac;
  next(s1) :=         -- T_1
    case
      !x : s1;
      1 : !s1;
    esac;
```

Figure 5.7: SMV program for a modulo-4 counter.

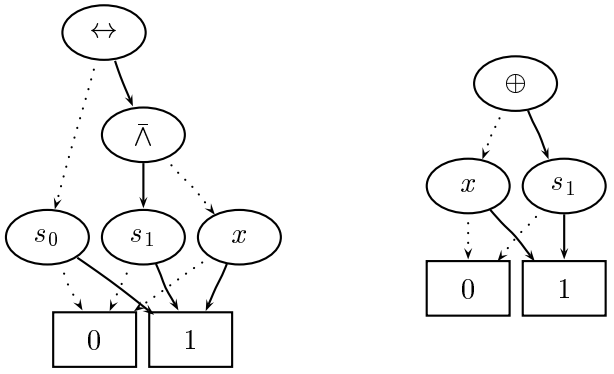

Figure 5.8: The BEDs for the transition function functions T_0 (left) and T_1 (right). The rewriting rules have been applied to reduce the BEDs.

BEDs are **0**. Likewise, we write T in the form of Equation 5.1:

$$\begin{aligned} s' \leftrightarrow T &= s'_0 \leftrightarrow \Big((\neg x \wedge s_0) \vee (x \wedge \neg s_1 \wedge s_0) \vee (x \wedge s_1 \wedge \neg s_0) \Big) \\ &\wedge \quad s'_1 \leftrightarrow \Big((\neg x \wedge s_1) \vee (x \wedge \neg s_1) \Big) . \end{aligned}$$

We construct the BEDs corresponding to T_0 and T_1 in Equation 5.1; see Figure 5.8.

For the specification we choose the liveness property: It should always be possible for the counter to reach the initial state. In CTL this becomes $\phi = \mathbf{AG}\ \mathbf{EF}\ (\neg s_0 \wedge \neg s_1)$.

We have to compute $[\![\phi]\!] = [\![\mathbf{AG}\ \mathbf{EF}\ (\neg s_0 \wedge \neg s_1)]\!]$. We start by computing $[\![\mathbf{EF}\ (\neg s_0 \wedge \neg s_1)]\!]$:

$$\begin{aligned} & [\![\mathbf{EF}\ (\neg s_0 \wedge \neg s_1)]\!] \\ = \ & [\![\mathbf{EU}\ (1, \neg s_0 \wedge \neg s_1)]\!] \\ = \ & \textsc{Lfp}\ Z \left[[\![\neg s_0 \wedge \neg s_1]\!] \vee \mathbf{EX}\ Z \right] \end{aligned}$$

Using Algorithm 5.3 we compute the least fixed-point as a series of approx-

imations $Q_0, Q_1, \ldots$:

$$\begin{aligned}
Q_0 &= 0 \\
Q_1 &= \left(s_0 \bar{\vee} s_1\right) \vee \mathbf{EX}\ 0 \\
&= s_0 \bar{\vee} s_1 \\
Q_2 &= \left(s_0 \bar{\vee} s_1\right) \vee \mathbf{EX}\, Q_1 \\
&= \left(s_0 \bar{\vee} s_1\right) \vee \Big(\exists x : \big((s_0 \leftrightarrow (s_1 \bar{\wedge} x)) \bar{\vee} (x \oplus s_1)\big)\Big) \\
&= \left(s_0 \bar{\vee} s_1\right) \vee \Big(\exists x : \big((s_0 \leftrightarrow (s_1 \bar{\wedge} x)) \not\leftarrow (x \leftrightarrow s_1)\big)\Big) \\
&= \left(s_0 \bar{\vee} s_1\right) \vee \left(s_0 \leftrightarrow s_1\right) \\
&= s_0 \leftrightarrow s_1
\end{aligned}$$

Q_1 is $s_0 \bar{\vee} s_1$, which is the result of the BED simplifications from Section 3.2 applied to $\neg s_0 \wedge \neg s_1$. The Q_2 approximation is $\left(s_0 \bar{\vee} s_1\right) \vee \mathbf{EX}\, Q_1$. The second line in the Q_2 calculation shows the result after quantification by substitution by Algorithm 5.4. The third line shows the simplified formula. The fourth line shows the formula after quantification of the input x. The fifth and final line shows the fully simplified Q_2.

To determine whether Q_1 is the fixed-point, we check $\mathrm{SAT}(Q_2 \wedge \neg Q_1)$. The variable assignment $(s_1, s_0) = (1, 1)$ is a witness and thus Q_1 is not the fixed-point.

It is possible to relate the fixed-point calculations to Figure 5.6. The function Q_0 corresponds to the empty set of states. Q_1 corresponds to the singleton set $\{(0, 0)\}$. The function Q_2 corresponds to $\{(0, 0), (1, 1)\}$.

$$\begin{aligned}
Q_3 &= \left(s_0 \bar{\vee} s_1\right) \vee \Big(\exists x : \big((s_0 \leftrightarrow (s_1 \bar{\wedge} x)) \leftrightarrow (x \oplus s_1)\big)\Big) \\
&= s_0 \leftarrow s_1 \\
Q_4 &= \left(s_0 \bar{\vee} s_1\right) \vee \Big(\exists x : \big((s_0 \leftrightarrow (s_1 \bar{\wedge} x)) \leftarrow (x \oplus s_1)\big)\Big) \\
&= 1 \\
Q_5 &= \left(s_0 \bar{\vee} s_1\right) \vee \exists x : 1 \\
&= 1
\end{aligned}$$

First after calculating Q_5 do we detect the fixed-point. Both $\mathrm{SAT}(Q_3 \wedge \neg Q_2)$ and $\mathrm{SAT}(Q_4 \wedge \neg Q_3)$ are true, but $\mathrm{SAT}(Q_5 \wedge \neg Q_4)$ is false and thus Q_4 is the fixed-point. We have now computed $[\![\mathbf{EF}\ (\neg s_0 \wedge \neg s_1)]\!]$ to be 1.

The function Q_3 corresponds to $\{(0, 0), (1, 1), (1, 0)\}$, and both Q_4 and Q_5 correspond to the set of all states $\{(0, 0), (1, 1), (1, 0), (0, 1)\}$.

We are now ready to compute $[\![\phi]\!] = [\![\mathbf{AG\ EF}\ (\neg s_0 \wedge \neg s_1)]\!]$:

$$\begin{aligned} & [\![\mathbf{AG\ EF}\ (\neg s_0 \wedge \neg s_1)]\!] \\ = \ & [\![\mathbf{AG}\ 1]\!] \\ = \ & [\![\neg\mathbf{EU}(1, \neg 1)]\!] \\ = \ & \neg \textsc{Lfp}\ Z\ [\,\mathbf{EX}\ Z\,] \end{aligned}$$

Again we compute a series of approximations $Q_0, Q_1, \ldots$ to the least fixed-point:

$$\begin{aligned} Q_0 &= 0 \\ Q_1 &= 0 \vee \mathbf{EX}\ 0 \\ &= 0 \end{aligned}$$

We are already at the fixed-point as $\mathrm{SAT}(Q_1 \wedge \neg Q_0) = \mathrm{SAT}(0)$ is false. Thus $[\![\phi]\!] = [\![\mathbf{AG\ EF}\ (\neg s_0 \wedge \neg s_1)]\!] = \neg 0 = 1$.

To determine whether the specification holds, we check $\mathrm{TAUT}(I \rightarrow [\![\phi]\!])$:

$$\begin{aligned} & \mathrm{TAUT}(I \rightarrow [\![\phi]\!]) \\ = \ & \mathrm{TAUT}(\neg s_0 \wedge \neg s_1 \rightarrow 1) \\ = \ & \mathrm{TAUT}(1) \\ = \ & 1 \end{aligned}$$

$I \rightarrow [\![\phi]\!]$ is a tautology. This means that the specification holds for our modulo-4 counter. This can, of course, trivially been seen from Figure 5.6.

5.4 Experimental Results

We have constructed a prototype implementation of our proposed model checking method. It performs CTL model checking on SMV programs. For the experiments presented here we use SATO as our SAT-solver. We compare our method with the NUSMV model checker (release 1.1) [CCGR99] and with Bwolen Yang's modified version of SMV[7], both of which are state-of-the-art in BDD-based model checking. Finally we compare reachability results with FIXIT from Adbulla, Bjesse, and Eén [ABE00].

[7] See `http://www.cs.cmu.edu/~bwolen`.

The FixIt results are taken directly from the paper by Abdulla and his group[8]. All other experiments are run on a Linux computer with a Pentium Pro 200 MHz processor and 1 gigabyte of main memory.

5.4.1 Multiplier

This example comes from the BMC-1.0f distribution[9]. It is a $16 \times 16 \mapsto 32$ bit shift-and-add multiplier. The specification is the c6288 combinational multiplier from the ISCAS'85 benchmark series [BF85]. For each output bit we verify that we cannot reach a state where the shift-and-add multiplier has finished its computation and the output bits of the two multipliers differ.

The multiplier fits into the category of SMV programs that we handle well. The operands are not modeled as inputs. Instead they are modeled as state variables with an unspecified initial state and the identity function as the next-state function. This lets us use quantification by substitution for all quantifications in the fixed-point calculations.

Table 5.2 shows the runtimes for verifying that the multiplier satisfies the specification. Our BED-based method out-performs both NuSMV and Bwolen Yang's SMV as we are able to model check twice as many outputs as they do. FixIt handles the same number of outputs as our method, however, our method is faster by up to an order of magnitude.

For the most difficult output in Table 5.2, the fixed-point iteration accounts for only a fraction of the total runtime for our method. It takes less than a minute and almost no memory to calculate the fixed-point. By far the most time is spent in proving $\mathrm{TAUT}(I \rightarrow [\![\phi]\!])$. SAT-solvers gave poor results, so we converted the BED for $I \rightarrow [\![\phi]\!]$ to a BDD. The FixIt tool uses a SAT-solver to check $\mathrm{TAUT}(I \rightarrow [\![\phi]\!])$. We expect this is the reason why their runtimes are much longer than ours. However, FixIt does not use much memory, while the memory required for the BED to BDD conversion is quite large. Of course, this is expected since the formulae originate from multiplier circuits which are known to be difficult for BDDs. But even though we have to revert to BDDs, we still outperform standard BDD-based model checkers.

Figure 5.9 shows the runtimes from Table 5.2 for the FixIt and the BED methods as a graph. Up to output 8, the BED runtimes are dominated by the fixed-point computations. The runtimes for output 9 to 12 are dominated

[8] From personal correspondence with the authors we have learned that they used a 296 MHz Sun UltraSPARC-II for the barrel shifter experiments and a 333 MHz Sun UltraSPARC-IIi for the multiplier experiments.

[9] See `http://www.cs.cmu.edu/~modelcheck`.

Bit	BED	NuSMV	Bwolen	FixIt
0	2.2	11	9.4	2.9
1	2.3	23	17	3.1
2	2.9	50	33	3.7
3	3.8	130	71	4.8
4	5.2	290	159	6.6
5	7.0	702	383	11
6	9.2	-	1031	20
7	12	-	-	47
8	16	-	-	150
9	31	-	-	544
10	68	-	-	2078
11	352	-	-	8134
12	2201	-	-	30330

Table 5.2: Runtimes in seconds for verifying the correctness of a 16-bit multiplier. A dash indicates that the verification could not be completed with 800 MB of memory.

by the BDD to BED conversion of $I \to [\![\phi]\!]$. The size of the BED for $I \to [\![\phi]\!]$ as a function of the output grows as a polynomial with degree around 3/2. Since the BDD to BED conversion is exponential, this explains the super-exponential curve for the BED method. It looks as if the FixIt method has a better asymptotic behavior and will from output 14 or 15 be the faster method. However, both methods have at least exponential runtimes and neither method handles more than the first 13 outputs at the moment.

We did the experiments in Table 5.2 without use of iterative squaring to enable fair comparisons. However, iterative squaring speeds up the fixed-point calculations. Table 5.3 shows the runtimes for calculating the fixed-points – with and without iterative squaring – for the same model checking problem as above. Note the case for bit 30 where iterative squaring allows us to calculate the fixed-point. Without iterative squaring the SAT-solver gets stuck. After each iteration the SAT-solver looks for new states. With iterative squaring many more new states are added per iteration making it easier for the SAT-solver to find a satisfying assignment.

To see how our method handles erroneous designs, we introduced an error in the specification of the multiplier by negating one of the internal nodes (this is marked as "bug D" in the multiplier file in the BMC distribution). We observe that the fixed-points are computed in roughly the same amount of CPU time and memory (both with and without iterative squaring). The

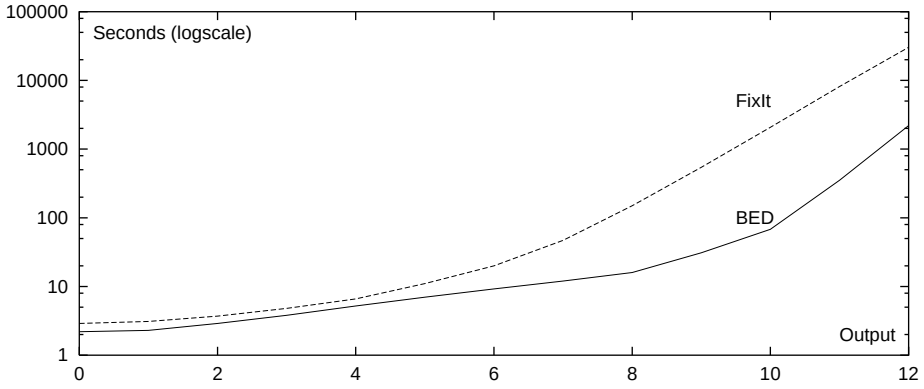

Figure 5.9: Graph of runtimes for the multiplier example in Table 5.2 as a function of the output number. Only the runtimes for the FIXIT and BED methods are shown.

difference is when we prove TAUT($I \rightarrow [\![\phi]\!]$). Using BED to BDD conversion as with the correct design, we now get poorer results because $I \rightarrow [\![\phi]\!]$ is *not* a tautology and the final BDD is not necessarily small. However, using a SAT-solver, we get much better results. In many cases, the SAT-solver is able to find a counterexample almost immediately. We are able to model check all but one output (bit 30) of the multiplier using less than 16 MB of memory and a few minutes of CPU time per output. Using iterative squaring we are able to model check bit 30 in 3.9 seconds. NUSMV and Bwolen Yang's SMV perform as badly as before. Table 5.4 shows the results for our method.

We were able to find a bug in the "correct" specification of the multiplier for the two most significant outputs. Iterative squaring allowed us to quickly compute the fixed-points, and SATO instantly found the errors. The total runtimes to find these errors were seven and eight seconds, respectively. It turns out that the two outputs have been swapped. The original net-list for c6288 does not contain information about which gates correspond to which multiplier outputs. However, each gate is numbered and the output numbers seem to be increasing with the gate numbers – with the exception of the last pair of outputs. This emphasizes the fact that SAT-based methods are good at finding bugs in a system.

We constructed shift-and-add multipliers of different sizes and verified that they always terminate, i.e., we checked "**AF** done". The number of iterations needed to reach the fixed-point is equal to the size of the multiplier.

Bit	Without I.S.	With I.S.
0	2.1	0.9
5	6.8	1.6
10	14	3.7
15	16	8.3
20	37	12
25	19	8.8
30	> 12 *hours*	6.4

Table 5.3: Runtimes in seconds for the fixed-point calculation in verifying the correctness of the 16-bit shift-and-add multiplier. Results are shown for computations with and without iterative squaring (I.S.). The space requirements are small, i.e., less than 16 MB.

This lets us test how well our method handles cases with lots of iterations. Table 5.5 shows the results. We compare our method with NuSMV and Bwolen Yang's SMV. Our method performs much better as we are both significantly faster and we are able to handle much larger designs. We cannot compare with FixIt as it does not handle **AF** properties.

5.4.2 Barrel Shifter

This example is a barrel shifter from the BMC-1.0f distribution and like the multiplier, it also falls within the category of systems which we handle well. A barrel shifter consists of two register files. The contents of one of the register files is rotated at each step while the other file stays the same. The width of a register is $log_2\ R$, where R is the number of registers in the register file.

The correctness of the barrel shifter is proven by showing that if two registers from the files have the same contents, then their neighbors are also identical. The set of initial states is restricted to states where this invariant holds. The left part of Table 5.6 shows the results. The BED and FixIt methods are both fast, however, the BED method scales better and thus outperforms FixIt. NuSMV and Bwolen Yang's SMV are both unable to construct the BDD for the transition relation for all but the smallest examples.

We prove liveness for the barrel shifter by showing that a pair of registers in the files will eventually become equal. The number of iterations for the fixed-point calculation is equal to the size of the register file. The right part of Table 5.6 shows the results. FixIt cannot handle liveness properties so

Bit	Runtime	Bit	Runtime
0	2.4	16	41
1	3.4	17	32
2	3.9	18	32
3	4.6	19	375
4	5.6	20	176
5	6.8	21	94
6	8.3	22	115
7	9.7	23	34
8	11	24	51
9	17	25	103
10	22	26	53
11	42	27	69
12	37	28	101
13	49	29	176
14	105	30	[> 600]
15	127	31	19

Table 5.4: Runtimes in seconds for model checking the 16-bit multiplier with bug D. The first 3 outputs are unaffected by the bug. The rest of the outputs are erroneous. The verification of output 30 was no completed in 10 minutes, however, using iterative squaring we can complete the model checking in 3.9 seconds. NuSMV and Bwolen Yang's SMV give results similar to those of the correct multiplier; see Table 5.2.

Size	BED	NuSMV	Bwolen
16	1.6	2.2	5.2
18	1.8	18	9.1
20	2.0	90	24
22	2.3	472	104
23	2.7	-	253
24	2.8	-	-
32	3.7	-	-
64	17	-	-
128	119	-	-
256	1185	-	-

Table 5.5: Runtimes in seconds for verifying that shift-and-add multipliers of different sizes always terminate, i.e., we check "**AF** done". The number of iterations to reach the fixed-point is equal to the size of the multiplier. A dash indicates that the verification could not be completed with 800 MB of memory.

we cannot compare with it. As in the previous case, NuSMV and Bwolen Yang's SMV can only handle small examples.

5.5 Model Checking of lfp-CTL

Symbolic model checking using fixed-points involves quantification of both state and input variables. A naïve 0/1 expansion as in Equation 5.3 does not work in practice. The formulae blow up in size after few quantifications. The model checking method presented so far in this chapter uses quantification by substitution to quantify out the state variables. It has proven effective for systems with few or no input variables.

Bounded Model Checking (BMC) [BCC+99, BCCZ99, BCRZ99] uses an unfolding of the transition relation. Each unfolding gives rise to a new set of state and input variables. Instead of quantifying out the variables, BMC simply leaves them in the formulae.

In this section we examine how to do model checking using quantification by substitution for state variables and unfolding for input variables. We leave the input variables in the formulae instead of quantifying them out. Unfortunately, this means we cannot detect fixed-points and hence we restrict ourselves to a fixed-depth. Another drawback is that we have to restrict ourselves to a subset of CTL. However, by placing ourselves half way between BMC and standard fixed-point methods, we can hope to handle

Size	BED	NuSMV	Bwolen	FixIt
2	0.1	0.1	1.0	0.1
4	0.3	0.2	2.5	0.1
6	0.4	609	-	0.2
8	0.4	-	-	0.5
10	0.6	-	-	1.1
20	1.9	-	-	14
30	4.0	-	-	52
40	8.0	-	-	231
50	13	-	-	502
60	19	-	-	?
70	30	-	-	?

Size	BED	NuSMV	Bwolen
2	0.2	0.1	1.0
4	0.5	0.2	2.1
6	0.7	521	-
8	0.9	-	-
10	1.2	-	-
20	3.2	-	-
30	5.9	-	-
40	11	-	-
50	18	-	-
60	28	-	-
70	47	-	-

Table 5.6: Runtimes in seconds for invariant (left) and liveness (right) checking of the barrel shifter example. A question mark indicates that the runtime for FixIt was not reported in [ABE00]. For the BED method we use Sato for checking $\mathrm{TAUT}(I \rightarrow [\![\phi]\!])$. A dash indicates that the verification could not be completed with 800 MB of memory.

designs which neither of those methods can handle.

5.5.1 Theory

We use the same setup as described in Section 5.2, i.e., we use Kripke structures to represent finite state machines. Properties of systems are still modeled in CTL. However, in this section we only consider CTL formulae on *negation normal form*. This means formulae where the negations are only on atomic propositions, and conjunction and disjunction are the only two binary Boolean connectives. We further restrict ourselves to a subset of negation normal form CTL without the globally **G** and release **R** operators. The subset is called lfp-CTL and is defined in Definition 5.5.1. The remaining temporal operators are next state **X**, future **F** and until **U**. Definition 5.5.2 gives the semantics of lfp-CTL.

Definition 5.5.1 (lfp-CTL Syntax). *lfp-CTL* is the subset of negation normal form CTL where all greatest fixed-point operators (globally and release operators) have been removed:

$$\begin{aligned} f \quad ::= \quad & 0 \mid 1 \mid s_i \mid \neg s_i \mid f \vee f \mid f \wedge f \mid \mathbf{EX}\, f \mid \mathbf{EF}\, f \mid \mathbf{EU}(f, f) \mid \\ & \mathbf{AX}\, f \mid \mathbf{AF}\, f \mid \mathbf{AU}(f, f)\,, \end{aligned}$$

where s_i is any state variable.

Definition 5.5.2 (lfp-CTL Semantics). Given a Kripke structure $M = (S, X, I, T, \ell)$, the semantics of an lfp-CTL formula is a set of states $[\![\cdot]\!] \subseteq \mathcal{P}(S)$. In terms of characteristic functions, $[\![\cdot]\!]$ is a Boolean function of the state vector s defined recursively as follows:

$$
\begin{aligned}
[\![0]\!] &= 0 \\
[\![1]\!] &= 1 \\
[\![s_i]\!] &= s_i \\
[\![\neg s_i]\!] &= \neg s_i \\
[\![\phi_1 \vee \phi_2]\!] &= [\![\phi_1]\!] \vee [\![\phi_2]\!] \\
[\![\phi_1 \wedge \phi_2]\!] &= [\![\phi_1]\!] \wedge [\![\phi_2]\!] \\
[\![\mathbf{EX}\ \phi]\!] &= \exists s', x : \big((s' \leftrightarrow T(s,x)\big) \wedge [\![\phi]\!][s'/s] \\
[\![\mathbf{AX}\ \phi]\!] &= \neg[\![\mathbf{EX}\ \neg\phi]\!] \\
[\![\mathbf{EF}\ \phi]\!] &= \textsc{Lfp}\ Z \left[[\![\phi]\!] \vee \mathbf{EX}\ Z \right] \\
[\![\mathbf{AF}\ \phi]\!] &= \textsc{Lfp}\ Z \left[[\![\phi]\!] \vee \mathbf{AX}\ Z \right] \\
[\![\mathbf{EU}(\phi_1, \phi_2)]\!] &= \textsc{Lfp}\ Z \left[[\![\phi_2]\!] \vee \big([\![\phi_1]\!] \wedge \mathbf{EX}\ Z\big) \right] \\
[\![\mathbf{AU}(\phi_1, \phi_2)]\!] &= \textsc{Lfp}\ Z \left[[\![\phi_2]\!] \vee \big([\![\phi_1]\!] \wedge \mathbf{AX}\ Z\big) \right]
\end{aligned}
$$

where $[\![\phi]\!][s'/s]$ is a substitution of s' for s in $[\![\phi]\!]$.

The set transformers in Definition 5.5.2 are all monotonic. The proofs are similar to the one in Lemma 5.2.11.

Lemma 5.5.3. *Let τ be a monotonic set transformer. Let $\tau^k(\emptyset)$ be k applications of τ to $\emptyset$: $\tau(\tau(\ldots\tau(\emptyset)\ldots))$. Then $\tau^k(\emptyset) \subseteq \tau^{k+1}(\emptyset)$.*

Proof. (by induction) The lemma holds for $k = 0$: $\emptyset \subseteq \tau(\emptyset)$. Assume it holds for $k = j - 1$: $\tau^{j-1}(\emptyset) \subseteq \tau^j(\emptyset)$. As τ is monotonic, we can apply τ on each side: $\tau(\tau^{j-1}(\emptyset)) \subseteq \tau(\tau^j(\emptyset))$ After regrouping we get $\tau^j(\emptyset) \subseteq \tau^{j+1}(\emptyset)$, which shows that the lemma holds for $k = j$. □

Definition 5.5.4 (k-CTL). $[\![\phi]\!]_k$ is the set of states represented by CTL formula ϕ in which we make exactly k iterations (k applications of the set transformer τ) in each fixed-point computation.

The following two theorems state that for lfp-CTL we can determine an under-approximation to $[\![\phi]\!]$, and if we know the diameter of the system M, we can compute $[\![\phi]\!]$ exactly.

Theorem 5.5.5 (Under-approximation). *Let ϕ be an lfp-CTL formula. Then, for all non-negative integers k,*

$$[\![\phi]\!]_k \subseteq [\![\phi]\!]_{k+1} \subseteq [\![\phi]\!] .$$

Proof. We first prove $[\![\phi]\!]_k \subseteq [\![\phi]\!]_{k+1}$ by induction over the recursive depth of the semantics in Definition 5.5.2. We use set notation in the argumentation. The four base cases 0, 1, s_i and $\neg s_i$ trivially hold. The disjunction case $[\![\phi_1 \vee \phi_2]\!]_k$ is equal to $[\![\phi_1]\!]_k \cup [\![\phi_2]\!]_k$, which per induction hypothesis is a subset of $[\![\phi_1]\!]_{k+1} \cup [\![\phi_2]\!]_{k+1}$, which is equal to $[\![\phi_1 \vee \phi_2]\!]_{k+1}$. A similar arguments is valid for the conjunction case. For $[\![\mathbf{EX}\ \phi]\!]_k$ we argue that it is the set of all states s such that there exists an s' where $s \rightsquigarrow s'$ and $s' \in [\![\phi]\!]_k$. Per induction hypothesis $[\![\phi]\!]_k \subseteq [\![\phi]\!]_{k+1}$, and thus $s \in [\![\mathbf{EX}\ \phi]\!]_{k+1}$. A similar argument holds for $[\![\mathbf{AX}\ \phi]\!]_k$.

All the remaining cases involve the least fixed-point operator with a monotonic set transformer τ. We prove the inclusion for $[\![\mathbf{EF}\ \phi]\!]_k$; the other cases are handled in a similar way. Let τ_1 be the set transformer $[\![\phi]\!]_k \cup \mathbf{EX}\ Z$ and τ_2 the set transformer $[\![\phi]\!]_{k+1} \cup \mathbf{EX}\ Z$. We prove by induction that $\tau_1^k(\emptyset) \subseteq \tau_2^{k+1}(\emptyset)$. The base case $k = 0$ holds trivially. For $k = i$, assume $\tau_1^{i-1}(\emptyset) \subseteq \tau_2^i(\emptyset)$. Let s be some state in $\tau_1(\tau_1^{i-1}(\emptyset)) = \tau_1^{i-1}(\emptyset) \cup \mathbf{EX}\ \tau_1^{i-1}(\emptyset)$. Remember that $\tau_2(\tau_2^i(\emptyset)) = \tau_2^i(\emptyset) \cup \mathbf{EX}\ \tau_2^i(\emptyset)$. If $s \in \tau_1^{i-1}(\emptyset)$, then per induction hypothesis $s \in \tau_2(\tau_2^i(\emptyset))$. Otherwise $s \in \mathbf{EX}\ \tau_1^{i-1}(\emptyset)$, in which case, per induction hypothesis, $s \in \mathbf{EX}\ \tau_2^i(\emptyset)$. In either case we have that $\tau_1^k(\emptyset) \subseteq \tau_2^{k+1}(\emptyset)$ and thus $[\![\mathbf{EF}\ \phi]\!]_k \subseteq [\![\mathbf{EF}\ \phi]\!]_{k+1}$.

We now prove $[\![\phi]\!]_k \subseteq [\![\phi]\!]$, however, we only consider the case where ϕ has an LFP-construction at the outermost level. Let s be some state in $[\![\phi]\!]_k$. Then it has been found within k iterations in LFP. Let d be the number of iterations to reach the fixed-point. If $k \geq d$, then $s \in [\![\phi]\!]$. Otherwise $k < d$. Because of Lemma 5.5.3, $s \in \tau^d(\emptyset)$, where τ is the set transformer for the LFP-construction. But $\tau^d(\emptyset)$ is just another name for $[\![\phi]\!]$, so $s \in [\![\phi]\!]$. □

Theorem 5.5.6 (Completeness). *Let ϕ be an lfp-CTL formula. Then there exists a non-negative number d such that:*

$$[\![\phi]\!]_d = [\![\phi]\!] .$$

We call the least such d the diameter of the system M.

Proof. First we prove the existence of d. From Theorem 5.5.5 we know that $[\![\phi]\!]_k \subseteq [\![\phi]\!]_{k+1}$. For each LFP-computation, each step either adds at least one new state or we are at a fixed-point. Since the state space is finite there exists some finite d such that: $[\![\phi]\!]_0 \subseteq [\![\phi]\!]_1 \subseteq \ldots \subseteq [\![\phi]\!]_d = [\![\phi]\!]_{d+1} = [\![\phi]\!]_{d+2} = \ldots$.

Now we prove that $[\![\phi]\!]_d = [\![\phi]\!]$. Let s be some state in $[\![\phi]\!]_d$. Then s is also in $[\![\phi]\!]_{d+i}$ for all $i > 0$ and thus more iterations in the least fixed-point calculations do not result in any new states. We must therefore have reached the fixed-point in all LFP-computations. Hence $[\![\phi]\!]_d = [\![\phi]\!]$. □

Unfortunately, in practice it is difficult to find the diameter of a system short of actually performing a fixed-point calculation. It is also difficult to give useful bounds on the diameter. Without a tight bound on the diameter, our method is not complete in practice.

Instead of computing $[\![\phi]\!]_k$, we compute a related formula $[\phi]_k$, where $[\cdot]_k$ is defined as:

Definition 5.5.7. Let ϕ be a lfp-CTL formula and let k be a non-negative integer. Then $[\phi]_k$ is a function of state and input variables such that

$$\left(\exists \bar{x} : [\phi]_k \right) \leftrightarrow [\![\phi]\!]_k$$

For a given ϕ and k, more than one function $[\phi]_k$ satisfy Definition 5.5.7. However, for our purpose this does not matter. We use $[\phi]_k$ to indicate *one* such function.

Definition 5.5.8 (α-renaming). Let ϕ be a function of input variables $x^i, \ldots, x^{i+j}$, then α is a renaming of the input variables such that $\alpha\phi$ is a function of $x^k, \ldots, x^{k+j}$ where $x^k, \ldots, x^{k+j}$ are new, fresh variable vectors.

We use $\bar{x}$ to indicate that extra input variables may have been added through α-renaming.

We now introduce the notion of s-equality and note that $[\cdot]_k$ and $[\![\cdot]\!]_k$ are s-equal.

Definition 5.5.9 (s-equality). Let ϕ_1 and ϕ_2 be formulae of state (s) and input ($\bar{x}$) variables. Then $=_s$ is the equivalence relation:

$$\phi_1 =_s \phi_2 \quad \equiv \quad \exists \bar{x} : \phi_1 \leftrightarrow \exists \bar{x} : \phi_2$$

We say ϕ_1 and ϕ_2 are s-equal if they relate through $=_s$.

Observation 5.5.10 shows a recursive way to compute $[\cdot]_k$. It is related to Definition 5.5.2, but with $=_s$ instead of $=$. We leave the input variables inside the formulae instead of quantifying them out.

Observation 5.5.10 (Computation of $[\cdot]_k$). Let ϕ be an lfp-CTL formula and let k be a non-negative integer. The function $[\phi]_k$ can be recursively computed as follows:

$$\begin{aligned}
[s_i]_k &=_s s_i \\
[\neg s_i]_k &=_s \neg s_i \\
[\phi_1 \vee \phi_2]_k &=_s [\phi_1]_k \vee [\phi_2]_k \\
[\phi_1 \wedge \phi_2]_k &=_s [\phi_1]_k \wedge \alpha[\phi_2]_k \\
[\mathbf{EX}\ \phi]_k &=_s \exists s' : \alpha\big((s' \leftrightarrow T(s,x)\big) \wedge [\phi]_k(s') \\
[\mathbf{AX}\ \phi]_k &=_s \neg\exists s' : \big(s' \leftrightarrow T(s,x)\big) \wedge \neg\exists \bar{x} : [\phi]_k(s') \\
[\mathbf{EF}\ \phi]_k &=_s \textsc{Lfp}_k\ Z\ \Big[\ [\phi]_k \vee [\mathbf{EX}\ Z]_k\ \Big] \\
[\mathbf{AF}\ \phi]_k &=_s \textsc{Lfp}_k\ Z\ \Big[\ [\phi]_k \vee [\mathbf{AX}\ Z]_k\ \Big] \\
[\mathbf{EU}(\phi_1,\phi_2)]_k &=_s \textsc{Lfp}_k\ Z\ \Big[\ [\phi_2]_k \vee (\alpha[\phi_1]_k \wedge [\mathbf{EX}\ Z]_k)\ \Big] \\
[\mathbf{AU}(\phi_1,\phi_2)]_k &=_s \textsc{Lfp}_k\ Z\ \Big[\ [\phi_2]_k \vee (\alpha[\phi_1]_k \wedge [\mathbf{AX}\ Z]_k)\ \Big]
\end{aligned}$$

where $[\phi]_k(s')$ is a shift of variables from s to s' in $[\phi]_k$ and $\textsc{Lfp}_k$ indicates the result of a least fixed-point algorithm after the first k iterations.

Theorem 5.5.11. *Let ϕ be an lfp-CTL formula and $M = (S, X, I, T, \ell)$ a Kripke structure. Then, for all non-negative integers k, if I is a* singleton *set,*

$$SAT(I \wedge [\![\phi]\!]_k) \ \rightarrow\ M \models \phi$$

Proof. In general, if I is a subset of $[\![\phi]\!]_k$, then $M \models \phi$. Since I is a singleton set, it is enough to prove the existence of an element in the intersection of I and $[\![\phi]\!]_k$. □

Lemma 5.5.12. *Let ϕ be an lfp-CTL formula and $M = (S, X, I, T, \ell)$ a Kripke structure. Then, for all non-negative integers k, if I is a singleton set,*

$$\begin{aligned}
&\big(SAT\,(I \wedge [\![\phi]\!]_k)\ \rightarrow\ M \models \phi\big) \\
\rightarrow\ &\big(SAT\,\big(I \wedge [\![\phi]\!]_{k+1}\big)\ \rightarrow\ M \models \phi\big)
\end{aligned}$$

Proof. Follows from Theorems 5.5.5 and 5.5.11. □

Theorem 5.5.13. *Let ϕ be an lfp-CTL formula and $M = (S, X, I, T, \ell)$ a Kripke structure. Then, for all non-negative integers k,*

$$SAT(I \wedge [\![\phi]\!]_k)\ =\ SAT(I \wedge [\phi]_k)$$

Proof. Apply the definition of $[\cdot]_k$, move the quantifier outward, and note that $SAT(\phi)$ corresponds to an existential quantification of all variables in ϕ:

$$\begin{aligned} SAT(I \wedge [\![\phi]\!]_k) &= SAT(I \wedge \exists \bar{x} : [\phi]_k) \\ &= SAT(\exists \bar{x} : I \wedge [\phi]_k) \\ &= SAT(I \wedge [\phi]_k) \end{aligned}$$

□

The restriction in Theorem 5.5.11 that I has to be a singleton set is to prevent an alternation of quantifiers. The typical check is whether I is a subset of $[\![\phi]\!]_k$. In terms of characteristic functions we check $\forall s : I \rightarrow [\![\phi]\!]_k$. However, since we compute $[\phi]_k$ and not $[\![\phi]\!]_k$, we would introduce an existential quantifier within the scope of the universal quantifier. By limiting ourselves to exactly *one* initial state, it is enough to check whether I and $[\![\phi]\!]_k$ have a state in common. It is possible to overcome this restriction by constructing a new system with a new single initial state $\tilde{i}$ and transitions from $\tilde{i}$ to all states in the old set of initial states I. Using the **AX** operator we go one step backward and only include $\tilde{i}$ if all states in I were in $[\![\phi]\!]$. Unfortunately, the computation of $[\mathbf{AX}\ \cdot]$ contains quantifications so unless we find some smart way of handling $[\mathbf{AX}\ \cdot]$, we have just pushed the problem from a restriction of I to **AX** computation.

5.5.2 Implementation

We assume that the set of initial states is a singleton set. Algorithm 5.10 shows the pseudo-code for the basic model checking algorithm ModelCheck.

Name: ModelCheck(M, ϕ, k)
1: Compute $[\phi]_k$ using Observation 5.5.10
2: **if** SAT$(I \wedge [\phi]_k)$ **then**
3: **return** "M models ϕ"
4: **else if** $k \geq$ diameter of M **then**
5: **return** "M does not model ϕ"
6: **else**
7: **return** "No result – increase k"

Algorithm 5.10: The ModelCheck algorithm. It determines whether a Kripke structure $M = (S, X, I, T, \ell)$ is a model for an lfp-CTL formula ϕ. The number of iterations in the fixed-point calculations is k.

Computing $[\phi]_k$

We use BEDs to represent the Boolean formulae. Each state variable corresponds to a BED variable. Boolean connectives (negation, conjunction, disjunction) correspond to operator vertices. α-renaming corresponds to variable substitution. We use quantification by substitution for quantification of state variables.

The **AX** operator causes trouble as it requires quantification of all input variables to compute $[\mathbf{AX}\ \phi]_k$. We cannot use quantification by substitution, so we are stuck with a naïve 0/1 expansion as in Equation 5.3. If we avoid the **AX** operator, we are still able to express **EX**, **EF**, and **EU**. That is enough to perform reachability analysis, or find errors in **AG** properties[10].

Computing SAT$(I \wedge [\phi]_k)$

Recall that we only have *one* initial state. Since we have a conjunction between this state and $[\phi]_k$, we can compute SAT$(I \wedge [\phi]_k)$ as SAT$([\phi]_k(I))$, where $[\phi]_k(I)$ means a substitution of all state variables in $[\phi]_k$ with their unique assignment from I. This leaves only input variables. SAT-solvers like GRASP [MSS99] and SATO [Zha97] can then be used to determine satisfiability.

5.6 Related Work

Model checking was invented by Clarke, Emerson, and Sistla in the 1980s [CES86]. Their model checking method required an explicit enumeration of states which limited the size of the systems they could handle. Burch *et. al.* [BCM$^+$92] showed how to do model checking without enumerating the states. They called this symbolic model checking. The idea is to represent sets of states by characteristic functions. The data structure of Binary Decision Diagrams turns out to be a very efficient representation for characteristic functions. The advantages of BDDs are compactness, canonicity, and ease of manipulation.

Biere, Clarke *et. al.* have proposed Bounded Model Checking (BMC) as an alternative method to BDD-based model checking [BCC$^+$99, BCCZ99, BCRZ99]. They unfold the transition relation and look for repeatedly longer and longer counterexamples, and they use SAT-solvers instead of BDDs. BMC is good at finding errors with short counterexamples. The diameter of

[10] If **AG** ϕ does not hold, then there is a state such that $\neg\phi$ holds. In other words, **EF** $\neg\phi$ holds.

the system determines the number of unfoldings of the transition relation. Unfortunately, for many examples the diameter cannot be calculated and the estimates are too rough. In such cases BMC reduces to a partial verification method in practice.

The work most closely related to ours is by Abdulla, Bjesse and Eén. They consider symbolic reachability analysis using SAT-solvers [ABE00]. For representing Boolean functions they use the Reduced Boolean Circuit data structure which closely resembles our Boolean Expression Diagrams. They perform reachability analysis using a fixed-point iteration, and like us they make use of the quantification by substitution rule. They use Stålmarck's patented method [SS98] to determine satisfiability of Boolean functions. While related, their method and ours differ in a number of ways: In our method the quantification by substitution rule is extensively used at three different places while they only use it during fixed-point calculation. We have heuristics for choosing different SAT-procedure depending on the expected result of the satisfiability check. Candidates are various SAT-solvers or an explicit BED to BDD conversion. They use Stålmarck's method as the only SAT-procedure used. BEDs are always locally reduced and we identify further important simplification rules. We also make use of iterative squaring. Finally, we do lfp-CTL model checking, something which they do not.

For a thorough description of model checking we refer the reader to Clarke, Grumberg and Peled's book *Model Checking* [CGP99].

There are other temporal logics than CTL. LTL and CTL* are two examples. Both logics do not require a path quantifier to immediately precede a path operator. This leads to the notion of both state and path formulae. A state formula describes a set of states whereas a path formula describes a set of paths. CTL contains only state formulae. In CTL* we have both state and path formulae. In LTL we have state formulae of the form $\mathbf{A}\,\phi$, where ϕ is a path formula.

The three temporal logics have different expressive powers. There are CTL formulae not expressible in LTL, and vice versa. CTL* is a superset of both CTL and LTL. Any CTL and LTL formula is expressible in CTL*, however, there are CTL* formulae not expressible in either CTL or LTL. Please see [CGP99] for a full discussion on CTL, LTL, and CTL*.

Converting a formula to CNF is necessary for standard SAT-procedures like Greedy SAT (GSAT) [SLM92] and Davis-Putnam [DP60, DLL62]. The CNF conversion may lead to an exponential growth. A way to overcome this is to introduce new variables, each representing a subformula in the original formula. Unfortunately, this greatly enlarges the search space for the SAT-

procedures. Research has been made in the area of applying SAT-procedures to formulae *not* in CNF. Giunchiglia and Sebastiani [GS99, Seb94] have examined GSAT and Davis-Putnam. Stålmarck's method also works on a non-CNF representation. Chapter 6 discusses how to do satisfiability checking on the BED data structure.

Symbolic model checking can be expressed in QBF. We can think of our model checking algorithm as a decision procedure for QBF. Both Cadoli *et. al.* [CGS98] and Rintanen [Rin99] have presented algorithms for evaluation of QBF. Their work has been centered on the AI community and they have, as far as we know, not experimented with their QBF algorithms on model checking problems.

On some systems model checking is not feasible due to state explosion. In such cases *Symbolic Trajectory Evaluation* (STE) [BBS91] may be used instead. STE is similar to simulation. However, instead of simulating just one input vector at a time, in STE we simulate a set of vectors at a time. STE can only verify limited properties like constraints on finite sequences of states. However, recently STE has been extended to handle all omega-regular properties. STE is actively used by companies like IBM, Motorola and Intel. Typically, STE is implemented on top of a BDD package. The number of BDD variables required in the verification depends on the property, and *not* on the system. This is opposed to symbolic model checking where the number of BDD variables depends on the system.

5.7 Conclusion

We have presented a BED-based CTL model checking method based on the classical fixed-point iterations. Quantification is often the Achilles heel in CTL fixed-point iterations but by using quantification by substitution we are in some cases able to deal effectively with it. While our method is complete, it performs best on examples with a low number of inputs. In this case we can fully exploit the quantification by substitution rule.

We have shown how the quantification by substitution rule can also help simplify the final set inclusion problem of model checking and help perform efficient iterative squaring. Our proposed method combines SAT-solvers and BED to BDD conversions to perform satisfiability checking. We use a set of local rewriting rules which helps to keep the size of the BEDs small.

We have demonstrated our method by model checking large shift-and-add multipliers and barrel shifters, and we obtain results superior to standard BDD-based model checking methods. Furthermore, we were able to

find a previously undetected bug in the specification of a 16-bit multiplier.

In order to deal with the input variables, we have proposed lfp-CTL model checking. Inputs are left in the formulae and SAT-solvers are used to implicitly quantify them out when needed. Unfortunately, this restricts the power of the logic. Only reachability analysis is practically possible.

Future work includes investigating two variable ordering problems. One is the variable ordering when converting the BED for $I \rightarrow [\![\phi]\!]$ to a BDD. The variable ordering is known to be very important in BDD construction, and since we, in some cases, spend much time on converting $I \rightarrow [\![\phi]\!]$ to a BDD, our method will benefit from a good variable ordering heuristic. The other problem is the order in which we quantify the variables in the $[\![\mathbf{EX}\ \cdot\ ]\!]$ computation. This is interesting especially in cases where we cannot use the quantification by substitution rule. Finally, it would be worth looking into finding a way around the quantification problem with **AX** in lfp-CTL.

Chapter 6

Satisfiability

In this chapter we show how to determine satisfiability of a formula represented by a Boolean Expression Diagram. We compare our method with traditional SAT-solvers and with BED to BDD conversion. This chapter is based on the paper [WAH01].

6.1 Introduction

In Chapter 3 we have dealt with the tautology problem for BEDs. Two flat combinational circuits are equivalent if $f \leftrightarrow g$ is a tautology, where f and g are the functions modeling the outputs of the two circuits. In case of an error, $f \leftrightarrow g$ is *not* a tautology. In other words, if $\neg(f \leftrightarrow g)$ is satisfiable then there is an error.

In Chapter 5 we determine whether $I \rightarrow [\![\phi]\!]$ is a tautology. If it is, then the CTL property ϕ holds for the system. Otherwise $\neg(I \rightarrow [\![\phi]\!])$ is satisfiable, and the property does *not* hold.

In both cases, tautology is associated with correct behavior while satisfiability is associated with errors.

Given a BED for a formula, one way of proving satisfiability is to convert the BED to a BDD. If the resulting BDD is 0 (a contradiction), then the formula is *not* satisfiable. Otherwise the formula is indeed satisfiable. Table 6.1 illustrates it for both satisfiability (SAT) and tautology (TAUT). Recall from Section 1.2 that there is the following relation between the two:

$$\text{SAT } \phi \quad = \quad \neg\text{TAUT } \neg\phi .$$

Converting a BED into a BDD allows us to determine satisfiability. However, we obtain more information than just a "yes, the formula is satisfiable"

Formula	SAT	TAUT
Tautology	yes	yes
Contradiction	no	no
Neither	yes	no

Table 6.1: The table shows the result of solving the satisfiability (SAT) and tautology (TAUT) problem for three different types of formulae: tautologies (always 1), contradictions (always 0), and formulae which can take both values. For readability we use "yes" and "no" instead of 1 and 0.

or a "no, the formula is not satisfiable" answer. The resulting BDD encodes *all* possible variable assignments satisfying the formula. This is overkill as we are only interested in *one* of them – and sometimes just the *existence* of one.

In Section 5.3.2 we discussed how to determine satisfiability for a formula represented by a BED. We converted the BED to a CNF and then fed it to a SAT-solver like SATO or GRASP. Such SAT-solvers are very efficient in proving satisfiability. Unfortunately, the conversion from BED to CNF introduces k new variables, where k is the number of non-terminal vertices in the BED. It is possible to avoid the extra k variables, but at the expense of a potential exponential blowup in the formula size.

6.2 Satisfiability Using Conjunctive Normal Form Formulae

A formula on conjunctive normal form (CNF) consists of a set of clauses. Each clause contains a number of literals, where a literal is either a variable or the negation of a variable. The literals within a clause are OR'ed together, whereas the clauses are AND'ed together.

The Davis-Putnam SAT-procedure [DP60, DLL62] works as described in Algorithm 6.1. Line 1 is the base case. Line 3 is the backtracking. Line 5 handles unit clauses, and line 8 and 9 handle splitting on literals. There are different heuristics for choosing a literal in line 8. One heuristic is to choose the literal in such a way that the assignments in line 9 produce the most unit clauses.

Name: DP ϕ
1: **if** ϕ is the empty set of clauses **then**
2: **return** 1
3: **else if** ϕ contains the empty clause **then**
4: **return** 0
5: **else if** a unit clause l occurs in ϕ **then**
6: **return** DP($assign(l, \phi)$)
7: **else**
8: $l \leftarrow$ *choose-literal*(ϕ)
9: **return** DP($assign(l, \phi)$) $\vee$ DP($assign(\neg l, \phi)$)

Algorithm 6.1: The basic version of Davis-Putnam. The function $assign(l, \phi)$ applies the truth value of literal l to the CNF formula ϕ. The function *choose-literal*(ϕ) picks a literal for DP to split on.

6.3 Satisfiability Using Boolean Expression Diagrams

The basis of Davis-Putnam is a splitting on literals. In BEDs we can obtain the same effect by pulling a variable to the root using Up_One. After pulling a variable x up using Up_One, there are two situations:

- The new root vertex is a variable x vertex. Both *low* and *high* children are BEDs. The formula is satisfiable if either the *low* child or the *high* child (or both) represent a satisfiable formula.
- The new BED does not contain the variable x anywhere. The formula does not depend on x and we can pick a new variable to pull up.

If at any point we reach the terminal **1**, then we know that the formula is satisfiable. This suggests a recursive algorithm which pulls up variables one at a time. The test for the empty set of clauses (line 1 in Algorithm 6.1) becomes a test for the terminal **1**. The test for whether ϕ contains the empty clause (line 3) becomes a test for the terminal **0**. We cannot find unit clauses with BEDs. The unit clauses are used to reduce the CNF formula. Instead we use another type of reductions: The rewriting rules from Section 3.2.2. Algorithm 6.2 shows the pseudo-code for the SAT-procedure BedSat.

The function *choose-variable* in line 6 of Algorithm 6.2 picks a variable to split on. With a clause form representation of the formula, it is natural to pick the variable in such a way as to obtain the most unit clauses after the split. This gives the most reductions due to unit propagation. We do

Name: BedSat u
1: **if** $u = \mathbf{1}$ **then**
2: **return** 1
3: **else if** $u = \mathbf{0}$ **then**
4: **return** 0
5: **else**
6: $x \leftarrow$ *choose-variable*(u)
7: $u' \leftarrow$ Up_One(x, u)
8: **if** u' is a variable x vertex **then**
9: **return** BedSat *low*(u') $\vee$ BedSat *high*(u')
10: **else**
11: **return** BedSat u'

Algorithm 6.2: The BedSat algorithm. The argument u is a BED. The function *choose-variable*(u) picks a variable to split on.

not have a clause form formulae. However, we can still choose the variable as to get the most reductions. We perform the splitting using Up_One. In Section 3.3 we discussed different heuristics for picking good variable orderings for Up_One. The first variable in such an ordering would probably be a good variable to split on. In BedSat we do not need to split on the variables in the same order along different branches. We have chosen a simple implementation which does a depth-first traversal of the BED and picks the first variable it encounters.

In line 9 the algorithm forks in two: one fork for the low child and one fork for the high child. If a satisfying assignment is found in one fork, then it is not necessary to consider the other one. We have implemented a simple strategy of first examining the fork with the smaller BED size (least number of vertices). We do not have any a priori knowledge of which fork to choose so picking the smaller one makes sense as the runtime of Up_One depends on the the size of the BED.

Figure 6.3 shows graphically how BedSat works. The circles correspond to splitting points and the triangles correspond to parts of the BED which have (gray triangles) or have not (white triangles) been examined. The numbers next to the triangles indicate the size of the state space represented by each triangle assuming that there are n variables in total. At any point during the algorithm we can compute the fraction of the state space examined so far by adding the numbers from the gray triangles and dividing by the size of the complete state space 2^n.

For a user, seeing the percentage of the state space examined so far is nice. It is often frustrating to do a computation and not knowing whether it is just about to finish or it is going nowhere. With BEDSAT it is very easy to compute the percentage. Of course, the percentage does not say anything about the time remaining in the computation. However, it does allow us to detect wether we are making progress. One could even imagine a more sophisticated SAT-solver which jumps back a number of splits if the user felt that the current choice of split variables did not produce any progress. Or we could do it automatically by tracking how the percentage changes over time. No or little growth could indicate that we were picking the wrong sequence of variables to split on and we should consider backtracking and picking new split variables. We call such backtracking *premature* since we give up the current series of splits, back up, and try a new one. Premature backtracking is also done in many implementations of Davis-Putnam. It is geared toward satisfiable functions since we give up our search in a particular part of the state space and concentrate on other, and hopefully easier, parts. If we find a satisfying assignment in the easy part of the state space then we never need to revisit the difficult parts. For unsatisfiable functions we need to examine the whole state space, and giving up on one part now just postpones the problems. The only hope is that by choosing a different sequence of variables to split on, the rewriting rules collapse the difficult part of the state space.

6.4 Experimental Results

To see how well BEDSAT works in practice, we compare it to other techniques for solving SAT problems. The problems we use in the comparison are from the ISCAS'85 benchmark series (see Chapter 3) and from model checking (see Chapter 5). All the problems have been turned into satisfiability problems.

All the experiments are performed on a 450 MHz Pentium III PC running Linux. We set a limit of 32 MB of memory for the BED data structure and 4 MB of memory for caches.

We compare BEDSAT to UP_ONE and UP_ALL with the FANIN variabel ordering heuristic. Furthermore, we compare with state-of-the-art SAT-solvers SATO and GRASP. Since both SATO and GRASP require their input to be in CNF form, we need to convert the BEDs to CNF using the method described in Section 5.3.2. This increases the number of variables and thus also the state space for SATO and GRASP. We do not compare with Stålmarck's method as it is targeted toward proving tautology and for

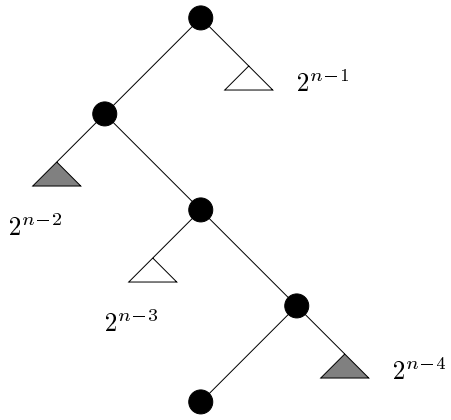

Figure 6.3: An illustration of the BEDSAT algorithm. Each circle represents a split on a variable. The top circle is the starting point. The triangles represent sub-BEDs; the white ones are as yet unexamined while the gray ones have already been examined. Assume that there are n variables in total and that the current position in BEDSAT corresponds to the bottom circle. Then the fraction of the state space which has already been examined is $\frac{2^{n-2}+2^{n-4}}{2^n}$.

non-tautology formulae, Stålmarck's method is not complete in practice.

Table 6.2 shows the ISCAS'85 results. The first ten rows represent 475 unsatisfiable functions. We mark this with "U" in the second column. UP_ONE and UP_ALL (both with the FANIN ordering heuristic) work quite well. The SAT-solvers SATO and GRASP perform well on the smaller circuits, but give up on some of the larger ones. BEDSAT does not perform well at all with runtimes which are an order of magnitude larger than the runtimes for the other methods. The long runtimes are due to BEDSAT's poor performance on some (but not all) unsatisfiable formulae. In our simple implementation of BEDSAT there is no premature backtracking. This means that if we chose a wrong sequence of split variables, then we are stuck with that sequence.

The last five rows of Table 6.2 show the results for the erroneous circuits. Here there are 340 functions in total out of which 267 are unsatisfiable and 73 are satisfiable. We indicate this with "S/U" in the second column. The UP_ONE and UP_ALL methods take slightly longer on the erroneous circuits since not all BDDs collapse to a terminal. The SAT-solvers (SATO, GRASP and BEDSAT) perform strictly better on the erroneous circuits compared to the correct circuits; sometimes going from impossible to possible as for

SATO and BEDSAT on c7552. BEDSAT is the only SAT-solver to handle c3540 and it outperforms SATO and GRASP on c5315. However, on c7552 BEDSAT is two orders of magnitude slower.

Consider the case of BEDSAT on c3540. In the correct version, BEDSAT uses 185 seconds. This number reduces to 35.9 seconds for the erroneous version. The c3540 example has 22 outputs where five are faulty in the erroneous version. BEDSAT has no problem detecting the errors in the five faulty outputs. In the correct version, about 149 seconds were spent on proving those five outputs to be unsatisfiable. Another example is c1908 where, in the correct case, BEDSAT spends all its time (242 seconds) on one unsatisfiable output. In the erroneous version the difficult output has an error and the corresponding function becomes satisfiable. BEDSAT finds a satisfying assignment instantaneously (0.1 seconds). This indicates that BEDSAT is not very good at handling unsatisfiable formulae.

Description	Result	UP_ONE	UP_ALL	SATO	GRASP	BEDSAT
c432/nr	U	2.1	1.7	0.5	0.4	36.4
c499/nr	U	4.3	1.8	1.8	1.4	17.8
c1355/nr	U	4.3	1.8	1.8	1.5	18.1
c1908/nr	U	0.7	0.6	0.4	0.4	242
c2670/nr	U	1.2	0.6	1.0	0.9	38.6
c3540/nr	U	32.3	39.2	-	-	185
c5315/nr	U	16.2	1.9	-	15.0	1.1
c6288/nr	U	2.7	-	-	-	-
c7552/nr	U	3.6	1.1	-	4.4	-
c1908/nr-err	S/U	0.7	0.6	0.4	0.4	0.1
c2670/nr-err	S/U	2.9	0.7	0.9	0.8	0.4
c3540/nr-err	S/U	42.8	40.2	-	-	35.9
c5315/nr-err	S/U	32.7	2.4	31.7	10.3	0.7
c7552/nr-err	S/U	8.1	1.8	2.5	2.6	176

Table 6.2: Runtimes in seconds for determining satisfiability of problems arising in verification of the ISCAS'85 benchmarks using different approaches. In the "Result" column, "U" indicates unsatisfiable problems while "S/U" indicates both satisfiable and unsatisfiable problems. Both UP_ONE and UP_ALL use the FANIN variable ordering heuristic. All methods were limited to 32 MB of memory and 15 minutes of CPU time. A dash indicates that the computation could not be done within the resource limits.

Table 6.3 shows the results for the model checking problems. We have

extracted satisfiability problems for the model checking experiments in Table 5.2 (first nine rows) and Table 5.4 (last nine rows). The numbers 10, 20 and 30 indicate the output bit we are considering. The word "final" indicates the satisfiability problem for the final check of the specification. The word "last_lp" indicates the satisfiability problem for the last iteration in the fixed-point computation (where we detect that we have reached the fixed-point). The word "second_last_fp" indicates the satisfiability problem for the previous iteration in the fixed-point computation. The result column indicates whether the satisfiability problem is satisfiable (S) or not (U).

For the model checking problems, UP_ONE and UP_ALL perform very poorly. UP_ONE is only able to handle four out of 18 problems and UP_ALL only handles a single one. However, both UP_ONE and UP_ALL handle the `10_final` problem which the SAT-solvers are unable to handle. The SAT-solvers perform quite well – both on the satisfiable and the unsatisfiable problems. Most of the problems are solved in less than a second by all three SAT-solvers. While both SATO and GRASP take a long time on a few of the problems, BEDSAT seems to be more consistent in its performance.

6.5 Related Work

The satisfiability problem has been studied for a long time. The Davis-Putnam SAT-procedure [DP60, DLL62] has been known for around 40 years. It is still one of the best procedures for determining satisfiability.

People have also studied incomplete algorithms like Greedy SAT (GSAT) [SLM92]. They are typically fast – often faster than the complete methods. However, they are incomplete which means that they are not always able to come up with an answer.

As mentioned in Chapter 5, Giunchiglia and Sebastiani [GS99, Seb94] have examined GSAT and Davis-Putnam for use on non-CNF formulae. Stålmarck's method also works on a non-CNF representation.

6.6 Conclusion

In this chapter we have presented BEDSAT as an algorithm for solving the satisfiability problem on BEDs. Many traditional SAT-solvers require a CNF formula, but BEDSAT works directly on the BED and is thus able to take advantage of the data structure – for example by using the rewriting rules from Section 3.2.2 during the algorithm. Furthermore, BEDSAT avoids

Description	Result	UP_ONE	UP_ALL	SATO	GRASP	BEDSAT
10_final	U	13.1	43.5	-	-	-
10_last_fp	U	10.3	-	0.1	0.1	0.2
10_second_last_fp	S	-	-	0.1	0.1	0.1
20_final	?	-	-	-	-	-
20_last_fp	U	-	-	0.1	0.1	0.1
20_second_last_fp	S	-	-	0.5	40.9	0.5
30_final	S	-	-	0.3	0.6	0.2
30_last_fp	U	-	-	0.1	0.2	0.2
30_second_last_fp	S	-	-	0.6	1.4	0.5
bug_10_final	S	13.0	-	6.7	0.1	0.1
bug_10_last_fp	U	9.9	-	0.1	0.1	0.2
bug_10_second_last_fp	S	-	-	0.1	0.1	0.1
bug_20_final	S	-	-	113	-	0.3
bug_20_last_fp	U	-	-	0.1	0.1	0.1
bug_20_second_last_fp	S	-	-	0.5	499	0.5
bug_30_final	S	-	-	0.3	0.6	0.2
bug_30_last_fp	U	-	-	0.1	0.2	0.2
bug_30_second_last_fp	S	-	-	0.6	1.5	0.5

Table 6.3: Runtimes in seconds for determining satisfiability of problems arising in model checking of multipliers using different approaches. In the "Result" column, "U" indicates unsatisfiable problems while "S" indicates satisfiable problems. Both UP_ONE and UP_ALL use the FANIN variable ordering heuristic. All methods were limited to 32 MB of memory and 15 minutes of CPU time. A dash indicates that the computation could not be done within the resource limits.

the conversion from BED into CNF which either adds extra variables or the CNF risks blowing up in size.

The experiments with BedSat are quite promising. Especially on examples from model checking the BedSat algorithm performs well. The performance on combinational circuits is not so good. We believe this to be due to the large number of unsatisfiable problems. Implementation of premature backtracking may also help on the performance since BedSat could backtrack out of difficult parts of the state space and either find a satisfying assignment elsewhere or let the rewriting rules and different split sequences handle the more difficult parts.

Chapter 7

Extending Boolean Expression Diagrams

A Boolean Expression Diagram is an extension of a Binary Decision Diagram with operator vertices. The operators in the new vertices are binary Boolean connectives. In this chapter we add other types of operators to the data structure and examine their advantages compared to a data structure without them.

7.1 Introduction

What is needed to add new types of operator vertices to the BED data structure? In Chapter 2 we presented the BED data structure and algorithms for manipulating them. The algorithms for transforming BEDs into BDDs are based on Equation 2.2. The equation is depicted in Figure 2.5. This equation and the truth tables in Table 2.1 for the binary Boolean connectives form the mathematical foundation for the BED to BDD transformation algorithms:

- The truth tables in Table 2.1 show how the binary Boolean connectives handle the terminal cases, i.e., what happens when an operator is applied to constants.
- Equation 2.2 shows how binary Boolean connectives distribute over the *if-then-else* operator.

In order to introduce other operators in the BEDs, we need to know how they behave in the terminal case and how they distribute over *if-then-else*.

The data structure itself poses a requirement for new operators: Only a limited amount of memory is available per vertex. Figure 2.2 shows that there is one variable field *var*, one operator field *op*, and two BED fields *low* and *high* available per vertex. The *low* and *high* fields must point to valid BED vertices as the fields are used in traversal of the BED[1]. The *op* field is an identification of the vertex type. For binary Boolean operator vertices, it contains the connective. For variable vertices it contains a "this is a variable vertex" tag. For new types of vertices the *op* field should contain an identification of the vertex type. The *var* field is an integer attribute. For variable vertices it contains the variable. For operator vertices it is ignored.

The semantics of a BED vertex is a Boolean function. We stick to Boolean functions as the semantics of new types of vertices. The semantics of a vertex should be defined in terms of the three attributes: *low*, *high* and *var*.

7.2 New Types of Vertices

It is natural to extend the BED data structure with operators for quantification and substitution. Quantification, for example, is used extensively in symbolic model checking. In this section we show how to add vertices for such operators.

7.2.1 Existential Quantification

The first new vertex type we introduce is one for existential quantification. We use the following attributes:

var : The variable to be existentially quantified.

low : The function in which the variable is quantified.

high : Not used[2].

An existential quantification vertex v denotes the Boolean function:

$$f^v = \exists var(v) : f^{low(v)} .$$

[1] We could use the fields for other purposes, but that would require more complicated traversal algorithms and we have opted not to do it.

[2] Just like with negation vertices, the *high* attribute is not used. In an implementation, in order to make traversal of the data structure easier, we let *high* have the same value as *low*.

Let x and y be two different variables and let f and g be functions represented by BEDs. The terminal cases for existential quantification are:

$$\begin{aligned} \exists x : 0 &= 0 \\ \exists x : 1 &= 1 \end{aligned}$$

Existential quantification distributes over *if-then-else* in the following way:

$$\begin{aligned} \exists x : x \to f, g &= g \vee f \quad , \quad x \notin sup(f) \cup sup(g) \\ \exists x : y \to f, g &= y \to (\exists x : f), (\exists x : g) \end{aligned} \tag{7.1}$$

The requirement that x is not a free variable of f or g in $\exists x : x \to f, g$ is not a problem because we assume BEDs are always free as per Definition 2.1.4. Since the BEDs for f and g are children of a variable x vertex, then neither f nor g can contain x as a free variable.

The proof of 7.1 is a straightforward application of the naïve quantifier expansion in Equation 5.3:

$$\begin{aligned} & \exists x : x \to f, g \\ = \; & (x \to f, g)_0 \vee (x \to f, g)_1 \\ = \; & (0 \to f, g) \vee (1 \to f, g) \\ = \; & g \vee f \end{aligned}$$

and

$$\begin{aligned} & \exists x : y \to f, g \\ = \; & (y \to f, g)_0 \vee (y \to f, g)_1 \\ = \; & (y \to f_0, g_0) \vee (y \to f_1, g_1) \\ = \; & (y \wedge f_0) \vee (\neg y \wedge g_0) \vee (y \wedge f_1) \vee (\neg y \wedge g_1) \\ = \; & y \wedge (f_0 \vee f_1) \vee \neg y \wedge (g_0 \vee g_1) \\ = \; & y \to (\exists x : f), (\exists x : g) \end{aligned}$$

where a subscript 0 (1) indicates the negative (positive) co-factor with respect to x.

7.2.2 Universal Quantification

The universal quantifier is the dual of the existential quantifier. We add it to the data structure in a similar way. We use the following attributes:

var : The variable to be universally quantified.

low : The function in which the variable is quantified.

high : Not used.

A universal quantification vertex v denotes the Boolean function:

$$f^v = \forall var(v) : f^{low(v)}.$$

Let x and y be two different variables and let f and g be functions represented by BEDs. The terminal cases for universal quantification are:

$$\begin{aligned} \forall x : 0 &= 0 \\ \forall x : 1 &= 1 \end{aligned}$$

Universal quantification distributes over *if-then-else* in the following way:

$$\begin{aligned} \forall x : x \to f, g &= g \wedge f \quad , \quad x \notin sup(f) \cup sup(g) \\ \forall x : y \to f, g &= y \to (\forall x : f), (\forall x : g) \end{aligned} \tag{7.2}$$

The proof of 7.2 follows along the same lines as the corresponding proof for existential quantification.

7.2.3 Substitution

Replacing a variable with another variable or a Boolean function often comes in handy. For example, in Chapter 5 we replaced one set of variables with another set in the calculation of $[\![\mathbf{EX}\ \phi]\!]$. Back then we simply reconstructed the whole BED, but with a new set of variables. This worked fine, although in some cases one might not want to reconstruct the whole BED just with other variables. Consider a BED for $f(x)$, where x is a variable vector. If we want to have BEDs for $f(a)$, $f(b)$ and $f(c)$, where a, b and c are vectors of different variables, then we would have three representations of the same function f. On the BED level there would be no sharing between the three as they use different variables. It would be more memory efficient to have just *one* copy of f, and then have different ways to access f depending on which variable vector was needed. We introduce a substitution vertex to capture this idea.

var : The variable to be substituted.

low : The function in which the substitution takes place.

high : The function which is substituted in place of *var*.

A substitution vertex v denotes the Boolean function:

$$f^v = f^{low(v)}[f^{high(v)}/var(v)]\,.$$

Let x and y be two different variables and let f, g and h be functions represented by BEDs. The terminal cases for substitution are:

$$\begin{aligned} 0[f/x] &= 0 \\ 1[f/x] &= 1 \end{aligned}$$

Substitution distributes over *if-then-else* in the following way:

$$\begin{aligned} (x \to f, g)[h/x] &= (h \wedge f) \vee (\neg h \wedge g) \quad , \quad x \notin sup(f) \cup sup(g) \cup sup(h) \\ (y \to f, g)[h/x] &= y \to f[h/x], g[h/x] \end{aligned} \tag{7.3}$$

The requirement that x is a free variable of f or g holds because we assume BEDs are always free. The requirement that x is free in h is to avoid cycles. Note that $(h \wedge f) \vee (\neg h \wedge g)$ corresponds to $h \to f, g$. However, we write $(h \wedge f) \vee (\neg h \wedge g)$ to indicate that that is way we construct the BED.

The proof of 7.3 is straightforward. The first part is a Shannon expansion of the *if-then-else* operator and then a replacement of h for x. In the second part we push the substitution to the children of a variable y vertex. This is correct as y is not the variable to be substituted.

7.2.4 Operators on Vectors

The operators so far operate on *one* variable: quantification of one variable or substitution of one variable with a function. In many situations it is interesting to perform quantification of a vector of variables or substitutions of a vector of variables with a vector of functions.

Vector Existential Quantification

In the one variable version of existential quantification, we use the *var* field to hold the variable to quantify. Now we need a vector of variables. We represent the vector as a conjunction of the variables. The *high* field contains the BED for the conjunction.

var : Not used.

low : The function in which the variables are quantified.

high : A conjunction of the variable to be quantified. (A conjunction of *zero* elements is 1.)

An existential quantification vertex v denotes the Boolean function:

$$f^v = \exists\, x \in sup(high(v)) : f^{low(v)}\,.$$

Let $\bar{x}$ be a vector of variables. Let x be one of those variable, and let y be a variable *not* in $\bar{x}$. Let f and g be functions represented by BEDs. We use $\emptyset$ to denote the empty set of variables (a vector of dimension zero). The terminal cases for existential quantification are:

$$\begin{aligned} \exists \bar{x} : 0 &= 0 \\ \exists \bar{x} : 1 &= 1 \\ \exists \emptyset : f &= f \end{aligned}$$

Existential quantification distributes over *if-then-else* in the following way:

$$\begin{aligned} \exists \bar{x} : x \to f, g &= \exists \bar{x} \setminus x : g \vee f \quad , \quad x \notin sup(f) \cup sup(g) \\ \exists \bar{x} : y \to f, g &= y \to (\exists \bar{x} : f), (\exists \bar{x} : g) \end{aligned}$$

where $\bar{x} \setminus x$ is the vector of all the variables in $\bar{x}$ excluding the variable x. The BED in the $\exists \bar{x} : x \to f, g$ case looks like this: The top vertex v has $low(v) = x \to f, g$ and $high(v) = x \to h, 0$ [3]. The resulting vertex v' has $low(v') = g \vee f$ and $high(v') = h$. The BED h is a conjunction of the variables $\bar{x} \setminus x$. See Figure 7.1.

Vector Universal Quantification

The vector version of universal quantification resembles the vector version of existential quantification.

var : Not used.

low : The function in which the variables are quantified.

[3] Since x is one of the variables we are quantifying over, we know that $high(v)$ contains x. If this operation takes place inside Up_One or Up_All then $high(v)$ will have a variable x vertex at the top. Otherwise we can bring $high(v)$ in that form by using Up_One to pull x up.

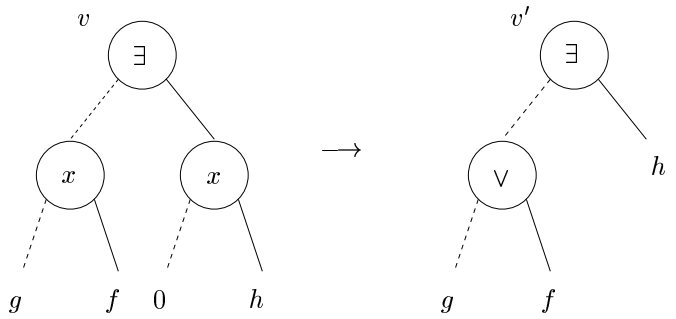

Figure 7.1: The BED v for $\exists \bar{x} : x \to f, g$ and the BED v' for $\exists \bar{x} \setminus x : g \vee f$. The BEDs show how the vector existential quantification operator distributes over *if-then-else*.

high **:** A conjunction of the variable to be quantified. (A conjunction of *zero* elements is 1.)

A universal quantification vertex v denotes the Boolean function:

$$f^v = \forall\, x \in sup(high(v)) : f^{low(v)}\,.$$

Let $\bar{x}$ be a vector of variables. Let x be one of those variable, and let y be a variable *not* in $\bar{x}$. Let f and g be functions represented by BEDs. We use $\emptyset$ to denote the empty set of variables (a vector of dimension zero). The terminal cases for universal quantification are:

$$\begin{aligned} \forall \bar{x} : 0 &= 0 \\ \forall \bar{x} : 1 &= 1 \\ \forall \emptyset : f &= f \end{aligned}$$

Universal quantification distributes over *if-then-else* in the following way:

$$\begin{aligned} \forall \bar{x} : x \to f, g &= \forall \bar{x} \setminus x : g \wedge f \quad , \quad x \notin sup(f) \cup sup(g) \\ \forall \bar{x} : y \to f, g &= y \to (\forall \bar{x} : f), (\forall \bar{x} : g) \end{aligned}$$

Vector Substitution

Vector substitution is a substitution of a vector of functions for a vector variables; each element in the function vector is substituted for the corresponding element in the variable vector. This means we need two vectors and a function in which to perform the substitutions. To fit it all in the data structure we need an auxiliary vertex, which we call *map*.

The map vertex acts a placeholder for a variable / function pair. Multiple map vertices form a list terminated with **0**.

var : A variable.

low : The next map vertex in the list; or **0** if last element in list.

high : A function.

Figure 7.2 shows how to move a map vertex up. We follow the *low* edge to remove the first element in a map list.

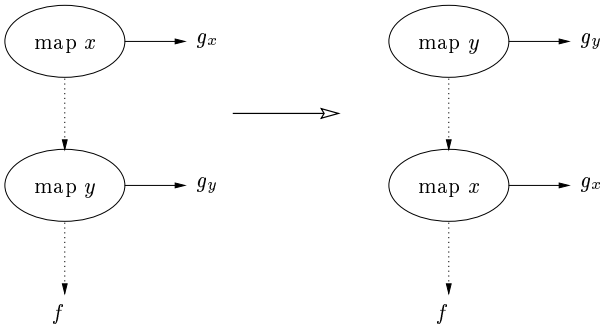

Figure 7.2: The up step for map vertices. The map for variable y is moved above the map for variable x.

A vector substitution vertex has the following attributes:

var : Not used.

low : The function in which the substitution takes place.

high : The map list containing a list of pair of variables and functions.

A substitution vertex v denotes the Boolean function:

$$f^v = f^{low(v)}[\bar{h}/\bar{x}]\,,$$

where $[\bar{h}/\bar{x}]$ is the map of variables $\bar{x}$ to functions $\bar{h}$ in the $high(v)$ map list.

Let $\bar{x}$ be a vector of variables. Let x be one of those variable, and let y be a variable *not* in $\bar{x}$. Let f and g be functions represented by BEDs. We use $\emptyset$ to denote the empty map list. The terminal cases for substitution are:

$$\begin{aligned} 0[\bar{h}/\bar{x}] &= 0 \\ 1[\bar{h}/\bar{x}] &= 1 \\ f[\emptyset] &= f \end{aligned}$$

Substitution distributes over *if-then-else* in the following way:

$$\begin{aligned}(x \to f, g)[\bar{h}/\bar{x}] &= \Big((h_x \wedge f) \vee (\neg h_x \wedge g)\Big)[\bar{h} \setminus h_x \,/\, \bar{x} \setminus x] \\ (y \to f, g)[\bar{h}/\bar{x}] &= y \to f[\bar{h}/\bar{x}], g[\bar{h}/\bar{x}]\end{aligned}$$

where h_x is the function in $\bar{h}$ to be substituted for x, and assuming that in the first equation $x \notin sup(f) \cup sup(g)$.

Vector Existential Quantification and Substitution

In symbolic model checking, we compute $[\![\mathbf{EX}\ \phi]\!]$ as $\exists s', x : T(s, x, s') \wedge [\![\phi]\!][s'/s]$. BDD-based model checkers often have one specialized function for performing the vector quantification and the conjunction in one single step. We now show how to combine the vector quantification and the vector substitution in one new operator type in BEDs[4]. We call the new operator ESUB.

The previous vector operators had explicit vector arguments represented as BEDs. We could actually do the same with ESUB, but we chose not to do so. Instead we illustrate another method. The idea is that we need relatively few different vectors: It is the same set of variables we constantly quantify out, or it is the same vector of variables we substitute with the same vector of functions. We let the vectors be implicitly given.

In the case of ESUB, we quantify out the input variables and the primed state variables. Then the unprimed state variables are replaced with their primed versions. Internally in the BED each variable is assigned a number. Assume that all unprimed state variables are assigned even numbers and the primed state variables are assigned odd numbers. A primed variable has a number one higher than the corresponding unprimed variable. The input variables are assigned odd numbers. This means that odd variables should be quantified and even variables should be replaced by the variable with one higher number.

We use the following fields in an ESUB operator vertex:

var : Not used.

low : The function on which the ESUB operates.

high : Not used.

[4]The idea of combining the two is due to Henrik Reif Andersen.

An ESUB vertex v denotes the Boolean function:

$$f^v = \left(\exists \bar{x}' : f^{low(v)}\right)[\bar{x}'/\bar{x}]\,.$$

where $\bar{x}$ is a vector of even variables and $\bar{x}'$ is a vector of odd variables.

Let x be an even variable and x' the corresponding odd variable. Let f and g be functions represented by BEDs. The terminal cases for ESUB are:

$$\begin{aligned} \text{ESUB}\,0 &= 0 \\ \text{ESUB}\,1 &= 1 \end{aligned} \tag{7.4}$$

ESUB distributes over *if-then-else* in the following way:

$$\begin{aligned} \text{ESUB}\,x' \to f, g &= \text{ESUB}\,g \;\vee\; \text{ESUB}\,f \quad , \quad x' \notin sup(f) \cup sup(g) \\ \text{ESUB}\,x \to f, g &= x' \to \text{ESUB}\,f, \text{ESUB}\,g \end{aligned} \tag{7.5}$$

7.2.5 Implementation

We show how to incorporate the new operators in the Up_One and Up_All algorithms by using the ESUB operator as an example.

The Mk algorithm handles the terminal cases of ESUB in Equation 7.4. We modify Mk in Algorithm 2.3 by adding two extra lines. Algorithm 7.3 shows the pseudo-code for the new version of Mk. Lines (7) and (8) are new. Line (7) tests for the ESUB terminal cases. If we are in such a case then line (8) returns the correct terminal vertex.

Name: Mk(α, l, h)
1: **if** there exists a (α, l, h) vertex **then**
2: **return** that vertex
3: **else if** α is a variable and $l = h$ **then**
4: **return** l
5: **else if** α is a Boolean connective and either l and h are identical or one of them is a terminal **then**
6: **return** either **0**, **1**, l, h, Mk$(\neg, l, \cdot)$ or Mk$(\neg, h, \cdot)$
7: **else if** α is an ESUB operator and l is a terminal **then**
8: **return** l
9: **else**
10: **return** new vertex (α, l, h)

Algorithm 7.3: The Mk algorithm modified to handle the terminal case of ESUB. Line (7) and (8) are the new ones.

Algorithm 7.4 shows the pseudo-code for the modified version of UP_ONE from Algorithm 2.7. The lines (6) through (12) implement Equation 7.5. Algorithm 7.5 shows the pseudo-code for the modified version of UP_ALL from Algorithm 2.9. The lines (8) through (17) implement Equation 7.5. Note how we let the *high* attribute be equal to the *low* attribute when we create ESUB vertices. This way the recursive traversal of the BEDs in the beginning of both UP_ONE and UP_ALL remains unchanged.

Name: UP_ONE$'(x, u)$
Require: The memorization table M is initialized to empty prior to the first call.
1: **if** (x, u) is in M **then return** M(x, u)
2: **if** u is a terminal vertex **then return** (u, u)
3: **if** u is a variable x vertex **then return** $(low(u), high(u))$
4: $(ll, lh) \leftarrow$ UP_ONE$'(x, low(u))$
5: $(hl, hh) \leftarrow$ UP_ONE$'(x, high(u))$
6: **if** u is an ESUB vertex **then**
7: $\quad rl \leftarrow$ MK(ESUB$, ll, ll)$
8: $\quad rh \leftarrow$ MK(ESUB$, lh, lh)$
9: $\quad$ **if** x is even **then**
10: $\quad\quad rl \leftarrow rh \leftarrow$ MK$(x + 1, rl, rh)$
11: $\quad$ **else**
12: $\quad\quad rl \leftarrow rh \leftarrow$ MK$(\vee, rl, rh)$
13: **else**
14: $\quad rl \leftarrow$ MK$(\alpha(u), ll, hl)$
15: $\quad rh \leftarrow$ MK$(\alpha(u), lh, hh)$
16: insert $((x, u), (rl, rh))$ in M
17: **return** (rl, rh)

Algorithm 7.4: The UP_ONE$'$ algorithm modified to handle ESUB. The new lines are (6) through (12).

7.3 Fault Tree Analysis

In this section, we propose an algorithm based on BEDs for computing the *minimal p-cuts* of Boolean reliability models such as fault trees. BEDs make it possible to bypass the BDD construction, which is the main cost of fault tree assessment. This section is based on the paper [WNR00].

We consider Boolean formulae built over a set of variables $X = \{x_1, \ldots, x_n\}$,

Name: UP_ALL(u)
Require: The memorization table M is initialized to empty prior to the first call.

```
 1: if u is in M then return M(u)
 2: if u is a terminal vertex then return u
 3: (l, h) ← (UP_ALL(low(u)), UP_ALL(high(u)))
 4: if l and h are terminal vertices then
 5:    r ← MK(α(u), l, h)
 6: else if u is a variable x vertex then
 7:    r ← MK(x, l, h)
 8: else if u is an ESUB vertex then
 9:    if l is a terminal then
10:       r ← l
11:    else
12:       rl ← UP_ALL(MK(ESUB, low(l), low(l)))
13:       rh ← UP_ALL(MK(ESUB, high(l), high(l)))
14:       if x is even then
15:          r ← MK(x + 1, rl, rh)
16:       else
17:          r ← UP_ALL(MK(∨, rl, rh))
18: else if var(l) = var(h) then
19:    tmp1 ← UP_ALL(MK(α(u), low(l), low(h)))
20:    tmp2 ← UP_ALL(MK(α(u), high(l), high(h)))
21:    r ← MK(var(l), tmp1, tmp2)
22: else if var(l) < var(h) then
23:    tmp1 ← UP_ALL(MK(α(u), low(l), h))
24:    tmp2 ← UP_ALL(MK(α(u), high(l), h))
25:    r ← MK(var(l), tmp1, tmp2)
26: else
27:    tmp1 ← UP_ALL(MK(α(u), l, low(h)))
28:    tmp2 ← UP_ALL(MK(α(u), l, high(h)))
29:    r ← MK(var(h), tmp1, tmp2)
30: insert (u, r) in M
31: return r
```

Algorithm 7.5: The UP_ALL algorithm modified to handle the ESUB operator. The new lines are (8) through (17).

the two constants $0, 1 \in \mathbb{B}$ and the usual operators $\wedge$, $\vee$, $\neg$, $x \rightarrow y, z$ (if-then-else) etc. A *literal* is either a variable x or its negation $\neg x$. A *product* is a set of literals that does not contain a literal and its negation. A product is assimilated with the conjunction of its elements. A minterm over X is a product that contains either positively or negatively all variables of X.

Let f be a formula and π be a product that contains only positive literals. We denote by π_X^c the minterm obtained by adding to π the negative literals formed over all of the variables occurring in f but not in π. π is a *p-cut* of f if $\pi_X^c \models f$. It is minimal if there is no product $\delta \subset \pi$ such that $\delta_X^c \models f$. We denote by $\Pi(f)$ the set of the minimal p-cuts of the formula f and $\Pi_k(f)$ the set of the minimal p-cuts with at most k literals.

Two decomposition theorems [DR97] allow the design of algorithms to compute the ZBDD [Min93] that encodes $\Pi_k(f)$ from the BDD that encodes f. Indeed, computing the former requires computing the latter. However, only part of the BDD is used, since some of the products it encodes are useless to the computation of $\Pi_k(f)$ [DR98]. We show that BEDs make it possible to compute only relevant parts of the BDD, therefore avoid a potential exponential blow-up.

Minimal p-cuts play a central role in the assessment of fault trees. Boolean formulae describe the potential failures of the system under study; variables represent component failures. Minimal p-cuts represent minimal sets of component failures that induce a failure of the whole system. This notion should be preferred to the classical notion of prime implicants[5] that also captures the idea of minimal solutions [DR97, DR98].

Positive literals represent failures of the individual components. The failure probabilities are assumed to be independent. It is the failures that are of practical interest. The failure probability of each literal is generally quite low and thus products with a large positive part represent a negligible probability. Therefore, it is in general a safe approximation to consider only products with very few positive literals.

Minimal p-cuts approximate prime implicants by considering only positive parts of implicants, and k-truncated minimal p-cuts restrict the result to those of size at most k. The latter is of practical importance in qualitative analysis of fault trees, as it identifies sets of component with high probability of simultaneous failure that would cause the entire system to fail. To determine whether there exists a prime implicant of length k or less is a ΔP complete problem [Pap94]. Therefore, unless NP=coNP=P, there

[5] An implicant σ for function f is a product over the variables in f such that $\sigma \models f$. The implicant σ is prime if it has the property that for all shorter implicants $\rho \subset \sigma$, $\rho \not\models f$.

do not exist efficient (i.e. polynomial) algorithms to compute short prime implicants. However, such algorithms do exist for minimal p-cuts [DR98] and are illustrated here. These algorithms are based on the following theorems. The first one establishes that minterms with more than k positive literals are useless for computing $\Pi_k(f)$. The second theorem gives a recursive principle for computing $\Pi_k(f)$ from the Shannon decomposition of f.

Theorem 7.3.1 (Dutuit & Rauzy [DR98]). *Let f be a Boolean formula over the set of variables X and k be a integer, then the following equality holds:*

$$\Pi_k(f) = \Pi_\infty(f \cap minterms_k^+(X)),$$

where $minterms_k^+(X)$ denotes the minterms built over X that contain at most k positive literals and f is viewed as the set of minterms that satisfy it.

Theorem 7.3.2 (Dutuit & Rauzy [DR97]). *Let $f = x \to f_1, f_0$ be a Boolean formula with f_1 and f_0 not depending on x. Then, $\Pi_k(f)$ can be obtained as the union of two sets $\Pi_k(f) = v.\Pi_1 \cup \Pi_0$ where $\Pi_0 = \Pi_k(f_0)$, $\Pi_1 = \Pi_{k-1}(f_1 \vee f_0) \setminus \Pi_0$, $v.P = \{v \wedge \pi \mid \pi \in P\}$ and $\setminus$ denotes set difference.*

We exploit this fact not only to compute $\Pi_k(f)$ incrementally, but to expand the formula f into a BDD incrementally. This is possible using the BED data structure.

7.3.1 Minimal P-Cuts with BEDs

It is clear that minimal p-cuts can be computed using BEDs, since it is sufficient to convert the BED for f to a BDD using Up_All, then to apply the standard algorithm from [DR97]. The disadvantage is that we construct the entire BDD for the f, when only part of this information is necessary for computing the p-cuts (Theorem 7.3.1). The Up_One transformation gives us finer control over the conversion of the BED to an BDD. We show that minimal p-cuts can be computed by a bottom-up expansion of the formula that only converts what is necessary for the computation. In practice, the resulting algorithm often does less work than the standard algorithm.

We extend the BED data structure with a new kind of unary operator node, `PC`, which marks the frontier between a Boolean formula and its p-cuts. Nodes above this frontier represent the BDD encoding the p-cuts for the formula f as the disjunction of the minterms π_X^c, where π is a k-truncated

minimal p-cut of f. In each step of the new algorithm, the UP_ONE transformation lifts the smallest variable in a set L over any Boolean operators or other variable nodes, until it reaches a PC operator. A p-cut vertex v denotes a Boolean function which encodes the set of k-truncated p-cuts for the function f over the variables in a set L. We write $\texttt{PC}(f)[k; L]$ to indicate such a vertex, and we use the following attributes:

var : The value k.

low : The function f

high : A conjunction of the variables in L. (A conjunction of zero variables is the vertex **1**.)

In accordance with Theorem 7.3.2, the terminal cases for PC are:

$$\begin{aligned} \texttt{PC}(0)[k; L] &= 0 \\ \texttt{PC}(1)[k; \emptyset] &= 1 \\ \texttt{PC}(1)[k; x.L] &= \neg x \wedge \texttt{PC}(1)[k; L] \end{aligned}$$

where $x.L$ denotes the union of the sets $\{x\}$ and L ($x \notin L$). PC distributes over *if-then-else* in the following way:

$$\begin{aligned} \texttt{PC}(x \to f, g)[0; x.L] &= \neg x \wedge \texttt{PC}(g)[0; L] \\ \texttt{PC}(x \to f, g)[k; x.L] &= x \to S, T && (k > 0) \\ & \quad T = \texttt{PC}(g)[k; L] \\ & \quad S = \texttt{PC}(f \vee g)[k-1; L] \wedge \neg T \\ \texttt{PC}(x \to f, g)[k; y.L] &= \neg y \wedge \texttt{PC}(x \to f, g)[k; L] && (x \notin L) \end{aligned}$$

To calculate the minimal truncated p-cuts we use either UP_ALL (corresponding to the standard algorithm) or UP_ONE. Figure 7.6 shows how the PC operators "drive" the computation, pulling BED variables up to the frontier. The process is started by seeding a PC operator at the root of the original formula. As long as there are variable nodes below a PC operator, we pull them up one by one from the set L, until either no variables remain or the PC nodes in the frontier exhaust their capacity ($k = 0$).

The number of minterms in π^c_X for a k-truncated p-cut π is equal to $\sum_{i=0}^{k} \binom{n}{i}$, where n is the number of variables in X. This number is bounded by $O(n^k)$. Each k-truncated p-cut uses at most k variable vertices. Thus $O(n^k)$ also bounds the number of BDD vertices needed to represent the p-cuts.

Proposition 7.3.3. *The number of BDD vertices created to encode the k-truncated p-cuts is bounded by $O(n^k)$.*

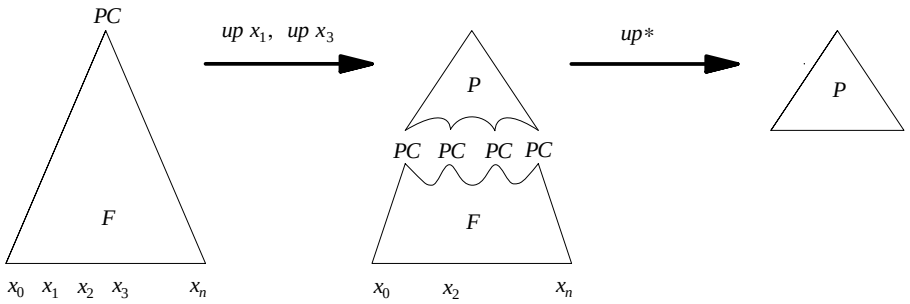

Figure 7.6: Computation of p-cuts.

7.4 Experimental Results

In this section we give experimental results for the use of substitution and p-cut vertices.

7.4.1 Substitution

Integer multipliers are notoriously difficult to represent using BDDs as the representation is always at least exponential in the size of the multiplier [Bry86]. The BDD representation of a 15-bit multiplier uses more than 12 million vertices [OYY93] and around 40 million vertices for a 16-bit multiplier [YCBO98]. The number of vertices grows exponentially with the number of bits in the operands.

The ISCAS'85 benchmark suite contains two 16-bit multiplier circuits, `c6288` and `c6288nr`. Using four instances of one of the multipliers and some extra addition network, we can build a 32-bit multiplier. This is shown in Figure 7.7. The idea is to express a 32-bit multiplication as four 16-bit multiplications. Let $x = x_1x_2$ and $y = y_1y_2$ be two 32-bit numbers where x_1 and y_1 represent the 16 most significant bits and x_2 and y_2 the 16 least significant bits. A 32-bit multiplication of x and y can be done by use of 16-bit multiplications of x_1, x_2, y_1, and y_2 in the following way:

$$z = x_1x_2 *_{32} y_1y_2 = (x_1 *_{16} y_1) * (2^{16})^2 + (x_1 *_{16} y_2 + x_2 *_{16} y_1) * 2^{16} + x_2 *_{16} y_2$$

Here $*_n$ represents n-bit multiplication while $*$ alone represents a multiplication that can be done by bit shifting.

Through the use of substitution, we can create the 32-bit multiplier using only *one* instance of the 16-bit multiplier; see Figure 7.8. Using substitution we create pairs of n-bit multipliers of increasing size. We can verify that

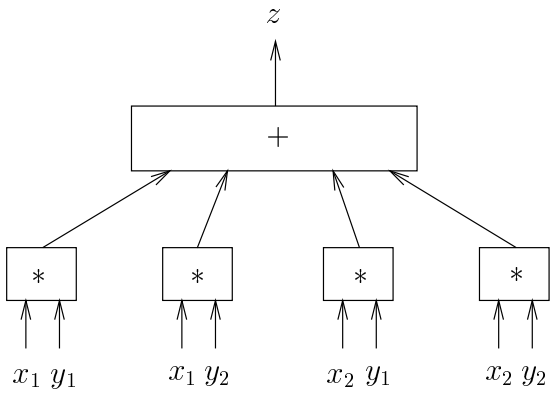

Figure 7.7: A $2n$-bit multiplier created from four n-bit multipliers.

each pair of multipliers are equivalent. Table 7.1 shows the results for the verification of large n-bit multipliers built this may. We are able to verify pairs of multipliers up to 1024-bit to be equivalent.

The original 16-bit multipliers are verified to be equivalent by pulling two variables (256gat and 290gat) up to the root. Each of the pairs of larger multipliers is verified by pulling the same two variables up to the root. No other knowledge of internal vertices is required.

It is also possible to build and verify the multipliers by using multiple instances of `c6288-c6288nr` instead of using substitution, but it is more complicated. Pulling the two variables 256gat and 290gat up to the root works for one of the instances of `c6288-c6288nr` but not for all of the other ones. This is because the inputs to the different instances are not identical. The right two variables to pull up to the root for one of the 16-bit multipliers would be the wrong two variables for another of the multipliers. We get exponential growth. It is possible to lift the right variables up to the top of each of the instances. It requires knowledge of internal vertices in the BED. One 1024-bit multiplier contains 4096 instances of a 16-bit multiplier. Verifying the two 1024-bit multipliers to be equivalent would require pulling the correct two variables up in 4096 sub-BEDs. The example shows that using substitution is an advantage.

7.4.2 P-Cut

We test our p-cut method experimentally on three fault trees: `cea9601`, `das9601`, and `wes9701`. They are from CEA (French Military), Dassault Aviation (French aviation company), and Westinghouse (American nuclear

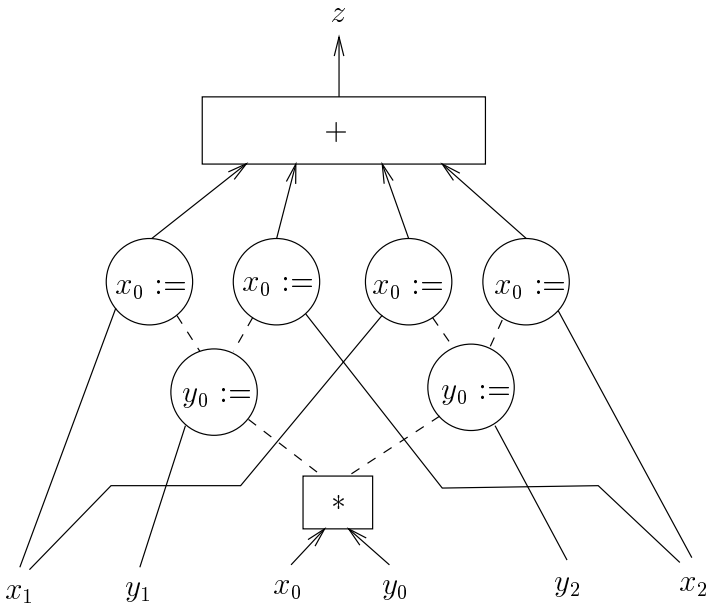

Figure 7.8: A $2n$-bit multiplier created from one n-bit multiplier using substitution. x_0 and y_0 are the formal parameters for the multiplication cell which are replaced with the actual parameters x_1, x_2, y_1, and y_2.

n	Ops	Subst	N_{total}	CPU sec
32	6 505	8 192	26 665	1.6
64	9 827	40 960	75 070	3.1
128	16 447	172 032	270 275	8.6
256	29 783	696 320	$1.05 \cdot 10^6$	29.5
512	56 401	2 793 472	$4.19 \cdot 10^6$	119
1024	109 643	11 192 080	$14.3 \cdot 10^6$	408

Table 7.1: Verifying equivalence between two n-bit integer multipliers; one based on `c6288` and one based on `c6288nr` from the ISCAS'85 benchmarks. Column Ops shows the number of operator vertices in the BED and column Subst shows the number of substitutions. N_{total} is the total number of vertices used in the verification. CPU is the verification time measured in seconds on a Sun UltraSPARC 1.

industry), respectively. All our experiments are run on a 500 MHz Digital Alpha.

Table 7.2 shows the number of p-cuts of order 1, 2, 3 and 4 for the three fault trees as well as the runtimes in seconds to find a BDD representation for the p-cuts using UP_ONE. For these calculations, the size of the BED data structure never exceeded 20 MB of memory.

	No. of p-cuts				Runtime [sec]			
Name	1	2	3	4	1	2	3	4
cea9601	0	0	1144	2024	2	27	122	683
das9601	0	47	80	446	1	2	11	75
wes9701	2	211	1079	54436	6	26	151	2000

Table 7.2: Number of p-cuts of order 1, 2, 3 and 4, and running times in seconds to compute them using the UP_ONE transformation.

These results should be compared with the standard method (the UP_ALL algorithm), which is unable to calculate the p-cuts for `cea9601` and `wes9701`. The former could not be build using 300 MB while the latter could not be build in 48 hours. For `das9601` it succeeds in building the BDD for the fault tree in about 2 hours. The variable orderings used in the experiments are the ones given in the anonymous data files. The standard method depends on the variable ordering, and using improved heuristics to determine a good initial variable ordering will definitely improve the performance. However, the UP_ONE method will also benefit from the use of an improved variable ordering heuristic.

7.5 Related Work

Extending binary decision diagrams with operators has been done by other researchers. Section 2.4 mentions some of the new data structures. The most common operators are the Boolean connectives. Jeong *et. al.* use existential and universal quantifiers in their XBDDs [JPHS91]. Hett, Drechsler, and Becker add existential quantifiers to obtain a new method for BDD construction [HDB96, HDB97].

Most BDD implementations have algorithms for doing substitution and quantification. In this chapter we have added the same functionality to BEDs using new operators. One way of thinking of the operators is as a lazy evaluation: The operators are placed in the data structure but they are first expanded or evaluated when it is needed.

The idea of giving the operators vectors as arguments is also known from standard BDDs. Most BDD implementations have quantification and substitution algorithms which allow substitution and quantification of vectors.

The ESUB operator combines two different operations in the computation of $[\![\mathbf{EX}\ \phi]\!]$: the existential quantification and the substitution. In BDD-based model checking, existential quantification and APPLY are often combined. Combinations of this kind are very effective compared to performing the operations one by one.

We refer to the papers [DR97] and [DR98] for a detailed description of p-cuts. The papers are also a good starting point for readers interested in Boolean reliability models and fault trees.

7.6 Conclusion

In this chapter we have explained how to extend the Boolean Expression Diagram data structure with new types of vertices. We have given examples of existential and universal quantification vertices and of substitution vertices. Furthermore, we have presented an operator vertex, ESUB, combining existential quantification and substitution.

We have shown what properties an operator must have in order to be implemented as a vertex type in the BEDs. The properties are not very strict: The operator must have a terminal case and it must distribute in some form over *if-then-else*. Furthermore, only a limited number of attributes are available per vertex. An operator with these properties can be implemented in the BEDs by minor modifications to the MK, UP_ONE and UP_ALL algorithms.

As a more detailed example of a new type of vertex in the BEDs, we have chosen the `PC` operator. Based on this operator we proposed a new method to compute minimal truncated p-cuts. It makes it possible to compute minimal truncated p-cuts directly from the BED without ever constructing the BDD representation of the fault tree (that is often of gigabyte size). The experimental results show that our method has an advantage over the BDD methods.

Chapter 8

Conclusion

In this thesis we have examined the use of the Boolean Expression Diagram data structure in the area of formal verification. We have selected a number of different problems domains for investigation.

The first problem domain was verification of combinational circuits. We considered both circuits described in a flat way and circuits described in a hierarchical way.

From a flat description of two combinational circuits we obtained a Boolean formula which expressed the equivalence of the circuits. Using BEDs we decided whether the formula is a tautology (the two circuits are equivalent) or not a tautology (the two circuits are not equivalent). We examined the following methods for proving or disproving tautology:

Up_All : Similar to a standard BDD approach. The BED is converted to a BDD from the bottom up. The rewriting rules cannot be used during UP_ALL.

Up_One : Construction of a BDD by top down conversion of a BED. The rewriting rules are important for the performance of UP_ONE.

Stålmarck : Reasoning-based method. As opposed to the other two methods, Stålmarck's method does not convert or change the formula it works on. In terms of speed, Stålmarck's method does not work well on circuits. However, it uses little memory.

We have experimented with variable ordering heuristics. Both the FANIN and the DEPTH_FANOUT heuristics give good results.

Based on our research and experiments, we see the following uses for BEDs in combinational circuit verification:

- Use BEDs as BDDs with UP_ALL. The rewriting rules form a preprocessing step which helps speed up the verification by reducing the size of the initial BED. Especially the combination of UP_ALL, rewriting rules and the FANIN heuristic gives good results. Since the rewriting rules are used only as a preprocessing step, all BDD specific techniques can be used in the construction of the BDD.

- Use UP_ONE to convert BEDs to BDDs. This method works best if the two circuits being compared have a high degree of structural similarity. For example, the two 16-bit multipliers in the ISCAS'85 benchmark suite are easily verified using UP_ONE.

Circuits described in a hierarchical way may be converted to flat circuits and then verified by the techniques above. However, we have focused on exploiting the structural information in the hierarchical circuit descriptions. The main idea is to reuse previously calculated results.

Our method works best if the two circuits being compared have similar hierarchical structures. The advantage is that instead of working with representations of the functionality of the circuits, we work with representations of the relation between the circuits. We call such a relation between the circuits for a cut-relation. In some cases, the cut-relation has a simpler BED/BDD representation than the functionality of the circuits. In cases where the hierarchical structure of the circuits are quite similar, the method performs well. For example, we have successfully verified large adder and multiplier circuits using this technique. Unfortunately, if the circuits have dissimilar hierarchical structures, then the cut-relation may become complex and the performance of our method degrades.

The second problem domain is symbolic model checking. We have presented a method for CTL model checking based on fixed-point iterations. We use quantification by substitution for the quantification whenever possible. Quantification by substitution works well with BEDs. Quantification of all state variables can be done in just one traversal of the BED. However, we still need to quantify out the inputs. For this purpose we use scope reduction rules to press the quantifiers as far down as possible in the formulae. Then we perform the quantification using UP_ONE.

Symbolic model checking is typically done using the BDD data structure where equivalence and satisfiability checking are constant time operations. Other people have tried using SAT-solvers. In our method we combine BDDs and SAT-solvers.

Our model checking method works best on examples that can be modeled

with few or no input variables. In such examples we can fully exploit quantification by substitution, and we are able to achieve results superior to those obtained by standard BDD model checking tools. A feature of our method is that is it good at detecting errors.

Systems with input variables is a problem as we are not able to use quantification by substitution to quantifying out the input variables. We propose lfp-CTL as a means of model checking such systems. The lfp-CTL logic is weaker than CTL. However, the nature of the lfp-CTL logic is such that we (often) do not need to quantify out the inputs.

The third problem domain is fault tree analysis. We have chosen to focus on the computation of *p-cuts*. A p-cut is a representation of the most likely reasons for system failure. We have extended the BED data structure to facilitate p-cut computation. Using UP_ONE, we are able to gradually transform a BED for a fault tree to a BDD for the p-cuts.

Our p-cut algorithm utilizes the fact that not all the information in a fault tree is necessary to find the p-cuts. The standard method is to construct the BDD for the fault tree and then compute the p-cuts. However, the BDDs are often huge and in many cases it is not possible to construct them. Our method avoids the BDD construction by only concentrating on the parts of the fault tree which contribute to the p-cuts.

The algorithm works best for short p-cuts. Because of complexity reasons it is not well-suited for longer p-cuts. However, this may not be a serious limitation since short p-cuts are the ones of practical interest.

We have proposed a method for solving the satisfiability problem based on BEDs. The algorithm BEDSAT uses splitting on variables to divide the problem into smaller pieces. The splits are done using UP_ONE, which allows us to take advantage of the rewriting rules during satisfiability checking. We have compared a simple implementation of BEDSAT with state-of-the-art SAT-solvers. On satisfiability problems from model checking, BEDSAT performs well.

We have discussed what is needed for extending the BED data structure. The p-cut computation method is an example of such an extension. We have introduced other extensions, e.g., for quantification and substitution.

8.1 Future Directions

In this dissertation we have explored ways for doing formal verification using Boolean Expression Diagrams. However, we have by no means covered everything. There are still many paths to follow and directions to go. The

following is a list of some of the topics we think it would beneficial to explore further:

- Combining UP_ONE and UP_ALL. We have explored UP_ONE and UP_ALL independently. However, it is possible to combine them. We see two main ways to do this:
  - Use UP_ONE first on a number of variables. Then switch to UP_ALL to finish the BED to BDD conversion. We have suggested UP_ONE-minimizing in Section 3.2.3. This may be a starting point for further research.
  - Use UP, which works as a mix of UP_ONE and UP_ALL. Since the size of BDDs is quite sensitive to how far closely related variables are from each other in an ordering, it may be possible to keep the size down by pulling closely related variables up together.
- Our proposed SAT-procedure BEDSAT does not utilize premature backtracking. As a result it sometimes gets stuck – especially when working on unsatisfiable problem instances. We expect that the addition of premature backtracking to BEDSAT will make it more robust. It would also be interesting to compare BEDSAT with the SAT-solvers proposed by Giunchiglia and Sebastiani in [Seb94, GS99]. Their methods also work on non-clausal formulae.
- Characterizing the CNF formulae produced by BED to CNF conversion and tune the SAT-solvers for such formulae. This has been done by Shtrichman [Sht00] for Bounded Model Checking.
- Detect when we are getting close to a fixed-point. After each iteration in the fixed-point calculation we need to determine whether we have reached the fixed-point or we should continue with another iteration. At the moment we convert the BEDs to either CNF or BDDs. However, it might be worth using SAT-solvers (CNF conversion) in the beginning and then switch to BDD conversion near the fixed-point. This would require some metric which tells us how close we are to the fixed-point. One possible metric is the number of states in the set difference between two successive fixed-point approximations in the fixed-point iteration. This metric usually follows a bell-shaped curve: starts out low, increases, peaks, decreases. Based on such a metric we could skip some of the termination checks in the fixed-point algorithms.

Appendix A

The BED Tool

NAME `bed` — tool for manipulating Boolean formulae as Boolean Expression Diagrams.

SYNOPSIS

`bed` [`-h`] [`-b` *m*] [`-c` *n*] [`-f` *script-file*] [*filename*]

DESCRIPTION

`bed` is a program which allows the user to manipulate Boolean formulae represented as Boolean Expression Diagrams (BEDs). A BED is a generalization of a Binary Decision Diagram (BDD) which can represent any Boolean circuit in linear space and still maintain many of the desirable properties of BDDs. This BED package contains a number of algorithms for transforming a BED into a reduced ordered BDD. One (called Up_All) closely mimics the BDD apply-operator. Another (called Up_One) can exploit the structural information of a Boolean circuit.

AVAILABILITY

`bed` can be obtained from the World Wide Web at
`http://www.it-c.dk/research/bed/`

OPTIONS

Options may appear in any order as long as they appear before the filename.

`-h` Print a short help message.

`-b` Reserve m megabytes of memory for the BED data structure.

`-c` Reserve n megabytes of memory for a cache.

`-f` Execute the semicolon-separated commands in the *script-file*.

filename

Formula description, see file format below.

COMMANDS

The `bed` tool is used to manipulate Boolean formulae represented by the BED data structure. One typical use of it is to transform a BED into a (reduced and ordered) Binary Decision Diagram (BDD). There are different ways to do this: by using *upall* which mimics the standard BDD *apply*-call, by using *upone* to lift the variables up to the outputs one at a time, or by using *upsome* which lefts a set of variables to the outputs. Another usage is to determine satisfiability of a function represented by a BED. This can be done with the *bedsat* command. A number of other commands are available to obtain information about the BED data structure. All available commands a listed below. The syntax of the arguments is described after each command.

`upall` *output-list*

Lifts all the variables up over the operators thereby eliminating the operators and transforming the BED into a BDD. The resulting BDD is both reduced and ordered. The ordering used is the order in which the user has entered the variables in the circuit description file.

`upone` *input-list output-list*

Takes each variable at a time (starting from the first one) from the *input-list* and lifts it up in each of the BEDs rooted by the nodes in the *output-list*. A variable is either lifted up until it reaches the top or until it reaches a node containing a variable previously lifted by the same `upone` command.

`upsome` *input-list output-list*

Lifts the variables in *input-list* up in each of the BEDs rooted by the nodes in the *output-list*.

`anysat` *node*

Returns a satisfying assignment for Boolean function represented by *node*. An assignment is a list of inputs which are true. All other inputs are false. The command only works if the BED is transformed into a BDD first.

`anynonsat` *node*

Returns a non-satisfying assignment for Boolean function represented by *node*. This only works if the BED is transformed into a BDD first.

`satcount` *node*

Returns the number of satisfying assignments for Boolean function represented by *node*. This only works if the BED is transformed into a BDD first.

`eval` *node assignment-list*

Evaluates the Boolean function of a node given the list of input assignments *assignment-list*. *assignment-list* is a list of inputs enclosed in '`[`' and '`]`'. Inputs in the list are set to be true. All other inputs are false.

`bedsat` *output-list*

Determines whether each node represents a satisfiable function (result is **1**), or it is unsatisfiable (result is **0**). The variable *sattime* determines the maximum CPU time used per node.

`addinput` *input-list*

Adds one or more new inputs to the BED. This is useful when one wants to change the BED interactively.

`let` ID `=` *expr*

Creates a new output defined by the Boolean expression *expr*. This is useful when one wants to change the BED interactively. *expr* is a Boolean expression with the following syntax

$$
\begin{array}{rcl}
\mathit{expr} & \rightarrow & \mathrm{ID} \\
& | & (\mathit{expr}) \\
& | & \mathit{expr}\ \mathit{binop}\ \mathit{expr} \\
& | & \mathit{expr}\ <\mathit{var}>\ \mathit{expr} \\
& | & \texttt{not}\ \mathit{expr} \\
& | & \texttt{exists}\ \mathrm{ID}\ \texttt{.}\ \mathit{expr} \\
& | & \texttt{forall}\ \mathrm{ID}\ \texttt{.}\ \mathit{expr} \\
& | & \mathit{expr}\ \texttt{[}\ \mathrm{ID}\ \texttt{:=}\ \mathit{expr}\ \texttt{]} \\
\mathit{binop} & \rightarrow & \texttt{and} \mid \texttt{or} \mid \texttt{biimp} \mid \texttt{nand} \mid \texttt{nor} \mid \texttt{xor} \\
& & \mid \texttt{imp} \mid \texttt{limp} \mid \texttt{nimp} \mid \texttt{nlimp}
\end{array}
$$

The operators bind as you would expect. `imp` is the implication operator, `limp` is left implication, `nimp` is negated implication, and `nlimp` is negated left implication. The expression $\mathit{expr}_1\ <\mathit{var}>\ \mathit{expr}_2$ is the *if-then-else* operator where expr_1 denotes the low-child and expr_2 denotes the high-child.

`unlet` *output*

Removes the *output* from the BED.

`gc` Garbage collection of unreachable nodes. This is also done automatically whenever `bed` is running out of BED nodes.

`set` *option value*

Sets the *option* to *value*. There are the following options:

`support`

value is either `left` or `right`. This option controls whether the *support* command make a left-first (low-first) or right-first (high-first) traversal of the BED. Default is `left`.

`reductions`

value is either `on` or `off`. This option controls whether to use the rewriting rules for the BEDs. Default is *on*.

`dot_zero`
: *value* is either `on` or `off`. This option controls whether edges going to **0** are printed with the *dot* command. Default is *on*.

`dot_numbers`
: *value* is either `on` or `off`. This option controls whether vertex identifiers (numbers) are printed with the *dot* command. Default is *on*.

`dot_ranksep, dot_nodesep, dot_margin,`

`dot_fontsize, dot_ratio`
: Parameters passed to the DOT program. See the manual for DOT for more information.

`sort_roots`
: *value* is either `on` or `off`. This option controls whether the outputs (roots) are presented alphabetically to the user. Default is *on*.

`sattime`
: The maximum CPU time in seconds for each root with *bedsat* command. Zero indicates no limit. Default is 0 seconds.

`displaytime`
: The CPU time of each command is printed whenever the command takes longer than *value* CPU seconds. Default is 10 seconds.

`bedsize`
: *value* is the size in megabytes for the internal representation of the BED data structure. Default is *3.81* corresponding to 200000 vertices.

`cachesize`
: *value* is the size in megabytes for the internal cache. Default is *0.53* corresponding to 20011 entries.

`dot` *node-list* [> *filename*]
: Outputs the BEDs rooted at nodes in *node-list* to the file *filename* in the DOT format. DOT is a graph-drawing program from AT&T which can be obtained from the web at

`http://www.research.att.com/sw/tools/graphviz/`

`read` *filename*

Reads a file containing a description of Boolean formulae in the format described below. Any previously read circuit is discarded.

`write` [*filename*]

Writes the BED to the file *filename* or to `TMP.bed` if no filename is specified.

`source` *filename*

Reads and executes a script of commands. The commands must be separated by semicolons.

`dump` *node* `-` *node*

Prints all nodes between the two *node*s.

`dumpnode` *node*

Prints *node* and all nodes reachable from it.

`inputs`

Prints a list of all the inputs in the BED.

`outputs`

Prints a list of all the outputs in the BED.

`foreach` *root* `do "`*command*`"`

Iterates through all roots in the BED. For each root, the *command* is executed with *root* replaced by the current root. Multiple commands can be separated by semicolons.

`stat` [`bed` | `hash` | `outputs`]

Prints statistical information about either the BED data structure, the hashing or the outputs. Default is bed.

`support` *node*

Returns the support of the Boolean function represented by *node*.

`cd, pwd, ls`

The standard UNIX commands.

`exit` | `quit`

In interactive mode these commands exit the program. In script mode the script is terminated and the program goes into interactive mode.

`halt` Quits the program both in interactive mode and in script mode.

`help` [*command*]
An online help function. Returns a short help for the *command* or, if no argument is specified, an overview of the available commands.

The command arguments:

node is a name of an output node or a number of an internal node.

node-list
is either a single *node*, a list of *node*s enclosed in '`[`' and '`]`' and separated by spaces, or `*` denoting a list of all outputs.

output-list
is either a single output name, a list of output names enclosed in '`[`' and '`]`' and separated by spaces, or `*` denoting a list of all outputs.

input is the name of an input node (a variable name).

input-list
is either one *input*, a list of *input*s enclosed in '`[`' and '`]`' and separated by spaces, or a command returning a list of inputs. The commands available are either `*` denoting a list of all inputs (variables), `support(` *node* `)` which returns the support of *node*, `fanin(` *node* `)` which returns the FANIN ordering of inputs of *node*, and finally `fanout(` *node* `)` which returns the DEPTH_FANOUT ordering of inputs of *node*.

BED FILE FORMAT

The `bed` tool supports a simple file format for storing BEDs. The format is meant to be computer-friendly, not easy to use for humans.

A file in this format consists of three parts: inputs, assignments, and outputs. The input and output parts specify the

inputs and outputs of the formulae. The assignments specify vertices in the BED. The syntax for the file format is:

BED file	→	*inputs assignments outputs*
inputs	→	`inputs` *input-list*
input-list	→	*input input-list* | *input*
input	→	ID
assignments	→	`assign` *assign-list*
assign-list	→	*assign assign-list* | *assign*
assign	→	*var-assign* | *op-assign* | *misc-assign*
var-assign	→	NO ID `ite` NO NO
op-assign	→	NO `-` *op* NO NO
misc-assign	→	NO ID *misc-op* NO NO
op	→	`not` | `and` | `biimp` | `nand` | `nor` | `or` | `xor` | `imp` | `limp` |`nimp` | `nlimp` | `ESUB`
misc-op	→	`:=` | `?` | `!`
outputs	→	`outputs` *output-list*
output-list	→	ID NO *output-list* | ID NO

An ID is a series of one or more letters, numbers, and underscores, or any string in single quotes. A NO is a non-negative integer.

Each *assign* defines a BED vertex. The first NO is a unique identifier for the vertex. A vertex must be defined earlier in the file than a vertex which refers to it. The terminal vertices **0** and **1** are implicitly defined and have unique identifiers 0 and 1, respectively.

var-assign represents a variable vertex. The first NO is a unique vertex identifier. The ID is the variable name. The last two NO's are the identifiers of the *low* and *high* children.

op-assign represents an operator vertex. The first NO is a

unique vertex identifier. The last two NO's are the identifiers of the *low* and *high* children.

misc-assign represents a vertex in the extended version of BEDs described in Chapter 7. The first NO is a unique vertex identifier. The ID is a variable field. The two last NO's are the identifiers of the *low* and *high* children.

The operators `imp`, `limp`, `nimp`, `nlimp`, and `biimp` are implication, left implication, negated implication, negated left implication, and biimplication. The operators `:=`, `?`, `!` and `ESUB` correspond to substitution, existential quantification, universal quantification and ESUB. The rest of the operators have standard names. The negation operator `not` takes two identical arguments.

The substitution assignment denotes a Boolean function in which an input is replaced with another Boolean function. The existential (universal) quantification assignment denotes a Boolean function in which an input is existentially (universally) quantified. In this case the input is no longer free and thus not a real input anymore. However, it should still be listed in the input part.

The outputs are specified as an ID and a NO. The ID is the name of the output. The NO is the unique identifier for a vertex in the BED.

A full-adder can be specified like this

```
inputs
  a b ci
assign
  2    a  ite   0   1
  3    b  ite   0   1
  4    ci ite   0   1
  5    -  xor   2   3
  6    -  xor   5   4
  7    -  and   2   3
  8    -  and   3   4
  9    -  and   2   4
```

```
  10   -  or    7   8
  11   -  or    9   10
outputs
  sum     6
  carry   11
```

EXAMPLE 1

Consider verifying that two different implementations of a full-adder are logically identical. Let the full-adders be described by the following script:

```
addinput a b ci;

let s1 = (a and b and ci) or ((((a or b) and ci)
nor (a and b)) and (a or b or ci));

let c1 = not (((a or b) and ci) nor (a and b));

let s2 = a xor b xor ci;

let c2 = (a and ci) or ((a and b) or (b and ci));
```

The outputs of the two full-adders are combined using `biimp` operators:

```
let sum_check = s1 biimp s2;
let co_check = c1 biimp c2;
```

The outputs `sum_check` and `co_check` can be verified to be tautologies with the following commands

```
upall sum_check
upall co_check
```

or

```
upone support(sum_check) sum_check
upone support(co_check) co_check
```

All commands return 1 indicating a tautology.

We now introduce an error in one of the full-adders by changing the line

```
let s1 = (a and b and ci) or ((((a or b) and ci)
nor (a and b)) and (a or b or ci));
```

to

```
let s1 = (a and b and ci) or ((((a nor b) and ci)
nor (a and b)) and (a or b or ci));
```

and run the script again. The `upone` and `upall` commands now return a value different from 1. The two circuits no longer implement the same Boolean function. The command

```
anynonsat sum_check
```

returns a falsifying assignment for `sum_check`, for example `[ ci ]` which corresponds to the assignments `a = 0, b = 0, ci = 1`. Use `eval` on each of the `s`'s to observe the different value for this assignment:

```
eval s1 [ ci ]
```

```
eval s2 [ ci ]
```

The first command results in a 0, the second in a 1.

EXAMPLE 2

Consider verifying that two multi-output combinational circuits are equivalent. Assume that the two circuits are described in the file `circuits.bed` in the BED file format. Each output in `circuits.bed` is a biimplication of two corresponding outputs from the original circuits. In other words, our verification task is to prove all outputs of `circuits.bed` to be tautologies.

Write a script-file, `script.com`, containing the following commands:

```
foreach root do "order fanin(root); upall root";
stat outputs;
halt;
```

Now run the `bed` tool with the following options:

```
bed -f script.com circuits.bed
```

`bed` reads `circuits.bed` and executes in turn each command in `script.com`. The first command iterates through all outputs in `circuits.bed`. For each output it first sets the variable ordering according to the FANIN heuristic. Then it calls UP_ALL to convert the BED for the output to a BDD.

After converting all output BEDs to BDDs, the script executes the command `stat outputs` which gives the number of tautologies among the outputs. In this way we can see whether the verification succeeded (all outputs are tautologies) or failed (some outputs were not tautologies). The command `halt` exits the `bed` tool.

Change the `foreach` command in `script.com` to one of the following commands:

```
foreach root do "order fanout(root); upall root";

foreach root do "order fanin(root); upone * root";

foreach root do "order fanout(root); upone * root";
```

The first command will use the DEPTH_FANOUT heuristic and UP_ALL. The remaining two commands will use UP_ONE with the FANIN and DEPTH_FANOUT heuristics, respectively.

The script `script.com` was used in the experiments presented in Tables 3.5 and 3.6.

Bibliography

[ABE00] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT solvers. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2000. 5.1, 5.3.1, 20, 5.6, 5.6

[AGD91] P. Ashar, A. Ghosh, and S. Devadas. Boolean satisfiability and equivalence checking using general binary decision diagrams. In *Proc. International Conf. Computer Design (ICCD)*, pages 259–264. IEEE Computer Society Press, 1991. 7

[AH] H. R. Andersen and H. Hulgaard. Boolean expression diagrams. *Information and Computation*. (To appear). 2, 2.2, 2.3, 2, 7

[AH97] H. R. Andersen and H. Hulgaard. Boolean expression diagrams. In *IEEE Symposium on Logic in Computer Science (LICS)*, July 1997. 2, 2.2, 2.3, 6, 7

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509–516, June 1978. 2.4

[BBS91] R. E. Bryant, D. L. Beatty, and C.-J. H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 397–402, June 1991. 5.6

[BC95] R. E. Bryant and Y.-A. Chen. Verification of arithmetic functions with binary moment diagrams. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 535–541, 1995. 2.4, 7, 7, 3.6

[BCC+99] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of

BDDs. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, 1999. 3.2.3, 5.1, 5.3.2, 5.5, 5.6

[BCCZ99] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1579 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999. 3.2.3, 5.1, 5.5, 5.6

[BCL+94] J. R. Burch, E. M. Clarke, D. E. Long, K. L. MacMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994. 5.3.1

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992. 5.1, 5.6

[BCRZ99] A. Biere, E. Clarke, R. Raimi, and Y. Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In *Computer Aided Verification (CAV)*, volume 1633 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999. 3.2.3, 5.1, 5.5, 5.6

[BD97] B. Becker and R. Drechsler. Decision diagrams in synthesis – algorithms, applications and extensions. In *VLSI Design Conference*, pages 46–50, Hyderabad, January 1997. 2.4

[BF85] F. Brglez and H. Fujiware. A neutral netlist of 10 combinational benchmarks circuits and a target translator in Fortran. In *Special Session International Symposium on Circuits and Systems (ISCAS)*, 1985. 3.2.4, 3.5.1, 22

[BFG+93] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hactel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 188–191, 1993. 7, 3.6

[Bje99] P. Bjesse. Symbolic model checking with sets of states represented as formulas. Technical Report CS-1999-102, Chalmers University of Technology, Sweden, June 1999. 3.2.3

[Bra93] D. Brand. Verification of large synthesized designs. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 534–537, 1993. 10, 3.7, 3.6

[BRB90] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*. IEEE 0738, 1990. 2.1

[BRRM91] K. M. Butler, D. E. Ross, R.Kapur, and M. R. Mercer. Heuristics to compute variable orderings for efficient manipulation of ordered binary decision diagrams. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, number 28, pages 417–420, 1991. 3.3

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986. 2, 2.2, 2.2, 2.2, 6, 2.4, 7, 3.5.1, 5.1, 7.4.1

[Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992. 2, 2.2, 2.2, 6, 7

[Bry95] R. E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 236–243, November 1995. 2.4

[CBM89] O. Coudert, C. Berthet, and J.C. Madre. Verification of synchronous sequential machines using symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373, Grenoble, France, June 1989. Springer-Verlag. 3.2.3, 5.3.1

[CCGR99] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 495–499, Trento, Italy, July 1999. Springer-Verlag. 5.4

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic

specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986. 5.2.1, 5.6

[CFZ95] E. M. Clarke, M. Fujita, and X. Zhao. Hybrid decision diagrams overcoming the limitations of MTBDDs and BMDs. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 159–163, Los Alamitos, Ca., USA, November 1995. IEEE Computer Society Press. 7

[CGP99] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999. 16, 5.6

[CGS98] Marco Cadoli, Andrea Giovanardi, and Marco Schaerf. An algorithm to evaluate quantified Boolean formulae. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 262–267, Menlo Park, July 1998. AAAI Press. 5.6

[CHP93] P.-Y. Chung, I. N. Hajj, and J. H. Patel. Efficient variable ordering heuristics for shared ROBDD. In *Proc. International Symposium on Circuits and Systems (ISCAS)*, pages 1690–1693, 1993. 3.3, 3.3.1

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990. 2.2, 3.4.1

[CM90] E. Cerny and C. Mauras. Tautology checking using cross-controllability and cross-observability relations. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, 1990. 3.6, 4.5

[CMZ+93] E. M. Clarke, K.L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large boolean functions with application to technology mapping. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 54–60, 1993. 7, 3.6

[CW96] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996. 1.3

[DBR95] R. Drechsler, B. Becker, and S. Ruppertz. K*BMDs: a new data structure for verification. In *IFI WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, Frankfurt, Germany, October 1995. 7

[DBR97a] R. Drechsler, B. Becker, and S. Ruppertz. The K*BMD: A verification data structure. *IEEE Design and Test of Computers*, 14(2):51–59, 1997. 2.4, 7

[DBR97b] R. Drechsler, B. Becker, and S. Ruppertz. Manipulation algorithms for K*BMDs. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1217 of *Lecture Notes in Computer Science*, pages 4–18. Springer-Verlag, 1997. 7

[DLL62] M. Davis, G. Longemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962. 5.6, 6.2, 6.5

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960. 5.6, 6.2, 6.5

[DR97] Y. Dutuit and A. Rauzy. Exact and truncated computations of prime implicants of coherent and noncoherent fault trees within aralia. *Reliability Engineering and System Safety*, 58(2):127–144, 1997. 7.3, 28, 7.3.2, 7.3.1, 7.5

[DR98] Y. Dutuit and A. Rauzy. Polynomial approximations of boolean functions by means of positive binary decision diagrams. In Lydersen, Hansen, and Sandtorv, editors, *Proceedings of European Safety and Reliability Association Conference, ESREL'98*, pages 1467–1472. Balkerna, Rotterdam, 1998. ISBN 90 54 10 966 1. 7.3, 28, 7.3.1, 7.5

[DST+94] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M.A. Perkowski. Efficient representation and manipulation of switching functions based on ordered kronecker functional decision diagrams. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 415–419, 1994. 7, 3.6

[FFK88] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvements of boolean comparison method based on binary decision diagrams. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 2–5, November 1988. 3.3, 3.3.1

[FMK91] M. Fujita, Y. Matsunga, and T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level

synthesis. In *Proc. European Conference on Design Automation (EDAC)*, pages 50–54, 1991. 5, 7

[FOH93] H. Fujii, G. Ootomo, and C. Hori. Interleaving based variable ordering methods for ordered binary decision diagrams. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, 1993. 3.3

[Fuj96] M. Fujita. Verification of arithmetic circuits by comparing two similar circuits. In *Computer Aided Verification (CAV)*, Lecture Notes in Computer Science, pages 159–168. Springer-Verlag, 1996. 1.3, 3.1

[GK00] M. Ganai and A. Kuehlmann. On-the-fly compression of logical circuits. In *Proc. of International Workshop on Logic Synthesis*, Dana Point, CA, June 2000. 3.6

[GKB97] E. I. Goldberg, Y. Kukimoto, and R. K. Brayton. Canonical TBDD's and their application to combinational verification. In *Proc. International Workshop on Logic Synthesis*, 1997. 3.6

[GM94] J. Gergov and C. Meinel. Efficient boolean manipulation with OBDD's can be extended to FBDD's. *IEEE Transactions on Computers*, 43(10):1197–1209, October 1994. 7, 3.6

[Gra00] H. Grauslund. Linear decision diagrams. Master's thesis, Department of Information Technology, Technical University of Denmark, Denmark, May 2000. Reference IT-E 843. 2.4, 7

[GS99] E. Giunchiglia and R. Sebastiani. Applying the Davis-Putnam procedure to non-clausal formulas. In *Proc. Italian National Conference on Artificial Intelligence*, volume 1792 of *Lecture Notes in Computer Science*, Bologna, Italy, September 1999. Springer-Verlag. 5.6, 6.5, 8.1

[Gup92] A. Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1(2/3):151–238, October 1992. 1.3

[Har96] J. Harrison. Staalmarck's algorithm as a HOL derived rule. *Lecture Notes in Computer Science*, 1125:221–234, 1996. 3.5.3, 3.13

[HC74] G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic*. Methuen & Co. Ltd, London, 1974. 16

[HDB96] A. Hett, R. Drechsler, and B. Becker. MORE: Alternative implementation of BDD-pakages by multi-operand synthesis. In *European Design Conference*, 1996. 7, 7.5

[HDB97] A. Hett, R. Drechsler, and B. Becker. Fast and efficient construction of BDDs by reordering based synthesis. In *IEEE European Design & Test Conference*, 1997. 7, 7.5

[HO82] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1982. 3.2.2

[HWA97] H. Hulgaard, P. F. Williams, and H. R. Andersen. Combinational logic-level verification using boolean expression diagrams. In *3rd International Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, September 1997. 1.5, 3

[HWA99] H. Hulgaard, P. F. Williams, and H. R. Andersen. Equivalence checking of combinational circuits using boolean expression diagrams. *IEEE Transactions on Computer Aided Design*, July 1999. 1.5, 3

[JBA+97] J. Jain, J. Bitner, M. S. Abadir, J. A. Abraham, and D. S. Fussell. Indexed BDDs: Algorithmic advances in techniques to represent and verify boolean functions. *IEEE Transactions on Computers*, 46(11):1230–1245, November 1997. 7, 3.6

[JMF95] J. Jain, R. Mukherjee, and M. Fujita. Advanced verification techniques based on learning. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 629–634, 1995. 3.6

[JPHS91] S.-W. Jeong, B. Plessier, G. D. Hactel, and F. Somenzi. Extended BDD's: Trading off canonicity for structure in verification algorithms. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 464–467, 1991. 7, 7, 3.3, 3.3.1, 7.5

[KK97] A. Kuehlmann and F. Krohm. Equivalence checking using cuts and heaps. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, volume 34, pages 263–268, 1997. 3.6

[KP94] W. Kunz and D. K. Pradhan. Recursive learning: A new implication technique for efficient solutions to CAD problems – test, verification, and optimization. *IEEE Transactions on Computer Aided Design*, 13(9):1143–1158, September 1994. 3.5.1, 3.6

[KPR96] W. Kunz, D. K. Pradhan, and S. M. Reddy. A novel framework for logic verification in a synthesis environment. *IEEE Transactions on Computer Aided Design*, 15(1):20–32, January 1996. 10, 3.6

[KSR92] U. Kebschull, E. Schubert, and W. Rosenstiel. Multilevel logic synthesis based on functional decision diagrams. In *Proc. European Conference on Design Automation (EDAC)*, pages 43–47, 1992. 7, 3.6

[Kun93] W. Kunz. HANNIBAL: An efficient tool for logic verification based on recursive learning. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 538–543, 1993. 3.6

[Lev99] N. Leveson. Building safety into computer controlled systems. Talk at Carnegie Mellon University, November 1999. For more information, visit Dr. Leveson's website at `http://sunnyday.mit.edu`. 1.4

[LPV94] Y. T. Lai, M. Pedram, and S. B. K. Vrudhula. EVBDD-based algorithms for linear integer programming, spectral transformation and function decomposition. *IEEE Transactions on Computer Aided Design*, 13(8):959–975, August 1994. 7, 7

[Mat96] Y. Matsunaga. An efficient equivalence checker for combinational circuits. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 629–634, 1996. 10, 3.6

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993. 5.1, 16, 5.2.2, 5.3

[Min93] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. ACM/IEEE Design Automation Conference (DAC)*, pages 272–277, 1993. 7, 7.3

[Min96] S. Minato. *Binary Decision Diagrams and Applications for VLSI CAD.* Kluwer Academic Publishers, 1996. 7, 3.3, 3.3.2

[ML98] J. Møller and J. Lichtenberg. Difference decision diagrams. Master's thesis, Department of Information Technology, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, August 1998. 2.4, 7

[MLAH99] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In *Proceedings 13th International Workshop on Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 111–125, Madrid, Spain, September 1999. 2.4, 7

[MSS99] J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48, 1999. 14, 5.3.2, 23

[MTJ+97] R. Mukherjee, K. Takayama, J. Jain, M. Fujita, J. A. Abraham, and D. S. Fussell. Flover: Filtering oriented combinational verification approach. In *Proc. International Workshop on Logic Synthesis*, May 1997. 10

[MWBSV88] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 6–9, 1988. 3.3, 3.3.1

[OYY93] H. Ochi, K. Yasuoka, and S. Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 48–55, 1993. 7.4.1

[Pap94] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, New York, 1994. 28

[PPC96] D. K. Pradhan, D. Paul, and M. Chatterjee. VERILAT: Verification using logic augmentation and transformations. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, November 1996. 10, 3.6

[Ric98] M. Richards. A tautology checker loosely related to Stålmarck's algorithm. Talk at seminar given at Cambridge, March 1998. 3.6

[Rin99] J. T. Rintanen. Improvements to the evaluation of quantified boolean formulae. In D. Thomas, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99-Vol2)*, pages 1192–1197. Morgan Kaufmann Publishers, August 1999. 5.6

[Rud93] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 42–47, 1993. 5

[S+92] E. Sentovich et al. SIS: A system for sequential circuit synthesis. Technical Report Memorandum No. UCB/ERL M92/41, Electronics Research Laboratory, Dept. of EECS, University of California, Berkeley, 1992. 3.5.2

[SDG95] A. Shen, S. Devadas, and A. Ghosh. Probabilistic manipulation of boolean functions using free boolean diagrams. *IEEE Transactions on Computer Aided Design*, 14, 1995. 7

[Seb94] R. Sebastiani. Applying GSAT to non-clausal formulas. *Journal of Artificial Intelligence Research (JAIR)*, 1:309–314, January 1994. 5.6, 6.5, 8.1

[Sht00] O. Shtrichman. Tuning SAT checkers for bounded model checking. In *Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 480–494, Chicago, U.S.A., July 2000. Springer-Verlag. 8.1

[SLM92] B. Selman, H. J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In P. Rosenbloom and P. Szolovits, editors, *Proc. Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. American Association for Artificial Intelligence, AAAI Press. 5.6, 6.5

[Som98] F. Somenzi. CUDD: CU Decision Diagram Package version 2.3.0. University of Colorado at Boulder, September 1998. `http://vlsi.colorado.edu/~fabio`. 10

[SS98] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. In G. Gopalakrishnan and P. J. Windley, editors, *Proc. Formal Methods in Computer-Aided Design, Second International Conference, FMCAD'98,*

Palo Alto/CA, USA, volume 1522 of *Lecture Notes in Computer Science*, pages 82–99, November 1998. 3.4, 3.6, 5.6

[SW95] D. Sieling and I. Wegener. Graph driven BDDs – a new data structure for boolean functions. *Theoretical Computer Science*, 141(1-2):283–310, 1995. 7, 3.6

[US94] T. E. Uribe and M. E. Stickel. Ordered binary decision diagrams and the Davis-Putnam procedure. In J.P. Jouannaud, editor, *1st International Conference on Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, September 1994. 3.6

[vE97] C.A.J. van Eijk. *Formal Methods for the Verification of Digital Circuits*. PhD thesis, Technische Universitet Eindhoven, 1997. 10, 3.6

[vE98] C.A.J. van Eijk. Sequential equivalence checking without state space traversal. In *Proc. International Conf. on Design Automation and Test of Electronic-based Systems (DATE)*, 1998. 1.3, 3.1

[vEJ94] C.A.J. van Eijk and G. L. J. M. Janssen. Exploiting structural similarities in a BDD-based verification method. In *Theorem Provers in Circuit Design*, volume 901 of *Lecture Notes in Computer Science*, pages 110–125. Springer-Verlag, 1994. 3.6

[VPL96] S. B. K. Vrudhula, M. Pedram, and Y. T Lai. *Edge Valued Binary Decision Diagrams*. Kluwer Academic Publishers, 1996. 7, 7

[WAH01] P. F. Williams, H. R. Andersen, and H. Hulgaard. Satisfiability checking using boolean expression diagrams. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, 2001. 1.5, 6

[WBCG00] P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 124–138, Chicago, U.S.A., July 2000. Springer-Verlag. 1.5, 5

[WHA99] P. F. Williams, H. Hulgaard, and H. R. Andersen. Equivalence checking of hierarchical combinational circuits. In *IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, September 1999. 1.5, 4

[Wil00] P. F. Williams. *Formal Verification Based on Boolean Expression Diagrams*. PhD thesis, Dept. of Information Technology, Technical University of Denmark, Lyngby, Denmark, August 2000. ISBN 87-89112-59-8. (document)

[WNR00] P. F. Williams, M. Nikolskaïa, and A. Rauzy. Bypassing BDD construction for reliability analysis. *Information Processing Letters*, 75(1-2):85–89, July 2000. 1.5, 7.3

[YCBO98] B. Yang, Y.-A. Chen, R. E. Bryant, and D. R. O'Hallaron. Space- and time-efficient BDD construction by working set control. In *Asian-Pacific Design Automation Conference (ASPDAC)*, pages 423–432, February 1998. 3.5.1, 7.4.1

[Zha97] H. Zhang. SATO: An efficient propositional prover. In William McCune, editor, *Proceedings of the 14th International Conference on Automated deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 272–275, Berlin, July 1997. Springer-Verlag. 15, 5.3.2, 23

Index