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

*Formal Veri cation based on*

*Boolean Expression Diagrams*

*Poul Frederick Williams*

*c 2001 Published by Elsevier Science B. V. Open access under* [*CC BY-NC-ND license*](http://creativecommons.org/licenses/by-nc-nd/3.0/)*.*

# *Abstract*

*This dissertation examines the use of a new data structure called Boolean* Expression Diagrams (BEDs) in the area of formal veri cation. The recently developed data structure allows fast and eÆcient manipulation of Boolean formulae. Many problems in formal veri cation 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 veri cation* problem which translates into tautology checking of Boolean formulae. Us- ing 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 veri cation process.

*Sometimes combinational circuits are speci ed in a hierarchical or mod-* ular 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 speci -* cations of nite state machines. It is well known how nite state machines and the evaluation of the temporal speci cations 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 satis ability is a problem which occurs* in veri cation of combinational circuits and in symbolic model checking. Often satis ability checking is associated with detecting errors. We examine how satis ability checking can be done using the BED data structure.

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

*iii*

*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 ap-* pendix 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 Pro- fessors Henrik Reif Andersen and Henrik Hulgaard, for giving me the op- portunity 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 Mel-* lon University. I spent six months working with him and his model checking group in Pittsburgh. His vast knowledge of computer science and the inspir- ing atmosphere surrounding him and his group made it a joy to work with him.

*Thanks are also due to Charg e de Recherches Antoine Rauzy from Centre* National de la Recherche Scienti que, Associate Professor David Sherman and Assistant Professor Macha Nikolska a from Universit e Bordeaux I. Dur- ing 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 J rn 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 M ller Jensen, Ken Friis Larsen, Jakob Lichtenberg, Jesper M ller and many others have also been helpful.

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

*v*

# *Contents*

[*List of Algorithms and Figures xi*](#_TOC_250035)

[*List of Tables xiii*](#_TOC_250034)

[*Symbols xv*](#_TOC_250033)

1. *Introduction 1*
   1. [*Formal Veri cation 3*](#_TOC_250032)
   2. [*Logic 4*](#_TOC_250031)
   3. [*Aim of This Dissertation 6*](#_TOC_250030)
   4. [*Safety and Reliability 8*](#_TOC_250029)
   5. [*Overview 9*](#_TOC_250028)
2. *Boolean Expression Diagrams 13*
   1. [*Data Structure 13*](#_TOC_250027)
   2. [*Properties 18*](#_TOC_250026)
   3. [*Algorithms 20*](#_TOC_250025)
   4. [*Related Work 28*](#_TOC_250024)
3. *Flat Equivalence Checking 33*
   1. [*Introduction 33*](#_TOC_250023)
   2. [*Simpli cations 36*](#_TOC_250022)
      1. *Operator Sets 36*
      2. *Rewriting 37*
      3. *Other Simpli cation Methods 41*
      4. *Experimental Results for Simpli cations 43*
   3. [*Variable Ordering 45*](#_TOC_250021)
      1. *The Fanin Heuristic 48*
      2. *The Depth Fanout Heuristic 49*
   4. [*St almarck's Method 50*](#_TOC_250020)

*vii*

* + 1. *Implementation of St almarck's Method using BEDs . 55*
  1. [*Experimental Results 57*](#_TOC_250019)
     1. *The ISCAS'85 Benchmark Suite 57*
     2. *The LGSynth'91 Benchmark Suite 63*
     3. *St almarck's Method 66*
  2. [*Related Work 72*](#_TOC_250018)
  3. [*Conclusion 75*](#_TOC_250017)

1. *Hierarchical Equivalence Checking 77*
   1. [*Introduction 77*](#_TOC_250016)
   2. [*Hierarchical Combinational Circuits 79*](#_TOC_250015)
   3. [*Cut Propagation 82*](#_TOC_250014)
      1. *Example 82*
      2. *Moving Cuts 84*
      3. *Build vs. Propagate 86*
   4. [*Experimental Results 87*](#_TOC_250013)
   5. [*Related Work 89*](#_TOC_250012)
   6. [*Conclusion 89*](#_TOC_250011)
2. *Symbolic Model Checking 91*
   1. [*Introduction 91*](#_TOC_250010)
   2. [*Theory of Model Checking 93*](#_TOC_250009)
      1. *Computation Tree Logic 95*
      2. *Model Checking using Propositional Logic 99*
   3. [*Model Checking with BEDs 101*](#_TOC_250008)
      1. *Quanti cation 103*
      2. *Satis ability Checking 108*
      3. *Example 110*
   4. [*Experimental Results 114*](#_TOC_250007)
      1. *Multiplier 115*
      2. *Barrel Shifter 118*
   5. [*Model Checking of lfp-CTL 120*](#_TOC_250006)
      1. *Theory 121*
      2. *Implementation 126*
   6. [*Related Work 127*](#_TOC_250005)
   7. [*Conclusion 129*](#_TOC_250004)
3. *Satis ability 131*
   1. [*Introduction 131*](#_TOC_250003)
   2. *Satis ability Using CNF Formulae 132*

*ix*

* 1. *Satis ability Using BEDs 133*
  2. [*Experimental Results 135*](#_TOC_250002)
  3. [*Related Work 138*](#_TOC_250001)
  4. [*Conclusion 138*](#_TOC_250000)

|  |  |  |  |
| --- | --- | --- | --- |
| *7 Extending BEDs* | | | *141* |
| *7.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . .* | | | *141* |
| *7.2 New Types of Vertices . . . . . . . . . . . . . . . . . . . . . .* | | | *142* |
|  | *7.2.1* | *Existential Quanti cation . . . . . . . . . . . . . . . .* | *142* |
|  | *7.2.2* | *Universal Quanti cation . . . . . . . . . . . . . . . . .* | *143* |
|  | *7.2.3* | *Substitution . . . . . . . . . . . . . . . . . . . . . . . .* | *144* |
|  | *7.2.4* | *Operators on Vectors . . . . . . . . . . . . . . . . . . .* | *145* |
|  | *7.2.5* | *Implementation . . . . . . . . . . . . . . . . . . . . . .* | *150* |
| *7.3* | *Fault* | *Tree Analysis . . . . . . . . . . . . . . . . . . . . . . . .* | *151* |
|  | *7.3.1* | *Minimal P-Cuts with BEDs . . . . . . . . . . . . . . .* | *154* |

*7.4 Experimental Results 156*

*7.4.1 Substitution 156*

*7.4.2 P-Cut 157*

*7.5 Related Work 159*

*7.6 Conclusion 160*

|  |  |  |
| --- | --- | --- |
| *8* | *Conclusion* | *161* |
|  | *8.1 Future Directions . . . . . . . . . . . . . . . . . . . . . . . . .* | *163* |
| *A* | *The BED Tool* | *165* |

*Bibliography 177*

*Index 189*

# *List of Algorithms and* Figures

*1.1 Two equivalent combinational circuits 1*

*1.2 The real and the formal worlds 3*

*2.1 An example of a BED 17*

*2.2 The BED array 18*

*2.3 The Mk algorithm 19*

*2.4 Memory usage in BDD construction 21*

*2.5 The up step 22*

*2.6 The Up One algorithm 24*

*2.7 The Up One0 algorithm 25*

*2.8 The Apply algorithm 26*

*2.9 The Up All algorithm 27*

*2.10 The Up algorithm 29*

*3.1 Circuit with memory 34*

*3.2 Rewrite rules for BEDs 39*

*3.3 The Mk algorithm (modi ed for rewriting rules) 40*

*3.4 Lost sharing 41*

*3.5 Balancing BEDs 42*

*3.6 The 17th output bit of multipliers 46*

*3.7 The 17th output bit of multipliers with rewriting rules 47*

*3.8 A four-input AND gate and a corresponding BED 48*

*3.9 The Fanin heuristic 49*

*3.10 The Depth Fanout heuristic 50*

*3.11 Parse-tree for Boolean formulae 51*

*3.12 Rules Au for the triplet u $ l ^ h 52*

*3.13 The 0-Sat algorithm 53*

*3.14 The (k + 1)-Sat algorithm 53*

*xi*

*3.15 Parse-tree for 0-hard Boolean formulae 55*

*3.16 A 1-hard and a 0-easy BED 56*

*4.1 Hierarchical 4-bit adder 78*

*4.2 Two 4-bit adders 79*

*4.3 The Prop algorithm 84*

*4.4 The Build algorithm 85*

*4.5 The Propagate algorithm 85*

*4.6 Two hierarchical circuits 87*

*5.1 An SMV program and the associated Kripke structure 93*

*5.2 A modi ed Kripke structure 94*

*5.3 The Lfp and Gfp operators 97*

*5.4 The QbS algorithm 104*

*5.5 The EF algorithm 106*

*5.6 A modulo-4 counter 110*

*5.7 SMV program for a modulo-4 counter 111*

*5.8 BEDs for transition function 112*

*5.9 Graph of runtimes for multiplier example 117*

*5.10 The ModelCheck algorithm 126*

*6.1 The DP (Davis-Putnam) algorithm 133*

*6.2 The BedSat algorithm 134*

*6.3 An illustration of the BedSat algorithm 136*

*7.1 Vector existential quanti cation 147*

*7.2 The up step for map vertices 148*

*7.3 The Mk algorithm modi ed for ESUB 150*

*7.4 The Up One0 algorithm modi ed for ESUB 151*

*7.5 The Up All algorithm modi ed for ESUB 152*

*7.6 Illustration of p-cut computation 156*

*7.7 A 2n-bit multiplier 157*

*7.8 A 2n-bit multiplier using substitution 158*

# *List of Tables*

*2.1 The 16 binary Boolean connectives and their truth tables . . 15*

*2.2 Overview of decision diagrams 31*

*2.3 References to decision diagrams 32*

*3.1 ISCAS'85 results without simpli cations 44*

*3.2 ISCAS'85 results with simpli cations 44*

*3.3 Size and functionality of the ISCAS'85 benchmark circuits 58*

*3.4 Erroneous circuits from the ISCAS'85 benchmark suite 59*

*3.5 ISCAS'85 results for Up All 60*

*3.6 ISCAS'85 results for Up One 61*

*3.7 ISCAS'85 comparisons I 63*

*3.8 ISCAS'85 comparisons II 64*

*3.9 Combinational LGSynth'91 circuits (easy) 67*

*3.10 Combinational LGSynth'91 circuits (hard) 68*

*3.11 Sequential LGSynth'91 circuits (easy) 69*

*3.12 Sequential LGSynth'91 circuits (hard) 70*

*3.13 Boolean satis ability test cases 71*

*3.14 ISCAS'85 results for St almarck's method 71*

*3.15 The e ect of minimizing 72*

*4.1 Hierarchical adders and multipliers 88*

*5.1 Correspondence between set and logic notations 100*

*5.2 16-bit multipliers 116*

*5.3 Iterative squaring 118*

*5.4 Bug D 119*

*5.5 Large shift-and-add multipliers 120*

*5.6 Barrel shifters 121*

*6.1 Satis ability and tautology 132*

*6.2 BedSat on ISCAS'85 benchmarks 137*

*xiii*

*6.3 BedSat on model checking problems 139*

*7.1 Veri cation of large integer multipliers 158*

*7.2 P-cut results on industrial examples 159*

# *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 vari- ables*

*; Formulae f , g, h Formulae*

*[ =x] Substitution of formula for variable x in*

*: Negation*

*K0 Connective, constant 0 Table 2.1*

*^ Connective, conjunction Table 2.1*

*6! Connective, negated implication Table 2.1*

*1 Connective, projection on rst argument Table 2.1 6 Connective, negated left-implication Table 2.1*

*2 Connective, projection on second argument Table 2.1*

*(continued on next page)*

*xv*

|  |  |  |
| --- | --- | --- |
| *Notation* | *Description* | *Reference* |
|  | *Connective, exclusive or* | *Table 2.1* |
| *\_* | *Connective, disjunction* | *Table 2.1* |
| *\_* | *Connective, negated disjunction* | *Table 2.1* |
| *$* | *Connective, biimplication* | *Table 2.1* |
| *2* | *Connective, negation of second argument* | *Table 2.1* |
|  | *Connective, left-implication* | *Table 2.1* |
| *1* | *Connective, negation of rst argument* | *Table 2.1* |
| *!* | *Connective, implication* | *Table 2.1* |
| *^* | *Connective, negated conjunction* | *Table 2.1* |
| *K 1* | *Connective, constant 1* | *Table 2.1* |
| *x ! ;* | *If-then-else operator* | *Equation 2.1* |
|  | *Any binary Boolean connective* |  |
| *9* | *Existential quanti cation* |  |
| *8* | *Universal quanti cation* |  |
| *SAT( )* | *Satis ability problem* | *De nition 1.2.2* |
| *TAUT( )* | *Tautology problem* | *De nition 1.2.3* |
| *S* | *Characteristic function for set S* | *De nition 1.2.5* |
| *0*  *1*  *u, v, l, h low (v), high(v)*  *var (v), op(v)*    *f v sup(v)* | *BED*  *Zero terminal One terminal Vertices*  *Low and high attributes of vertex*  *Variable and operator attributes of vertex*  *Either a variable or an operator. (v) is short for either var (v) or op(v).*  *Boolean function de ned by vertex Support of BED* | *Section 2.1*  *Section 2.1*  *De nition 2.1.1*  *De nition 2.1.1*  *De nition 2.1.2*  *De nition 2.1.5* |

*(continued on next page)*

*Notation* jvj

*BED for*

*BED u*

*<*

*depth(v)* u v

*Description* Size of BED

*The BED obtained by a straightforward*

*translation of formula*

*The BED rooted at vertex u* Ordering of variables or vertices The depth of a vertex v

*Path from vertex u to vertex v*

*Reference*

*De nition 2.1.6*

*De nition 2.1.8*

*Section 2.1*

*De nition 3.3.1*

*De nition 2.1.7*

*ST ALMARCK*

*, R* k

*C* Au

*Equivalence relation* Saturation depth

*Set of axioms from equivalence relation Set of rules from vertex (triplet) u*

*Section 3.4*

*Section 3.4*

*Equation 3.2*

*Section 3.4*

*HIERARCHICAL CIRCUITS*

*HCC(C; c)*

*c* i

*s,t,u,v,x,y*

*p* K H

*Rin Rout*

*[M ap]*

*Hierarchical combinational circuit. It consists* of a set of cells C and a top cell c 2 C.

*Cell*

*Instantiation of a cell*

*Variables or vectors of variables* Path in a container cell

*Cut*

*Cut-relation* Input relation Output relation

*Renaming of variables*

*De nition 4.2.1*

*De nition 4.2.2*

*De nition 4.2.3*

*De nition 4.2.4*

*De nition 4.2.5*

*De nition 4.2.6*

*Equation 4.2*

*(Equation 4.2)*

*(continued on next page)*

*Notation*

*Description*

*MODEL CHECKING*

*Reference*

*M* T I X S

*s*

*x*

*`* A

*si x j*

*Finite state machine* Transition function / relation Set of initial states

*Set of inputs*

*Set of states* State

*Input*

*Labeling of states*

*Set of atomic propositions*

*i j*

*De nition 5.2.2*

*De nitions 5.2.1 and 5.2.2*

*De nition 5.2.2*

*De nition 5.2.2*

*De nition 5.2.2*

*De nition 5.2.2*

*De nition 5.2.2*

*s Transition from state s*

*x*

*to state s*

*on input*

*De nition 5.2.3*

*si k sj M j=*

*[[ ]]*

*[[ ]]k* [ ]k

*=s*

*Path of length k from state si to state sj* System M models speci cation Semantics of CTL formula

*Similar to [[ ]] but with exactly k iterations in*

*each xed point computation*

*Similar to [[ ]]k but with input variables still* in the formulae

*CTL formulae and are s-equivalent*

*De nition 5.2.4*

*De nitions 5.2.5 and 5.2.14*

*De nition 5.2.10*

*De nition 5.5.4*

*De nition 5.5.7*

*De nition 5.5.9*

*FAULT TREES*

*x* L; X

*x:L*

*c X*

*Boolean variable* Sets of variables

*Union of sets fxg and L*

*Product of only positive literals*

*Minterm for formula f obtained by adding to*

*the negative literals formed over all the vari-* ables in f but not

*Section 7.3*

*Section 7.3*

*Section 7.3*

*Section 7.3*

*Section 7.3*

*(continued on next page)*

*Notation*

*(f )*

*k(f )*

*Description*

*Set of minimal p-cuts for f*

*Set of minimal p-cuts for f with at most k literals in length*

*Reference*

*Section 7.3*

*Section 7.3*

*MISCELLANEOUS*

*B R Z*

*7!*

*P( )*

*The set f0; 1g*

*The set of all real numbers The set of all integers Mapping*

*Assignment*

*Power set*

*Set transformer*

*De nition 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 veri cation technique

*i1 i1*



*00*

*0*

*o*

*o*

*i2 1 i2 1*

*i3 i3*

*0*

*o2*

*00*

*i4 i4 2*

*o*

*Figure 1.1: Two equivalent combinational circuits.*

*by proving that output o0 and output o00 are equivalent. After the seminar a*

*1*

*1*

*couple of people from the audience came to me and said that the second pair*

*of outputs were not equivalent. They argued that output o0*

*2*

*only depends on*

*inputs i3 and i4, while output o00 depends on i2, i3, and i4. Their argument*

*2*

*is correct. However, a more careful analysis of output o00*

*2*

*shows that the*

*value of input i2 is masked by the circuitry and therefore cannot in uence* 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.*

*1*

*to solve. Again the equivalence checking problem is a good example. Each* input can be either high or low. With four inputs, we have 24 = 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 2100 = 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 nd the error.

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

*a (b + c) = a b + a 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 in nitely 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 di er 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 nd 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-

(1)

Real problem

Real solution

(2)

(4)

(3)

Formal problem

Formal solution

*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 in- teresting. 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.

## *Formal Veri cation*

*The title of this dissertation is \Formal Veri cation Based on Boolean Ex-* pression Diagrams". In this section we de ne what we mean by formal veri cation. According to Merriam-Webster's dictionary2, the word \ver- i cation" 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 ar-* rangement of elements rather than content." So formal veri cation is the process of establishing the truth using outward form, structure, relation- ships, or arrangement of elements rather than content.

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

1. *See* [*http://www.m-w.com.*](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 mathe-* matical or logical reasonings which establishes that the system has the property.

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

## *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 :, where negation is de ned as :0 = 1 and :1 = 0. There are 16 Boolean connectives of two arguments. Not all 16 Boolean connectives are necessary. For exam- ple, 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.

*De nition 1.2.1. A formula in propositional logic can be generated from* the following grammar:

*f ::= 0 j 1 j variable j :f j f \_ 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 , we can evaluate to either 0 or 1 by replacing all variables in 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 contra- diction is unsatis able since no variable assignment makes it evaluate to 1. A formula which is not a contradiction is satis able.

*We now de ne two related problems in propositional logic.*

*De nition 1.2.2 (Satis ability Problem). Let be a formula in propo-* sitional logic. Determine whether a variable assignment exists for , such that evaluates to 1 for this assignment.

*De nition 1.2.3 (Tautology Problem). Let be a formula in proposi-* tional logic. Determine if evaluates to 1 for all possible variable assign- ments.

*We use SAT( ) to denote the function that is 1 if is satis able and*

*0 otherwise. Likewise, TAUT( ) denotes the function that is 1 if is a* tautology and 0 otherwise. Note that SAT( ) = :TAUT(: ).

*Propositional logic can be extended to quanti ed Boolean formulae (QBF)* by introducing the existential quanti er 9:

*De nition 1.2.4. A formula in QBF can be generated from the following* grammar:

*f ::= 0 j 1 j variable j :f j f \_ f j 9 variable : f : The semantics of the existential quanti er is*

*9x : [0=x] \_ [1=x] ; (1.1)*

*where [b=x] means a substitution of b for x in . The universal quanti er*

*8 can be obtained from the existential quanti er using negation:*

*8x : :9x : : [0=x] ^ [1=x] : (1.2)*

*A variable is said to be free in formula if it is not bound by a quanti er.* Note that solving the satis ability (tautology) problem for corresponds to adding existential (universal) quanti ers for all free variables in 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* speci cation of a nite state machine. In Chapter 5 we describe both CTL and nite 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.

*De nition 1.2.5 (Characteristic Function). Let S be a set. The char-* acteristic function S : S 7! B for S is given by:

*(s) = 1 : if s 2 S*

*S 0 : otherwise*

*Since characteristic functions are used extensively, we often omit the 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 f0; 1; 2; 3; 4; 5; 6; 7g. Using three bits hs2s1s0i we can represent all eight numbers using their binary represen- tation such that hs2s1s0i = h000i represents the number 0, h001i the number 1, and so on up to h111i for the number 7. Now, the characteristic function:

*= s1 ^ :s0*

*represents the set f2; 6g because the encoding for two (h010i) and six (h110i)* are the only numbers to have s1 set to true and s0 set to false.

## *Aim of This Dissertation*

*In this dissertation we look at ways to solve problems in the domain of* formal veri cation. 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 struc-* ture called Boolean Expression Diagrams. Our aim is to apply this data structure to formal veri cation problems. We have chosen to concentrate on the following problems within formal veri cation:

*Equivalence checking of combinational circuits,*

*Model checking of transition systems, and*

*Fault tree analysis.*

*Equivalence checking is the problem of determining whether two com-* binational 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 modi ed. 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 veri cation 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 satis es* its temporal speci cation. 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 traÆc 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 speci cation and we can check whether our traÆc light system satis es 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 (\fail- ure" 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 di erent areas within formal veri cation.*

*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 di erent kinds of objects: A nite* state machine and a CTL speci cation. We encode the nite state machine in propositional logic. Based on the CTL speci cation, we compute a set of states which are valid initial states for the nite 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 veri cation 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 veri cation than the ones* we deal with in this dissertation. Gupta has written a thorough survey of formal veri cation methods with respect to hardware [Gup92]. We do not hesitate to recommend her paper to readers interested in getting an overview of formal veri cation. 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 veri cation) have been applied with success.

## *Safety and Reliability*

*In the previous sections we have presented formal veri cation 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 correct-* ness 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 cor-* rect 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 traÆc light intersection example. We verify that we have red light in at least one direction at all times. Assume that our particular traÆc light intersection has this property. Is it a correct, safe and reliable intersection? No, not nec- essarily. We have, for example, not veri ed that the lights actually change. A bug in our system may cause the traÆc lights to show red in both direc- tions at all times. This is naturally not a correct behavior of a traÆc 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 \ ight 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 nal products catch errors. Knowledge of psychology and cognitive engineering is also important to avoid interface errors.

*In this dissertation we only consider formal veri cation. We want, how-* ever, to stress that formal veri cation 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.

## *Overview*

*Chapter 2 introduces Boolean Expression Diagrams as a data structure for* representing and manipulating Boolean formulae. We explain how to im- plement 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 at 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 check- ing problem can then be viewed as an instance of the tautology problem TAUT( 1 $ 2), where 1 and 2 are the formulae for a pair of corre- sponding outputs for the two circuits in question. We consider a number of ideas including simpli cation 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. Combina-* tional logic-level veri cation using boolean expression diagrams. In 3rd International Workshop on Applications of the Reed-Muller Ex- pansion 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 at 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 ts 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 speci cations hold for nite state machines. The mathemat- ics behind the veri cation 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 Ex- pression Diagrams. This has some consequences, both positive and negative. For example, one of the consequences is that with Boolean Expression Di- agrams 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. Com-* bining decision diagrams and SAT procedures for eÆcient symbolic model checking. In Computer Aided Veri cation (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 satis ability 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 veri cation 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 do- ing satis ability 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. Satis abil-* ity checking using boolean expression diagrams. In T. Margaria and

*W. Yi, editors, Tools and Algorithms for the Construction and Anal-* ysis of Systems (TACAS), volume 2031 of Lecture Notes in Computer Science, 2001.

*Chapter 7 extends the Boolean Expression Diagram data structure. We* introduce quanti cation 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 eÆcient way than using standard methods. The work on fault tree analysis is based on the paper [WNR00]:

*[WNR00] P. F. Williams, M. Nikolska a, 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 online3.

1. *See* [*http://www.it-c.dk/research/bed*](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 repre-* senting and manipulating Boolean formulae [AH, AH97]. The data structure is called Boolean Expression Diagrams, or BEDs for short. It is a general- ization of Bryant's Binary Decision Diagrams (BDDs) [Bry86, Bry92]. In this chapter we present the BED data structure, its properties, and the al- gorithms for working with it. Part of this chapter is a review of Andersen and Hulgaard's work.

## *Data Structure*

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

*De nition 2.1.1 (Boolean Expression Diagram). A Boolean Expres-* sion 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) 2 f0; 1g.*

*A variable vertex v has as attributes a Boolean variable var (v), and* two children low (v); high (v) 2 V .

*An operator vertex v has as attributes a binary Boolean operator* op(v), and two children low (v), high(v) 2 V .

*13*

*The edge set E is de ned by*

*E = (v; low (v)); (v; high (v)) v 2 V and v is a non-terminal vertex :*

*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 ! f1; f0 de ned by

*x ! f1; f0 = (x ^ f1) \_ (:x ^ f0) : (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:

*De nition 2.1.2 (Semantics). A vertex v in a BED denotes a Boolean* function f v de ned recursively as:

*If v is a terminal vertex, then f v = val (v).*

*If v is a variable vertex, then f v = var (v) ! 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 de nitions. Negation* can be obtained by using the 1 operator with a dummy second argument. For readability, we use : for negation in BEDs instead of 1.

*De nition 2.1.3 (Reduced). A BED is called reduced if it has the follow-* ing 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.*

*De nition 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(x;y)*

|  |  |  |
| --- | --- | --- |
| *op* | *x:0011*  *y 0101* | *Name of Boolean connective* |
| *K 0* | *0000* | *Constant 0* |
| *^* | *0001* | *Conjunction* |
| *6!* | *0010* | *Negated implication* |
| *1* | *0011* | *Projection on rst argument* |
| *6* | *0100* | *Negated left-implication* |
| *2* | *0101* | *Projection on second argument* |
|  | *0110* | *Exclusive or* |
| *\_* | *0111* | *Disjunction* |
| *\_* | *1000* | *Negated disjunction* |
| *$* | *1001* | *Biimplication* |
| *2* | *1010* | *Negation of second argument* |
|  | *1011* | *Left-implication* |
| *1* | *1100* | *Negation of rst argument* |
| *!* | *1101* | *Implication* |
| *^* | *1110* | *Negated conjunction* |
| *K 1* | *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* De nition 2.1.3 and 2.1.4.

*De nition 2.1.5 (Support). The support of a BED u, written sup(u), is* the set of free variables1 in f u.

*De nition 2.1.6 (Size). The size of a BED u, written juj, is the number* of vertices in u.

*De nition 2.1.7 (Path). There is a path from vertex u to vertex v, and* we write u v, if there exists a nite sequence of vertices hu1; u2;::: ; uni (n 1) such that:

*u = u1*

*v = un*

*For all i = 1;::: ;n 1, either ui+1 = low (ui) or ui+1 = high(ui)*

*It is convenient to talk about the BED for a formula. We use this* terminology to mean the BED de ned in De nition 2.1.8:

*De nition 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 ver-* tex 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 $ a ^ (a \_ b).*

*The BED is both reduced and free. The support is fa; bg. 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 elds op, var, low, and high

*1All variables are free in a free BED. However, we later introduce quanti ers to the BED and quanti ed variables are not free.*

*Figure 2.1: The BED for a $ a ^ (a \_ b). The dotted edges are the low ones.*



*$*

*^*

*\_*

*b*

*a*

*0*

*1*

*corresponding to the operator, variable, low child, and high child attributes* of a vertex. Each vertex is identi ed by it position in the array. Position

*0 and 1 correspond to the vertices 0 and 1. To nd 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 elds to each entry in the array: H and next. The H eld in entry n contains the index of the vertex with hash value n. The next eld is used to resolve collisions. To nd a vertex ( ; low; high), where 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 mark-

*ing eld mark as well. We group the elds mark, var and op in one memory* word. The remaining elds each require one word. On a 32 bit machine ar- chitecture, 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 $ a ^ (a \_ 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* Mk( ; l; h) returns the identity of a ( ; l; h) vertex. If is a variable, then ( ; l; h) is a variable vertex with variable , low child l and high child h. If

*is an operator, then ( ; l; h) is an operator vertex with operator and low* and high children l and h. Lines 1 through 6 handle the three requirements of reduced BEDs; see De nition 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 eld for reference counting of the*

*H next*

*op var*

*low high*

*mark*

*0*



*7*

*4*

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  | *a* | *0* | *1* |
|  | *b* | *0* | *1* |
| *\_* |  | *2* | *3* |
| *^* |  | *2* | *4* |
| *$* |  | *2* | *5* |
|  |  |  |  |

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |

*1*

*2*

*3*

*4*

*5*

*6*

*7*

*Figure 2.2: Implementation of the BED data structure as an array. Looking up a vertex v = ( ; low; high) is done like this: The hash value of v, say 2, is used as an entry point in the array. The H eld points to a vertex, here 4. If this vertex is not v then we follow the pointer in the next eld 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 eld of all entries not in use.

## *Properties*

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

*De nition 2.2.1 (Binary Decision Diagrams). A Binary Decision Di-* agram (BDD) is a BED with only terminal and variable vertices.

*By de ning BDDs in terms of BEDs, we already have the semantics* (De nition 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( ; l; h)*

*1: if there exists a ( ; l; h) vertex then* 2: return that vertex

*3: else if is a variable and l = h then* 4: return l

*5: else if 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(:; l; ) or Mk(:; h; )* 7: else

*8: return new vertex ( ; l; h)*

*Algorithm 2.3: The Mk algorithm. The algorithm takes a variable or operator*

*and two BEDs l and h as arguments and returns a BED vertex with variable or operator , low child l and high child h. The two dots ( ) in line 6 indicate dummy second arguments as explained on page 14.*

*De nition 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 ^ y and y ^ x are clearly* semantically equivalent, but they are syntactically di erent.

*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 satis ability problem for BDDs is easy. Because of the canon-* icity of BDDs, there is only one representation of an unsatis able function: the terminal vertex 0. All other BDDs represent satis able functions. The satis ability 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 satis ability prob-* lem 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 satis ability 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.

## *Algorithms*

*One way to solve the satis ability 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 rst 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.*

*Memory*

*Time*

*drops. In some cases it drops to almost nothing. Consider constructing the* BDD for = $ , where the BDD for is large. The size of the end result is only one vertex as is a tautology. However, as an intermediate step we need to construct the BDD for , 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 e ective.

*Andersen and Hulgaard present in [AH, AH97] two algorithms for con-* verting 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 ordered2.

*The following equations form the basis of Up One and Up All. The*

*2Strictly speaking, the original BED must be ordered (as per De nition 2.2.2 but with*

*\BED" instead of \BDD"), but this is not a problem since the BED for any Boolean formula (as de ned in De nition 2.1.8) is always ordered. If the original BED is not ordered, it is possible to patch the algorithms such that the nal BDD is indeed ordered*

*{ see [AH] for details.*

*equations allow local modi cations to Boolean formulae.*

*(x ! f1; f2) (x ! g1; g2) = x ! (f1 g1); (f2 g2) (2.2)* (x ! f1; f2) g = x ! (f1 g); (f2 g)

*The operator 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

*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.



*x*

*x*



*x*

*f1 f2 g1 g2 f1 g1 f2 g2*

*g*

*f1 f2 f1 g f2 g*



*x*



*x*

*Figure 2.5: The up step from Equation 2.2 shown as BED operations. x is a variable and is an operator. The transformations are also valid if we replace 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 (De nition 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 runtime3.

*Observation 2.3.1 (Up One). For a variable x and a BED u, let v =* Up One(x; u). Then Up One has the following properties:

*(i) f v = f u.*

*(ii) jvj 2juj 1.*

*(iii) The running time of Up One is O(juj).*

*Consider repeated application of Up One to the variables x1; x2;::: ; xn,* where each variable is pulled to the root. On the way to the root, variable xi passes the variables x1;::: ; xi 1. The nal result is a BDD with the order xn; xn 1;::: ; x1. A lot of work goes into pulling variables up past the variables previously pulled up. The following \early stop" modi cation 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 rst. This would result in a BDD with the order x1; x2;::: ; xn. 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 interme-* diate 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 origi- nal Up One algorithm by Andersen and Hulgaard [AH97] (Algorithm 2.6). Instead of returning a vertex, Up One0 returns a pair of vertices. The cor- respondence between Up One and Up One0 is:

*Observation 2.3.2 (Up One0). Let x be a variable and u be a vertex.* Let v = Up One(x; u) and v0 = Mk(x; l; h), where (l; h) = Up One0(x; u). Then v and v0 are identical vertices.

*For the experimental results we use a version of Up One based on Al-* gorithm 2.7 modi ed for early stop in repeated applications.

*3In 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*

*rst 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: tmp1 Mk( (u); low (l); low (h))

*6: tmp2 Mk( (u); high (l); high (h))* 7: r Mk(x; tmp1; tmp2)

*8: else if l is a variable x vertex then* 9: tmp1 Mk( (u); low (l); h)

*10: tmp2 Mk( (u); high (l); h) 11: r Mk(x; tmp1; tmp2)*

*12: else if h is a variable x vertex then* 13: tmp1 Mk( (u); l; low (h))

*14: tmp2 Mk( (u); l; high (h)) 15: r Mk(x; tmp1; tmp2)*

*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. (u) is the \tag" op(u) for operator vertices and var (u) for variable vertices.*

*Name: Up One0(x; u)*

*Require: The memorization table M is initialized to empty prior to the*

*rst 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) Up One0(x; low (u))

*5: (hl; hh) Up One0(x; high(u))*

*6: rl Mk( (u); ll; hl)*

*7: rh Mk( (u); lh; hh)*

*8: insert ((x; u); (rl; rh)) in M* 9: return (rl; rh)

*Algorithm 2.7: The modi ed 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(jljjhj).

*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*

*rst 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*

*rst 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( (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( (u); low (l); low (h)))*

*10: tmp2 Up All(Mk( (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( (u); low (l); h))*

*14: tmp2 Up All(Mk( (u); high (l); h)) 15: r Mk(var (l); tmp1; tmp2)*

*16: else*

*17: tmp1 Up All(Mk( (u); l; low (h)))*

*18: tmp2 Up All(Mk( (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 de ned 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.

## *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 eÆcient 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 veri cation. 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 di erent 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 Dn and the range is R: f : Dn 7! R. Many decision diagrams, for example BDDs, have D = R = B .

*The graph vertices are labeled. All decision diagrams have vertices la-* beled with variables. For decision diagrams with D = 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 B . Consider a variable vertex v with variable x:

*f = (:x ^ f low(v)) \_ (x ^ f high(v)) Shannon (S)*

*f = f low(v) (x ^ f high(v)) positive Davio (pD)* f = f low(v) (:x ^ f high(v)) negative Davio (nD)

*The negative Davio decomposition is also referred to as the Reed-Muller* decomposition. The decomposition is xed for each variable in a decision diagram. However, it is possible to use two or more di erent decompositions within one decision diagram. The three composition types can be general- ized 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: Up(s; u)*

*Require: The memorization table M is initialized to empty prior to the*

*rst 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( (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( (u); low (l); low (h)))*

*9: tmp2 Up(s; Mk( (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( (u); l; low (h)))

*13: tmp2 Up(s; Mk( (u); l; high (h))) 14: r Mk(var (h); tmp1; tmp2)*

*15: else*

*16: tmp1 Up(s; Mk( (u); low (l); h))*

*17: tmp2 Up(s; Mk( (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( (u); l; low (h)))

*21: tmp2 Up(s; Mk( (u); l; high (h))) 22: r Mk(var (h); tmp1; tmp2)*

*23: else*

*24: r Mk( (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 de ned as for Apply (see Algorithm 2.8). For readability we use the abbreviation vl for \l is variable vertex and var (l) 2 s". The terms vh and vu are de ned 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 in- coming 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 di erence between two real variables. The di erence is compared to a constant. LDDs have a linear combination of a number of real variables in the vertices. The linear combi- nation 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 operators4. 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 men- tion it explicitly. Other possible edge attributes are weights ( BMDs [BC95] and EVBDD [VPL96, LPV94]) and existential and universal quanti ers (XBDDs [JPHS91]).

*Many decision diagrams restrict how variables may occur. Typical re-* strictions are:

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

*Free : The variable must obey an ordering, but the ordering may be di er-* ent along di erent paths.

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

*4Strictly 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 di erent 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* | *B n 7! B* | *S* | *I; S* | *no* | *order* | *yes* |
| *FBDD* | *B n 7! B* | *S* | *I; S* | *no* | *free* | *no* |
| *GBDD* | *B n 7! B* | *S* | *I; S* | *no* | *none* | *no* |
| *IBDD* | *B n 7! B* | *S* | *I; S* | *no* | *index* | *no* |
| *FDD* | *B n 7! B* | *nD* | *I; Z* | *no* | *order* | *yes* |
| *OKFDD* | *B n 7! B* | *S; pD; nD* | *I; S; Z* | *no* | *order* | *yes* |
| *XBDD* | *B n 7! B* | *S* | *I; S* | *on edges* | *order* | *semi* |
| *ZBDD* | *B n 7! B* | *S* | *I; Z* | *no* | *order* | *yes* |
| *MTBDD* | *B n 7! Z* | *S* | *I; S* | *no* | *order* | *yes* |
| *BMD* | *B n 7! Z* | *nD* | *I; S* | *no* | *order* | *yes* |
| *BMD* | *B n 7! Z* | *nD* | *I; S* | *no* | *order* | *yes* |
| *K BMD* | *B n 7! Z* | *S; pD; nD* | *I; S; Z* | *no* | *order* | *yes* |
| *EVBDD* | *B n 7! Z* | *S* | *I; S* | *no* | *order* | *yes* |
| *HDD* | *B n 7! Z* | *multiple* | *I; S* | *no* | *order* | *yes* |
| *FBD* | *B n 7! B* | *S* | *I; S* | *^;* | *free* | *no* |
| *BED* | *B n 7! B* | *S* | *I; S* | *yes* | *none* | *no* |
| *DDD* | *Rn 7! B* | *S* | *I; S* | *no* | *order* | *semi* |
| *LDD* | *Rn 7! 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 quanti cation. Their idea is to introduce extra variables, so called coding variables, in the BDD. The coding variables are implicitly existentially quanti ed. Vertices contain- ing a coding variable are in e ect OR-vertices since (9s : s ! f; g) $ (f \_ 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* | *Di erence 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 rst 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 at combinational circuits. First we de ne the problem and show how to convert it to the tautology problem for BEDs. Then we discuss di er- ent 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].

## *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 re nement or a modi-

*cation of the previous one. The re nements and modi cations 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 propo-* sitional logic. For a gate-level circuits, the following correspondences apply:

*33*

|  |  |  |
| --- | --- | --- |
| *circuit* |  | *logic* |
| *inputs* |  | *variables* |
| *gates* |  | *connectives* |
| *outputs* |  | *formulae* |

*We only consider combinational circuits without cycles. A cycle poten-* tially 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 Fig- ure 3.1. The circuit has a cycle which acts like a memory element. Assuming that everything is 0 to begin with, the input sequence h00; 01; 10; 11i pro- duces the output sequence h1; 1; 0; 0i while the input sequence h00; 10; 01; 11i produces the output sequence h1; 0; 1; 1i. In the two cases, the outputs for input 11 di er. By disallowing cycles, we avoid such behavior.

*x*



*z*

*y*

*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:

*0 = i1 ^ (i2 ^ i3) ^ (i3 ^ i4)*

*o*

*1*

*0 = i3 ^ i4*

*o*

*1 2 3 3 4*

*2*

*00 = i ^ (i ^ i ) \_ (i ^ i )*

*o*

*1*

*00 = (i2 ^ i3) \_ (i3 ^ i4) ^ i4*

*o*

*2*

*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 diÆcult 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 nal 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 nal hardware circuits. Formal veri cation proves that the design works, but testing examines the physical implementation of the design. Formal veri cation and testing complement each other.

*We consider two circuits equivalent if they have pairwise identical out-* puts 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 rst

*nding 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 $ g.

*The functionality of the two circuits may not be 100% speci ed. Or the* two circuits may be at di erent 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 di erent values. Let c be the characteristic function for the care-set. Then this problem can be cast as the tautology problem for c ! (f $ 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* veri cation problems. For example, when verifying arithmetic circuits by checking that they satisfy a given recurrence equation [Fuj96] or when ver- ifying 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 $ f . To do this we would rst 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 $ 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 $ f is a tautology and convert f $ f to a BDD without converting f to a BDD rst. The following section describes ways of simplifying BEDs based on similar observations.

## *Simpli cations*

*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 di erent vertices) of all the BEDs to be small. In the following sections we describe di erent ways of simplifying BEDs.

* + 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 f^; \_; :g and f^ g 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 di erent connectives, the chance of reuse of vertices is* greater thus reducing the BED size.

*With only a few di erent connectives available, some connectives re-* quire 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, ve can easily be eliminated:* K0, K1, 1, 2, and 2. They can be replaced by terminals, left or right argument, or negation. The remaining 11 connectives fall in four categories:

*Negation : f:g*

*Positive : f\_; !; ; ^ g* Negative : f\_ ; 6!; 6 ; ^g Neutral : f$; g

*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:

*f:; \_; !; ; ^ ; $g : (3.1)*

*Each of the remaining connectives can be expressed using at most one extra*

*negation. For example, a ^ b can be expressed as :(a ^*

*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.

* + 1. *Rewriting*

*Keeping the BEDs reduced, as mentioned in De nition 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 \_g and g \_f are syntactically di erent, but semantically* equivalent. We can increase sharing by always choosing one over the other. In general, we x 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 op1 and op2 are operators such that f op1 g is semantically equivalent to g op2 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 re- moved since for each binary operator op there exists another operator op0 such that f op0 g is equivalent to :f op g. In this way we only need nega- tion 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, :(f $ g) would be written f g even though the operator is not part of the set (3:1).

*We can exploit equivalences like the absorption laws, for example f \_* (f ^ g) = f , and distributive laws, for example (f ^ g) \_ (f ^ h) = f ^ (g \_ 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 pre-* sented as a lter and a transformation. Every time we create a new operator vertex, we match it against the lters. In case of a match, the correspond- ing transformation gives an equivalent, but locally smaller, BED. Since there are only 163 = 4096 combinations of three operators (and even less if we do not allow all 16 di erent operators) it is feasible to tabulate all possible rewriting rules.

*To incorporate rewriting rules in BEDs, we just have to alter the Mk al-* gorithm slightly. Algorithm 3.3 shows the pseudo-code. The only di erence 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 ) Transformation*

*(1) normalization ) g op2 f*

*f op1 g where g < f in the vertex order*

*(2) negation absorption ) f op2 g*

*:f op1 g where op2 is found using identities*

*(3) constant propagation ) 0; 1; f; :f*

*f op1 T depending on op1 and terminal T*

*(4) repeated children ) 0; 1; f; :f*

*f op1 f depending on op1*

*(5) absorption ) f op3 g* f op1 ( f op2 g )

*(6) distributivity 1 ) f op4 g* ( f op2 g ) op1 ( f op3 g )

*(7) distributivity 2 ) ( f op5 g ) op4 h* ( f op2 g ) op1 ( h op3 f ) if possible

*(8) distributivity 3 ) f op4 ( g op5 h )* ( f op2 g ) op1 ( h op3 f ) 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 op1, op2 and op3.*

*Name: Mk( ; l; h)*

*1: if there exists a ( ; l; h) vertex then* 2: return that vertex

*3: else if is a variable and l = h then* 4: return l

*5: else if is operator and ( ; l; h) is in rewriting table then* 6: return Mk(lookup( ; l; h))

*7: else*

*8: return new vertex ( ; l; h)*

*Algorithm 3.3: The modi ed Mk algorithm which includes rewriting rules. The algorithm works just like the old Mk algorithm; see Algorithm 2.3. The only di erence is that before creating new operator vertices, Mk now looks in a rewriting table to nd 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 u be an expanded graph for BED u such that

*u is identical to u except that shared vertices are duplicated so no vertex* has in-degree larger than one. Think of as an operator which turns a DAG into a tree. Consider a BED before (u) and after (u0) the application of a rewriting rule. Then the size of u0 is less than u, 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 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? Ho mann and O'Donnell [HO82] show di erent methods for pattern matching in trees. Similar techniques must be developed for the BED structure.



*u4*

*u2*

*u3*

*u5*

*a*

*b*

*c*

*1*

*0*

*Figure 3.4: Lost sharing of vertices. The u vertices are operator vertices. Assume vertices u2 and u3 are shared with another BED. If we create vertex u1, the BED matches rewrite rule (8). The resulting BED contains two new operator vertices u4 and u5 instead of u1, u2 and u3. But since u2 and u3 are shared, they cannot be removed.*



*u1*

*u2*

*u3*

*b*

*a*

*c*

*1*

*0*

* + 1. *Other Simpli cation Methods*

*In the previous sections we have described two simpli cation methods (op-* erator set restriction and rewriting). But there are other methods available. Here we just mention a few of them.

*The rst method is balancing. Consider the deep and thin BED for* a \_ (b \_ (c \_ a)). The rewriting rules do not work well for such BEDs. Had we placed the parentheses di erently, the rewriting rules would be more e ective: (a \_ b) \_ (c \_ a) transforms to (a \_ b) \_ 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 distri- bution [BCCZ99, BCC+99, BCRZ99], balancing reduced the longest path from 8933 to 21.

*A second simpli cation 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



*\_*

*\_*

*\_*

*a*

*c*

*b*

*0*

*1*



*\_*

*\_*

*\_*

*b*

*a*

*c*

*1*

*0*



*\_*

*\_*

*c*

*b*

*a*

*1*

*0*

*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 St almarck's method. We discuss St almarck's* method in detail later, but here it suÆces to say that the algorithm works on Boolean formulae by extracting knowledge from them under some assump- tion and then extracting knowledge under the negated assumption. The intersection of the two knowledge sets can be inferred without any assump- tions. 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 St almarck'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, St almarck'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 simpli cation method. It exploits the uneven dis-* tribution of 0s and 1s in the positive and the negative connectives. For example, consider the formula f \_ 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 \_x is equivalent to f [0=x] \_x, where f [0=x] is a substitution of 0 for x in f . In general f \_g is equivalent to f [0=g] \_ g, where f [0=g] is a simpli cation of f given that g is

*0. Coudert, Berthet and Madre [CBM89] presented a method for doing such* simpli cations. Their method simpli es a formula f to a smaller formula f 0 such that f and f 0 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.

* + 1. *Experimental Results for Simpli cations*

*In this section we give some experimental results showing the e ects of* operator and rewriting simpli cations. 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 circuits1. First, we run the BED package with no simpli cations (except to ensure that all BEDs are reduced as per De nition 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 veri cation. The Up All algorithm is both faster and uses signi cantly less memory in four out of ve cases. In the last case rewriting rules and the operator set does slightly worse. Note the case of c499 versus c1355. The simpli cations 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 simpli cations are* critical for the performance of Up One. For Up All, the simpli cations generally work well and give an improvement in speed and memory.

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

|  |  |  |  |
| --- | --- | --- | --- |
| *Circuit* | *Up One*  *Runtime Size* | *Up*  *Runtime* | *All*  *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 simpli cations. Run- times 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 com- putation required more than 3 million vertices.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Circuit* | *Up*  *Runtime* | *One*  *Size* | *Up*  *Runtime* | *All*  *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 simpli cations. Run- times 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 simpli cations have di erent e ects on Up One and* Up All, we need to look more closely at the algorithms. During the ini- tial 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 simpli cation of the initial BED. This leads to an interesting use for BEDs: A preprocess- ing 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

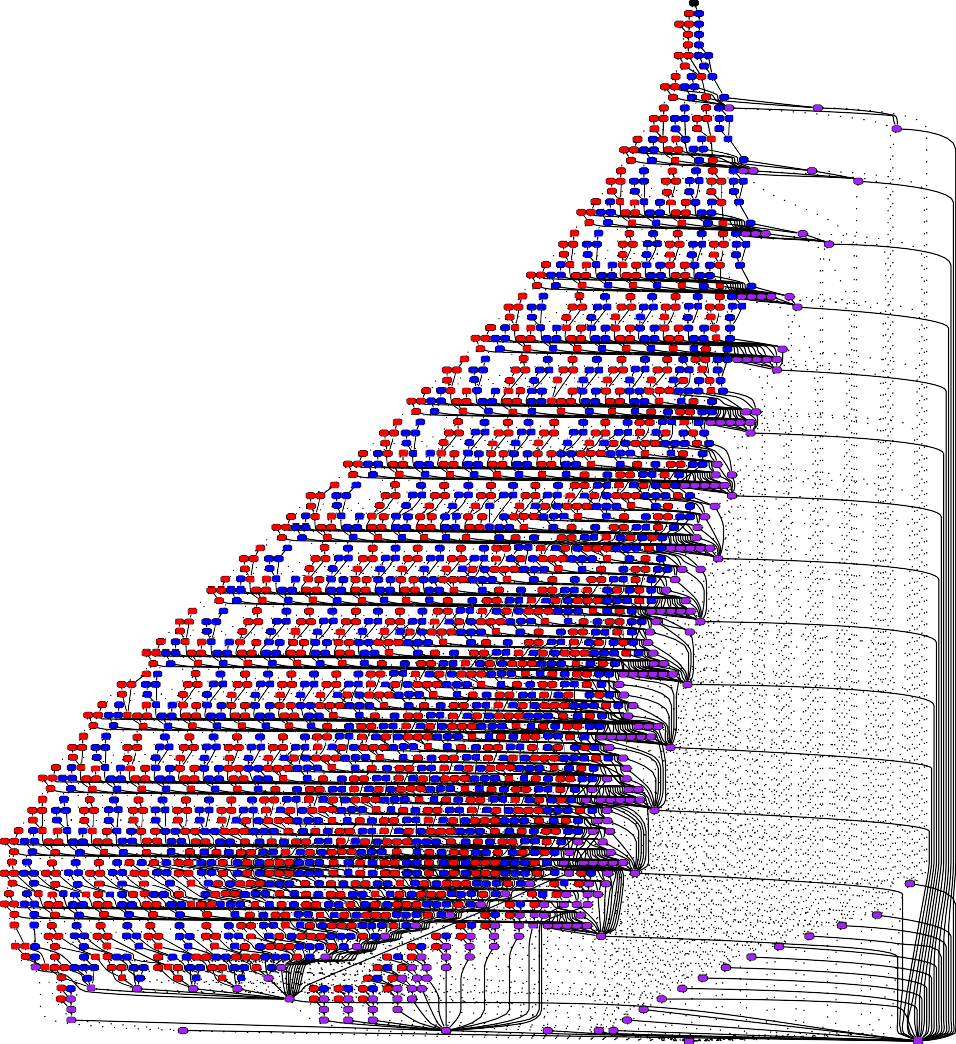
*gure shows that there is not much sharing between the two multipliers as* most vertices are either red or blue. Figure 3.7 shows the same picture, but this time with the rewriting rules turned on. The result is a mere 11 vertices compared to 2448 vertices before. The rewriting rules have eliminated 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).

## *Variable Ordering*

*The eÆciency of Up One and Up All depends on the variable order. Al-* though the initial and nal 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 re- spects all possible orderings. If the veri cation 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: Veri cation 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: Veri cation 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: rst, 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 di erent outputs are merged into a single global variable ordering. We only consider the step of nding a variable ordering for a given primary output, since di erent variable orders can be used for di erent roots of a BED. This allows a greater exibility in nding 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 di erent 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 Observa-* tion 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 di erently than Up All, the variable order-* ing heuristics developed for BDDs may not be e ective 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 veri cation process. Also, a good variable order has the variables that a ect the output the most early in the order. Up One then pull these variables to the root rst which allow the most reductions.

*In the following we present two variable ordering heuristics, originally* developed for BDDs, but which have proven to be e ective 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 di erent results when run on circuits and on BEDs.

*1 1*

*^*

*^*

*^*

*^*

*1=4*

*1=4 1=4*

*1=4*

*1=4*

*1=2*

*1=8 1=8*

*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.*

* + 1. *The Fanin Heuristic*

*A number of variable ordering heuristics are based on a depth- rst traversal* of the circuit [CHP93, FFK88, MWBSV88]. A depth- rst traversal is a sim- ple 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- rst based heuristics di er 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:

*De nition 3.3.1. The depth of a vertex v is de ned recursively as:* depth(v) = 0 : v is a terminal vertex

*m : otherwise*

*where m is*

*max depth low (v) ; depth high(v) +1 :*

*The Fanin heuristic for a BED is shown in Algorithm 3.9. Calling* Fanin(u) determines a variable order 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(juj).

*Name: Fanin(u)*

*Require: The mark array must be initialized to false before calling Fanin.* 1: if mark[u] then

*2: return hi*

*3: else*

*4: mark[u] true*

*5: if u is variable x then* 6: hxi

*7: else if depth(low (u)) > depth(high(u)) then*

*8: Fanin(low (u)) Fanin(high(u)) 9: else*

*b*

*10: Fanin(high(u)) Fanin(low (u)) 11: return*

*b*

*Algorithm 3.9: The Fanin heuristic for determining a variable ordering for BED vertex u. Concatenation of sequences is denoted by the operator . The symbol hi denotes the empty sequence. The algorithm assumes that all variable vertices in the BED u have low child 0 and high child 1.*

*b*

* + 1. *The Depth Fanout Heuristic*

*The Fanin heuristic does not capture that variables that a ect the output* the most should be ordered rst, something which is particularly important for Up One. The Depth Fanout heuristic [Min96], shown for BEDs in- stead of circuits in Algorithm 3.10, attempts to determine the variables that a ect 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 incre-* mented 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 pri- mary 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(jsup(u)j juj) since the loop* (line 3 to 8) is repeated jsup(u)j 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: hi*

*2: S sup(u)*

*3: while S 6= ; do*

*4: propagate value 1.0 downwards from u.* 5: x the variable with the highest value.

*6: hxi*

*b*

*7: remove x from the BED.* 8: S S n fxg

*9: return*

*Algorithm 3.10: The Depth Fanout heuristic for determining a variable order- ing for BED vertex u.*

## *St almarck's Method*

*BED to BDD conversion using Up One and Up All is not the only way* to solve the satis ability and tautology problems for BEDs. St almarck's method [SS98] is a patented algorithm for tautology checking of formulae in propositional logic. In this section we show how to adapt St almarck's method to BEDs.

*The advantage of St almarck'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, St almarck's method is often slower than Up One and Up All.

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



*(i) $*

*(g) \_*

*(h) ^*

*a*

*(d) ^*

*(e) \_*

*(f ) \_*

*b*

*c*

*a*

*b*

*a*

*c*

*Figure 3.11: Parse-tree for a \_ (b ^ c) $ (a \_ b) ^ (a \_ c). Each vertex is labeled with a Boolean variable.*

*f ::= a j :f j f1 f2 ;* where denotes a binary operator from the set:

*f\_; ^; !; ; $; \_ ; ^ ; 6!; 6 ; g :*

*The idea behind St almarck's method is that to prove that f is a tautol-* ogy, we negate f , and then show that :f is unsatis able. 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 \_ (b ^ c) $ (a \_ b) ^ (a \_ c). Figure 3.11 shows* the parse-tree. It gives rise to the following triplets:

*i $ g $ h*

*g $ a \_ d*

*h $ e ^ f*

*d $ b ^ c*

*e $ a \_ b*

*f $ a \_ c*

*St almarck's method works by placing variables in equivalences classes.* Two variables a and b are in the same equivalence class, a 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 :c d. If a variable e is assigned the value true (false) we write e 1 (e 0). We use 0 as an abbreviation of :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 equiva- lence class of 0 to denote the assumption that the formula is unsatis able. St almarck's method merges equivalence classes. If at any point a variable and its negation are placed in the same equivalence class, i.e. a :a, then we have found a contradiction and thus proven the original formula to be a tautology.

*The equivalence relation is, by de nition, transitive, symmetric, and* re exive. It handles negations in the expected way, for example, :a :b implies a b and :a b implies a :b. Moreover, if the relation contains a contradiction then everything relates: a :a implies b c and b :c.

*An equivalence relation gives rise to a set of axioms C:*

*C = f ` x y j x y g ; (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 Au for triplet u $ l h* relate the variables u, l, and h (with and without negations) and 0 and 1

*depending on the connective . For example, in the case of ^, Au are the* rules in Figure 3.12. We denote the union of the rules for all triplets by A.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *` u :h* |  | *` u :l* |  | *` u 1* |  | *` h 0* |
| *` h 1*  *` u :h* |  | *` l 1*  *` l h* |  | *` h 1*  *` u 1* |  | *` u 0*  *` l 1* |
| *` l 0*  *` u :l* |  | *` u h*  *` l :h* |  | *` l 1*  *` h 1* |  | *` u h*  *` l 0* |
| *` h 0* |  | *` u 0* |  | *` u l* |  | *` u 0* |

*Figure 3.12: Rules Au for the triplet u $ l ^ h.*

*The zero-saturation, 0-Sat, of an equivalence relation is a new equiv-* alence relation:

*Sat( ; 0) = f (x; y) j A[ C ` x y g :*

*The zero-saturation of deduces from all possible relations between* triplet variables based on the rules in A. Algorithm 3.13 shows the pseudo- code for 0-Sat. The sign `1 in line 4 indicates application of exactly one rule in Au to one axiom in C. 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( )*

*1: U set of all variables* 2: while U is not empty do 3: remove some u from U

*4: while Au [ C `1 x y and not x y do* 5: update with x y

*6: U U [ P arents(x) [ P arents(y) [ fx; ygnu* 7: return

*Algorithm 3.13: The 0-Sat algorithm for zero-saturation.*

*0-Sat is seldom enough to prove a tautology. St almarck adds the dilemma* rule to his proof system. The dilemma rule states that if we can prove x y under a given assumption u 0, and we can prove the same thing under the negated assumption u 1, then we can prove x y without an assumption

*on u. In other words, if `u 0 and `u 1 then .*

*`x y*

*`x y*

*`x 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 set of all variables* 3: R0 R

*4: for all u in U do*

*5: R1 (k)-Sat(R [ (u 0))*

*6: R2 (k)-Sat(R [ (u 1))*

*7: R R1 \ R2*

*8: until R = R0*

*9: return R*

*Algorithm 3.14: The (k + 1)-Sat algorithm for (k + 1)-saturation.*

*When performing a (k + 1)-saturation, we rst 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.

*De nition 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 \_ (b ^ c) (a \_ b) ^ (a \_ c) :* The right implication gives a 1-hard formula:

*a \_ (b ^ c) ! (a \_ b) ^ (a \_ c) :*

*In the following, we use the terminology \vertex u is assigned the valued*

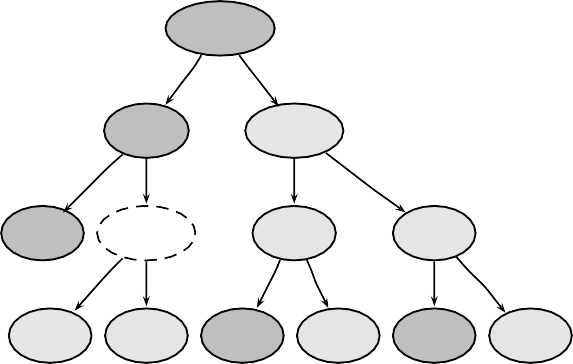
*0 (1)" to mean that the variable for vertex u is placed in the same equivalence* class as 0 (1).

*As an example of St almarck's method, we now prove the left-implication* of the distributive law: a \_ (b ^ c) $ (a \_ b) ^ (a \_ 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 ^ y) \_ (x ^ z) = x ^ (y \_ z) :*



*(i)*

*(g) \_*

*(h) ^*

*a*

*(d) ^*

*(e) \_*

*(f ) \_*

*b*

*c*

*a*

*b*

*a*

*c*

*Figure 3.15: Parse-tree for a \_ (b ^ c) (a \_ b) ^ (a \_ 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 con ict.*

*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.

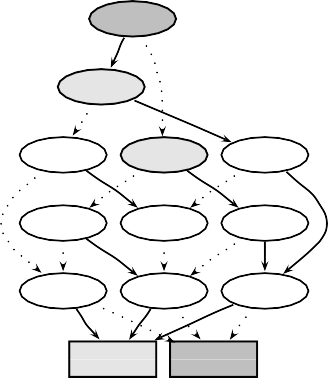
*A zero-saturation of the right BED results in the value 1 being assigned* to the two conjunction vertices. The 1 on the leftmost conjunction vertex propagates to 1 on both the disjunction and the negation vertex. The 1

*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. tu*

*3.4.1 Implementation of St almarck's Method using BEDs*

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



*^*

*\_*

*^*

*\_*

*^*

*^*

*:*

*^*

*a*

*b*

*c*

*1*

*0*



*^*

*^*

*^*

*:*

*\_*

*b*

*c*

*a*

*0*

*1*

*Figure 3.16: Two equivalent BEDs. The right one is obtained from the left one by twice applying the rule (x ^ y) \_ (x ^ z) = x ^ (y \_ 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 con ict.*

*Looking over St almarck's algorithms for saturation of formulae, we iden-* tify 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 modi ed 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 re- peatedly 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 b. If the pointer has a negation mark, then a :b. Disjoint sets gives us transitivity, symmetry, and re exivity 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 eÆcient. 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 Algo-* rithm 3.14 has no in uence on the results of the algorithm. However, it does have an e ect on how fast we obtain the results. We have experimentally found that a most-popular-vertex- rst 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- rst and top-down-depth- rst run slower than most-popular-vertex- rst.

## *Experimental Results*

*In this section we give experimental results for combinational circuit veri -* cation using BEDs. We use the two public benchmark suites ISCAS'85 and LGSynth'912.

*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 veri ed in less memory but at the expense of more frequent garbage collections and thus longer runtimes.

* + 1. *The ISCAS'85 Benchmark Suite*

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

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

*used to test di erent methods of tautology checking and thus it makes com-* parisons 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 veri cation 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 veri cation 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 nding bugs. Table 3.4 shows how many outputs were erroneous.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Circuit* | *Inputs* | *Number of 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 heuris- tic. 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-

|  |  |  |
| --- | --- | --- |
|  | *Number of Outputs* | |
| *Circuit* | *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 di erences 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 di erence. Most of this di erence 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 mil- lion vertices3. 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 veri cation. 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 prob- lematic. 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

*340 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 indi- cates 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 indi- cates 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 di erent veri cation prob-* lem. He does not compare the redundant and the non-redundant versions. Instead the circuits are synthesized and optimized, and he veri es these cir- cuits against the originals. This may well be a more diÆcult veri cation problem.

*Kunz et. al. [KPR96] and Pradhan et. al. [PPC96] use a technique based* on recursive learning combined with BDDs. The reported runtimes are com- parable to the BED runtimes for the smaller circuits, but for the larger cir- cuits 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 func-* tionally 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 ltering technique to verify the circuits. Each circuit passes through a series of lters. Each lter solves part of the veri cation problem. In this way they combine many di erent 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. With-* out variable reordering, CUDD cannot complete seven of the 15 veri cation problems. With the \sift" heuristic, CUDD completes all but one problem: the 16-bit multiplier which is notoriously diÆcult for BDDs. Variable re- ordering has its costs. It now takes longer to complete some veri cation problems. We have performed an out-of-the-box test of CUDD. This means that we have not attempted to nd 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 di erent veri cation problem is solved. \n/a" denotes that the runtime has not been reported.*

* + 1. *The LGSynth'91 Benchmark Suite*

*The LGSynth'91 benchmark suite contains 77 multilevel combinational cir-* cuits 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 veri cation problems, map and opt:

*map: Veri cation of the correctness of the mapping by comparing the original* descriptions to the mapped circuits.

*opt: Veri cation of the correctness of the optimization by comparing the* mapped circuits to the optimized ones.

*The circuits di er 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 diÆcult 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 veri cation 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 veri cation 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 ve 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 veri cation 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

*ve 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 veri ed in 32 MB of* memory and 15 minutes by all four BED methods. Of these 13 problems, only ve are not veri ed by any BED method. Two of the ve are veri ed by CUDD. The remaining 221 problems are veri ed by all four methods; 165 of these are veri ed 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 diÆcult 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 diÆcult* ones. A problem which is diÆcult for CUDD, is also diÆcult 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 prob-* lems (not counting the single one it could not complete). For all six problems, the Fanin / Up All combination was the fastest of the ve 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.)

* + 1. *St almarck's Method*

*To test the e ectiveness of St almarck'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 satis ability test cases from the Second DIMACS Challenge4. These examples do not fall in the category of combinational cir- cuits. However, since Harrison has reported results for his implementation of St almarck's method on these examples, they are well-suited for compar- ison. Harrison rewrote the formulae so they only contained the following Boolean connectives: ^; $; :. We also rewrote the formulae but to the set:

*^ ; \_; !; ; $. At the same time we performed the rewriting rules described* in Section 3.2.2. These di erences explain why our degree of hardness di ers for a couple of the test cases. Our implementation of St almarck's method is the fastest by up to an order or two of magnitude. This is also expected as

*4See* [*http://mat.gsia.cmu.edu/challenge.html.*](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 veri ed in less than ve seconds by all four BED methods and by CUDD (with the \sift" variable reordering heurstic).*

*Circuit*

*apex6-opt C1355-map C1355-opt C1908-map C1908-opt C2670-map C2670-opt C3540-map C3540-opt C432-map C432-opt C499-map C499-opt C5315-map C5315-opt C6288-map C6288-opt C7552-map C7552-opt C880-opt dalu-map dalu-opt des-map des-opt frg2-map frg2-opt i10-map i10-opt i7-map*

*i7-opt i8-map i8-opt i9-map i9-opt k2-map k2-opt pair-map pair-opt rot-map rot-opt*

*too large-map too large-opt*

*Up One*

*Fanin Depth Fanout*

*2.4 5.8*

*8.7 9.3*

*11.9 12.0*

*5.0 3.6*

*4.9 4.2*

*2.6 17.3*

*2.7 68.2*

*46.7 53.0*

*28.8 62.8*

*9.6 10.2*

*8.4 9.0*

*8.7 9.4*

*13.3 12.7*

*50.7 32.5*

*119.0 33.8*

*0.6 1.2*

*- -*

*8.7 41.4*

*9.3 43.6*

*2.1 3.3*

*1.7 13.3*

*1.8 10.7*

*6.8 62.0*

*6.6 64.3*

*3.4 14.1*

*3.2 13.2*

*35.4 151.9*

*37.8 154.2*

*1.9 5.6*

*2.0 5.9*

*3.0 19.3*

*2.8 16.8*

*2.2 9.1*

*2.4 9.1*

*1.7 7.1*

*1.8 7.2*

*4.1 19.9*

*4.4 21.7*

*4.0 13.2*

*3.0 9.7*

*19.1 4.6*

*1.4 1.4*

*Up All*

*Fanin Depth Fanout*

*2.4 5.6*

*2.0 4.1*

*3.1 6.4*

*3.1 3.7*

*2.5 3.7*

*1.3 14.2*

*1.2 13.6*

*30.5 66.2*

*13.8 46.9*

*4.6 5.5*

*4.8 5.3*

*2.0 4.1*

*2.9 6.4*

*3.1 29.9*

*3.2 31.4*

*0.5 1.3*

*- -*

*2.2 36.6*

*2.2 37.9*

*1.0 2.3*

*1.1 13.1*

*1.1 10.2*

*5.4 58.8*

*5.4 61.3*

*2.9 13.3*

*2.8 12.3*

*8.3 112.1*

*8.3 106.0*

*1.8 5.4*

*1.8 5.7*

*2.5 19.3*

*2.4 16.8*

*1.8 9.1*

*1.9 9.0*

*1.6 7.1*

*1.8 7.2*

*3.1 19.7*

*3.2 20.6*

*2.2 11.5*

*2.0 8.8*

*0.9 3.6*

*0.5 1.2*

*CUDD*

*(sift)*

*0.5*

*20.9*

*34.2*

*7.0*

*5.0*

*6.9*

*9.5*

*89.3*

*75.6*

*0.5*

*0.6*

*17.2*

*18.8*

*3.0*

*3.5*

*-*

*- 30.0*

*30.8*

*11.9*

*2.6*

*2.8*

*7.3*

*6.9*

*0.7*

*0.8*

*56.2*

*41.7*

*0.1*

*0.1*

*4.0*

*2.5*

*0.5*

*0.3*

*1.1*

*1.0*

*8.1*

*8.1*

*3.7*

*2.9*

*3.6*

*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 veri cation problems are shown. Of the 154 veri cation 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*  *sbc-map* | *s832-opt*  *sbc-opt* | *s838.1-map* | *s838.1-opt* |

*Table 3.11: The 50 sequential LGSynth'91 circuit problems which are veri ed in less than ve 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 veri cation time.

*The Up One algorithm allows for a gradual transformation of a BED* into a BDD. By breaking o this transformation and applying St almarck'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- rst 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 St almarck's method does not* compare favorably in terms of runtime with the other methods. However, St almarck's method uses only a few megabytes of memory, and the amount of memory is linear in the size of the circuit (given a xed saturation level).

*Circuit*

*bigkey-map bigkey-opt clma-map clma-opt clmb-map clmb-opt dsip-map dsip-opt mm30a-map mm30a-opt mm9a-map mm9a-opt mm9b-map mm9b-opt mult32a-map mult32a-opt s13207.1-map s13207.1-opt s1423-map s1423-opt s15850.1-map s15850.1-opt s38417-map s38417-opt s38584.1-map s38584.1-opt s5378-map s5378-opt s9234.1-map s9234.1-opt*

*Fanin*

*114.6*

*107.3*

*11.4*

*179.9*

*5.8*

*176.8*

*88.9*

*90.1*

*-*

*-*

*5.5*

*7.5*

*37.8*

*33.5*

*5.8*

*6.0*

*161.6*

*176.5*

*7.2*

*11.8*

*698.8*

*696.9*

*-*

*-*

*-*

*- 26.1*

*25.1*

*-*

*-*

*6.0*

*5.9*

*2.9*

*5.2*

*1.7*

*4.0*

*5.5*

*5.6*

*-*

*- 143.4*

*123.1*

*-*

*-*

*1.4*

*1.3*

*5.4*

*6.3*

*1.9*

*2.5*

*15.8*

*19.2*

*-*

*- 22.4*

*29.3*

*3.5*

*3.4*

*4.5*

*5.7*

*115.0*

*119.3*

*10.7*

*181.3*

*5.6*

*177.5*

*84.5*

*87.0*

*-*

*-*

*3.9*

*5.5*

*331.3*

*28.7*

*5.1*

*5.1*

*145.5*

*159.4*

*7.0*

*10.5*

*695.2*

*680.0*

*-*

*-*

*-*

*- 24.3*

*24.1*

*127.2*

*127.3*

*CUDD*

*(sift)*

*3.0*

*2.7*

*4.5*

*8.1*

*4.1*

*8.3*

*3.5*

*3.5*

*146.9*

*21.2*

*2.1*

*6.2*

*2.7*

*2.8*

*0.6*

*0.5*

*21.3*

*23.3*

*1.2*

*1.3*

*22.7*

*42.8*

*-*

*- 62.6*

*77.1*

*1.3*

*0.7*

*3.6*

*5.0*

*Up One Up All*

*Depth Fanout Fanin Depth Fanout*

*10.0*

*10.3*

*3.9*

*9.2*

*2.1*

*7.1*

*9.3*

*9.3*

*-*

*-*

*-*

*-*

*-*

*- 3.00*

*2.9*

*19.1*

*20.5*

*6.5*

*7.3*

*46.6*

*88.0*

*-*

*- 175.0*

*181.4*

*5.2*

*4.2*

*15.6*

*18.9*

*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 veri cation problems are shown. Of the 80*

*veri cation 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*

*aim-50-1 6-no-1* aim-50-1 6-no-2 aim-50-1 6-no-3 aim-50-1 6-no-4 aim-50-2 0-no-1 aim-50-2 0-no-2 aim-50-2 0-no-3 aim-50-2 0-no-4 aim-100-1 6-no-3 aim-100-2 0-no-1 aim-100-2 0-no-2 dubois20

*jnh211* ssa0432-003

*BED+St almarck Harrison Degree of*

*[sec] [sec] hardness*

*3.1 8.2 2 / 2*

*0.4 14.1 1 / 2*

*1.2 15.3 1 / 1*

*0.4 5.9 1 / 1*

*0.9 20.5 1 / 1*

*0.5 28.7 1 / 1*

*0.6 13.9 1 / 1*

*0.6 29.1 1 / 1*

*2.4 135.3 1 / 2*

*2.4 12.9 1 / 1*

*2.5 14.6 1 / 1*

*66.9 377.0 2 / 2*

*15.0 1762.0 1 / 1*

*8.5 490.1 1 / 1*

*Table 3.13: Boolean satis ability test cases from the Second DIMACS Challenge. The test cases, all being not satis able, 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 ( rst number) and for Harrison's (second number).*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Circuit* | *St almarck Up One / Up*  *Hardness Runtime Runtime* | | | *All* |
| *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 St almarck'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.*

|  |  |  |
| --- | --- | --- |
|  | *No Minimize* | *With Minimize* |
| *ISCAS'85* | *[sec]* | *[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: Veri cation results for St almarck's method on the ISCAS'85 bench- mark suite with and without minimizing the formulae rst. 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 St almarck's method in a BED* framework is the additional availability of BED and BDD methods for tau- tology checking. Standard BED and BDD methods typically run out of memory before they run out of time. St almarck's method has it the other way around. Combining St almarck's method with BEDs gives the user the choice between time and memory.

## *Related Work*

*Current approaches for equivalence checking of combinational circuits can* be classi ed 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 dia- grams 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 veri cation 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 di erence 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 nd a suÆciently 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 nding logical implications between nodes in the circuits by Kunz et. al. [Kun93, KP94]. To enable the veri cation 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 nding cuts in the circuits to split the ver- i cation problem into more manageable sizes. Recursive learning is closely related to St almarck'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 di erence between St almarck'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 de-* termined whether one node is functionally equivalent to another. If two nodes are found to be identical, they are replaced with a new, free vari- able. 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 repre- sents 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 perfor- mance 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 bene t signi cantly from replacing the used circuit representation with BEDs.

*BEDs can be seen as an intermediate representation between the com-* pact circuits and the canonical BDDs. Compared to the functional tech- niques, 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 nd equiv- alences between pairs of nodes (since only local operator reduction rules are included). Combining BEDs with structural techniques would be bene cial since information about equivalent nodes immediately reduce the size of the BED and make even further identi cations of nodes possible.

*Richards [Ric98] has some interesting notes on a tautology checker loosely* related to St almarck's algorithm. Whereas St almarck's method uses rela- tions 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 develop-* ing eÆcient satis ability 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 eÆcient for typical AI problems, they are quite inferior to BDDs on circuits [US94]. Our own experiments with St almarck's method support this. However, satis ability checkers are very memory eÆ-

*cient so one should not completely discard them.*

## *Conclusion*

*In this chapter we have shown how to use BEDs in equivalence checking* of at 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 im-* portant 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 St almarck's method for solving the tautology prob-* lem. We discussed the method and illustrated how to implement it on top of the BED data structure. The method works for combinational circuit veri cation, but it is much slower than Up One and Up All. However, St almarck's method uses little memory.

*Finally, we experimentally tested the BEDs on a number of combina-* tional circuit veri cation problems. The BEDs compare favorably with other methods both in terms of time and memory usage. Of 234 equiva- lence checking problems from the LGSynth'91 suite, only ve could not be veri ed when using 32 MB of memory. The veri cation 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 prob- lems 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 com-* binational 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].

## *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 gener- ated 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 modi cations of a circuit to ensure that the designer has not introduced errors.

*We use a hierarchical model of combinational circuits as opposed to* the at 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

*77*

*cell fa(in x0; x1; x2; out u0; u1) f* u0 := x0 x1 x2

*u1 := ( x0 ^ x1) \_ (x2 ^ (x0 \_ x1 ) )*

*g*

*cell 4bitadder (in s0;::: ; s8;*

*out s9; s11; s13; s15; s16) f* var s10; s12; s14

*instance fa(s1; s2; s0; s9; s10)* instance fa(s3; s4; s10; s11; s12) instance fa(s5; s6; s12; s13; s14) instance fa(s7; s8; s14; s15; s16)

*g*

*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 veri cation. 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 atten them into a single block of combinational logic on which the veri cation 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 gure is also a 4-bit adder, but with two instantiations of two di erent 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 di erent 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 veri cation 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 veri cation would reuse the comparisons to prove the equivalence.

s0

s1 s9

s2

+

s10

s3 s11

s4

+

s12

s5 s13

s6

+

s14

s7

s8

+

H 0

t0

t1 t2

H 1

t9

H

2

+

H

3

t10

H

t3 + t11

t4

t12

t5 t13

t6

+

t14

t7

t8

+

s15 s16

4

t15 t16

*Figure 4.2: Two 4-bit adders.*

## *Hierarchical Combinational Circuits*

*Most circuit description languages, for example Berkeley Logic Interchange* Format (BLIF), contain language constructs for modular circuit descrip- tions. Modules may contain other modules yielding a hierarchical descrip- tion. 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. De nition 4.2.1, 4.2.2, and 4.2.3 de ne a mathematical model for hierarchical combinational circuits based on these observations.

*De nition 4.2.1 (HCC). A hierarchical combinational circuit (HCC) is a* pair (C; c), where C is a set of cells and c 2 C is the top cell.

*For example, Figure 4.1 describes an HCC (C; c), where C = ffa; 4bitadder g* and c = 4bitadder .

*De nition 4.2.2 (Cell). A cell c has the following attributes:* Vars(c), a set of Boolean variables,

*In(c) Vars(c), a list of input variables1,* Out (c) Vars(c), a list of output variables,

*and either*

*1We 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) ! 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 fx0; x1; x2; u0; u1g. In(fa) is the list [x0; x1; x2], and Out(fa) is the list [u0; u1]. The full-adder cell has the Fct attribute; a list with two elements where the rst element is the function for the u0 output: x0 x1 x2. The 4bitadder cell has the Inst attribute; a list of four instantiations of fa.

*De nition 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 2 Inst(par(i))),* in(i) Vars(par(i)), a list of input variables,

*out(i) Vars (par(i)), a list of output variables.*

*The instantiation i must further ful ll the requirements:*

*jin(i)j = jIn(cell(i))j and jout(i)j = jOut(cell(i)j :*

*The topmost instantiation of a full-adder cell in the 4-bit adder exam-* ple has cell(i) = fa, par(i) = 4bitadder , in(i) = [s1; s2; s0], and out(i) = [s9; s10].

*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) =*

*^*

*k=1;:::;jOut(c)j*

*(Out (c)k $ 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:

*(u0 $ x0 x1 x2) ^*

*(u1 $ (x0 ^ x1) \_ (x2 ^ (x0 \_ x1))) :*

*For container cells, Rel is determined as:*

*Rel(c) = 9v2V :*

*^*

*i2Inst(c)*

*Rel(cell(i))[M ap] ;*

*where V is the set variables which are neither inputs nor outputs, i.e.,*

*V = Vars(c)n(In(c) [ Out(c)), and [M ap] is a renaming of In(cell(i)) and* Out (cell(i)) variables to in(i) and out(i) variables, respectively. The nota- tion 9v2V for V = fv1; v2;::: ; vkg is shorthand for 9v1 : 9v2 : : 9vk.

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

*We now de ne a path and a cut in a cell.*

*De nition 4.2.4 (Path). For a container cell c, a path p = hp1;::: ; pni is* a sequence of variables from Vars(c) such that for all k, 1 k < n, there exists an instantiation i 2 Inst (c), such that pk 2 in(i) and pk+1 2 out(i).

*De nition 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 = hp1;::: ; pni through c with p1 2 In(c) and pn 2 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 fs3; s4;::: ; s10g is a cut in the 4bitadder container cell. For two* cuts in di erent HCCs we de ne a cut-relation H as a relation over the values of the variables in the cuts.

*De nition 4.2.6 (Cut-relation). A cut-relation H between two cuts K1* and K2 in two cells is a relation over the values of the variables in K1 and K2, i.e., H B K1 [K2 .

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

*be:*

*^ (si $ ti) ^ ^*

*i=3;4;7;8*

*i=0;1;2;5;6*

*(si $ :ti) ; (4.1)*

*stating that si and ti have identical values for i = 0; 1; 2; 5; 6, and that si* and ti have opposite values for i = 3; 4; 7; 8.

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

*Given a cut-relation H for two cuts K1 and K2, and two instantiations* i1 and i2 such that the input variables of i1 and i2 are subsets of K1 and K2, respectively, we can determine a relation Rin between the input variables of i1 and i2:

*Rin = (9v2(K1[K2)n(in(i1)[in(i2)) : H)[M ap] ; (4.2)*

*where [M ap] is a renaming of in(i1), in(i2), out(i1), and out(i2) variables to* In(cell(i1)), In(cell(i2)), Out (cell(i1)), and Out(cell(i2)) variables, respec- tively.

## *Cut Propagation*

*Given two hierarchical combinational circuits HCC1 (C1; c1) and HCC2* (C2; c2), and an input relation Hin, the veri cation problem we consider is to determine whether the outputs satisfy a desired relation Hout. Typi- cally, Hin and Hout would represent \the circuits have identical inputs and outputs."

*The veri cation algorithm works by propagating a cut-relation from the* inputs to the outputs. Let H0 be the input relation Hin, a cut-relation between the input cuts of c1 and c2. We move their cut-relation past instan- tiations of cells in c1 and c2 (assuming that c1 and c2 are container cells). In each step we calculate a new cut-relation, Hk+1, based on the previous one, Hk. When the cut-relation has reached the outputs, the resulting cut- relation, Hn, relates the outputs of c1 to the outputs of c2. If Hn is a subset of Hout, Hn Hout, the circuits have the desired output relation.

* + 1. *Example*

*Before describing the algorithm in detail, we give an example to illustrate* the basic ideas. Consider again the two di erent 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 H0 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 Rin;1 between the leftmost full-adder cell in each of the two circuits using (4.2):

*Rin;1 = (x0 $ y0) ^ (x1 $ y1) ^ (x2 $ y2) : (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 Rin;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):

*Rout;1 = (u0 $ :v0) ^ (u1 $ :v1) : (4.4)*

*We move the cuts and determine the new cut-relation H1 based on Rout;1* (again, we will show later how to do this):

*H1 =*

*i=5;6(si $ ti) ^ (4.5)*

*i=3;4;7;8;9;10(si $ :ti) :*

*V V*

*H2 = Vi=5;6;11;12(si $ ti) ^ Vi=7;8;9(si $ :ti) : In a similar way we propagate the cuts one step further getting H2:*

*In the third step, we start by nding the input relation Rin;3:* Rin;3 = (x0 $ y0) ^ (x1 $ y1) ^ (x2 $ y2) :

*This is identical to the relation Rin;1 from the rst step. The full-adders in* the third step are also identical to the full-adders in the rst step, and thus we can immediately reuse the output relation Rout;1 instead of calculating Rout;3. We update the H2 relation using Rout;1 and obtain H3:

*H3 = (s9 $ :t9) ^ (s11 $ t11) ^*

*(s13 $ :t13) ^ (s14 $ :t14) :*

*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 H3, and get the nal output relation H4:

*H4 = (s9 $ :t9) ^ (s11 $ t11) ^ (s13 $ :t13)*

*^ (s15 $ t15) ^ (s16 $ t16) :*

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

* + 1. *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 Rin between the inputs of the two cells and from that calculate the output relation Rout 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; c1; c2)*

*Require: c1 and c2 are container cells* 1: (K1; K2) input cut for (c1,c2)

*2: while K1; K2 are not output cuts do* 3: Select method

*4: if method is build instantiation i 2 Inst(c1) then* 5: (H; K1) Build(i; H; K1)

*6: else if method is build instantiation i 2 Inst(c2) then* 7: (H; K2) Build(i; H; K2)

*8: else if method is propagate instantiations i1 2 Inst(c1) and i2 2*

*Inst(c2) then*

*9: (H; K1; K2) Propagate(i1; i2; H; K1; K2) 10: return H*

*Algorithm 4.3: The Prop(H; c1; c2) algorithm. H is a input relation between container cells c1 and c2. Prop returns the output relation for c1 and c2.*

*Prop(H0; 4bitadder 1; 4bitadder 2) calculates the output relation H4, where* 4bitadder 1 and 4bitadder 2 are the two di erent 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 ve argu- ments; two instantiations i1 and i2, two cuts K1 and K2, 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) K

*1: M ap map In(cell(i)) to in(i) and Out(cell(i)) to out(i)* 2: R Rel(cell(i))[M ap]

*3: H0 9v2in(i) : H ^ R 4: K0 K [ out(i)nin(i) 5: return (H0;K0)*

*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(i1; i2; H; K1; K2)* Require: in(i1) K1 and in(i2) K2

*1: Rin input relation between i1 and i2 based on H using (4.2)* 2: if memorized (Rin; cell(i1); cell(i2)) then

*3: Rout memorized result* 4: else

*5: either Rout Prop(Rin; cell(i1); cell(i2))*

*6: or Rout 9v2In(cell(i1 ))[In(cell(i2 )) : Rin ^ Rel(cell(i1)) ^ Rel(cell(i2)) 7: memorize (Rin; cell(i1); cell(i2); Rout)*

*8: M ap map Out (cell(i1)) to out(i1) and Out(cell(i2)) to out(i2)*

*9: H0 (9v2In(cell(i ))[In(cell(i )) : H) ^ Rout[M ap]*

*1 2*

*10: K0 K1 [ out(i1)nin(i1)*

*1*

*11: K0 K2 [ out(i2)nin(i2)* 12: return (H0;K0 ;K0 )

*2*

*1 2*

*Algorithm 4.5: The Propagate(i1; i2; H; K1; K2) algorithm. Both i1 and i2 are instances of container cell. H is a cut-relation over the cuts K1 and K2. The algorithm moves the cuts past the cell instances and updates the cut-relation ac- cordingly.*

*assumed that the input variables of the cell instantiations i1 and i2 belong* to the cuts K1 and K2, respectively.

*In line 1 Propagate calculates, using (4.2), the input relation Rin be-* tween i1 and i2 based on the cut-relation H. The input relation Rin is described in cell variables, not in instantiation variables. Next, we calculate the output relation Rout for i1 and i2. If we have previously propagated a similar cut past the same cells, we reuse the previous result (line 3). Other- wise we have two ways of calculating Rout. If both i1 and i2 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 Rout from the input/output relation Rel for each of the instantiations i1 and i2 (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 up-* dated cut-relation H1 (4.5) in line 10.

* + 1. *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 rela-* tion 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 corre- sponds to the standard veri cation 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 in- stantiations 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 function- ality 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 Rin between the cell inputs. Consider the two equivalent circuits in Figure 4.6. The input cut for the top circuit is K1 = fs0; s1g and for the bottom circuit it is K2 = ft0; t1g. Let H0 be the

s0

neg

s2

xor

s3

s1

H 0 H1 H 2

xor

neg

t2

t0

t3

t1

*Figure 4.6: Two combinational circuits. The relation H0 between the wires in the input cuts is (s0 $ t0) ^ (s1 $ t1).*

*cut-relation (s0 $ t0) ^ (s1 $ t1) when calling Propagate. We want to* move the cuts past the negation cells. The new cuts contain the variables s1 and s2, and t0 and t2. We build the input relation Rin for the two negation cells:

|  |  |  |
| --- | --- | --- |
| *Rin* | *=* | *(9v2(K1[K2)n(in(i1)[in(i2)) : H0)[M ap]* |
|  | *=* | *(9v2fs1 ;t0g : (s0 $ t0) ^ (s1 $ t1))[M ap]* |
|  | *=* | *1 ;* |

*where i1 and i2 are instantiations of the two negation cells. In this case Rin* evaluates to 1, meaning that the inputs are unrelated; knowing the value of s0 does not imply a particular value of t1, i.e., they are unrelated. Rin = 1 results in H1 also being 1 and not the expected (s1 $ :t2) ^ (:s2 $ t0).

*The problem is that H0 does not relate the cut variables s0 and t1. In* general we can state that Propagate(i1; i2; H; K1; K2) works without loss of information if the cut-relation H can be split in two parts: one part containing the variables in M = in(i1) [ in(i2) and one part containing the remaining variables:

*H () (9v2M : H) ^ (9v2(K1[K2 )nM : H) (4.6)*

*If H can be written as in (4.6), Propagate determines the exact cut-* relation H0. Otherwise, it gives a conservative approximation to the output relation: Hexact Happrox.

## *Experimental Results*

*We have built hierarchical adder and multiplier circuits of di erent 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 di erent types of full-adder cells: one full-adder outputting a negated carry-out, and one receiving a negated carry-in signal. The veri cation 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 log2(n) for n-bit adders.

*Using the standard technique of attening the two circuits and con-* structing a BDD for each of them, it is possible to get results comparable to those in Table 4.1 for the veri cation 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 us- ing 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 modi cations cause the runtimes to increase signi cantly.

*While adders are easy to handle using BDDs, multipliers are notoriously* diÆcult. We construct multipliers as series of adders and shifters. From the two di erent types of adders in the previous experiment, we create two di erent types of multipliers. The veri cation 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 simultaneously2. 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.

## *Related Work*

*A traditional way of verifying two hierarchical combinational circuits is to*

*rst expand them to at circuits and then use standard equivalence checking* methods for at circuits. In Chapter 3 we have studied the combinational logic-level veri cation problem for at circuits using Boolean Expression Di- agrams. 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 at equivalence check- ing 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 calcu- lating 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.

## *Conclusion*

*We have presented a method based on cut-propagation for obtaining a rela-* tion between the outputs of two hierarchically speci ed combinational cir- cuits. The key new feature of the method is its ability to exploit the hier- archy in the circuit description to reuse previously calculated results in the veri cation. We have demonstrated the power of the method by verifying large adders and multipliers.

*2For an n-bit unsigned multiplier, the greatest result is (2n 1)2, which is less than 22n 1, where 22n 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 con- tainer 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: i si $ ti. Using BDDs, such a cut-relation can be represented very eÆciently. An interleaved variable ordering with si and ti

*V*

*next to each other gives a very compact BDD representation. However, if* si and ti 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 xed- point algorithms, and we use both BDDs and SAT-solvers to perform sat- is ability 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].

## *Introduction*

*Symbolic model checking has been performed using xed-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 repre- senting 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 xed-point iteration, they construct formulae for possible counterexamples and use SAT-solvers to prove or disprove the existence of such counterexam- ples. Abdulla et. al. [ABE00] also use SAT-solvers but keep the xed-point iterations.

*In this chapter we combine BDDs and SAT-solvers in symbolic model* checking based on xed-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.

*91*

*Determining satis ability 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 di erent methods for satis ability checking: (1) SAT-solvers* like Grasp1 [MSS99] and Sato2 [Zha97], and (2) conversion of BEDs to BDDs. BDDs are canonical and thus we can check for satis ability in con- stant time. We perform symbolic model checking the classical way with

*xed-point iterations. One of the key elements of our method is the quan-* ti cation by substitution rule: 9y : g ^ (y $ f ) $ g[f =y]. The rule is used (1) during xed-point iterations, (2) while deciding whether an initial set of states is a subset of another set of states, and nally (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 quanti cation 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 xed- 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 speci cation 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 quanti cation by substitution rule helps remove some of the quan-* ti cations. However, the remaining quanti cations 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 quanti cation of the remaining variables by simply leaving the variables in the formula.

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

*1 Grasp version September 1999. See* [*http://algos.inesc.pt/*](http://algos.inesc.pt/) *jpms/grasp for more information.*

*2 Sato version 3.2. See* [*http://www.cs.uiowa.edu/*](http://www.cs.uiowa.edu/) *hzhang/sato.html for more in-*

*formation.*

## *Theory of Model Checking*

*In this section we describe model checking. We rst use set theory as the* underlying theory. Then we rede ne model checking in terms of Boolean functions3. The systems we consider are nite state machines (FSMs) rep- resented by nite Kripke structures [HC74, CGP99].

*De nition 5.2.1 (Finite Kripke Structure). A nite Kripke structure* M is a tuple (SM ; IM ; T M ; `M ) with a nite set of states SM , a set of initial states IM SM , a transition relation T M SM SM , and a labeling of the states `M : SM 7! P(A) with atomic propositions A.

*Consider the SMV [McM93] program and the accompanying Kripke* structure in Figure 5.1. There are four states in the Kripke structure rep- resented by two state variables s1 and x1. Only the s1 variable is restricted through the init and next statements. The x1 variable is unrestricted. We can interpret the system as follows: x1 is a variable which signals the presents (x1 is true) or absence (x1 is false) of an input which toggles s1.

*MODULE main VAR*

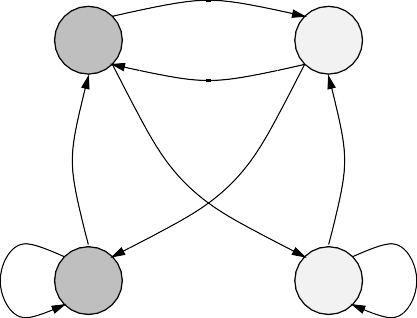
*s1 : boolean; x1 : boolean;*

*ASSIGN*

*init(s1) := 1;*

*next(s1) := !(s1 <-> x1);*

*(s1; x1) (:s1; x1)*



*(s1; :x1) (: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 modi-

*ed de nition of Kripke structures that allows us to distinguish between the* two kinds of variables.

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

*De nition 5.2.2 (Modi ed Kripke Structure). A modi ed Kripke struc-* ture M is a tuple (S; X; I; T; `) with a nite set of states S, a nite set of inputs X, a set of initial states I S, a transition function T : S X 7! S, and a labeling of the states ` : S 7! P(A) with atomic propositions A.

*The transition relation now becomes a transition function which we as-* sume is de ned for all (s; x) 2 S X. This has the e ect that a modi ed Kripke structure with an empty set of inputs X is deterministic. We call a modi ed 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 modi ed Kripke structure for the same SMV pro-* gram as in Figure 5.1.

*:x1*

*x1*

*:x1*



*s1*

*:s1*

*x1*

*Figure 5.2: A modi ed Kripke structure. The initial state is dark-gray.*

*Given a modi ed Kripke structure M = (S; X; I; T; `), we can obtain an* equivalent Kripke structure M 0 = (SM ;IM ;T M ; `M ) in the following way:

*SM = S X* IM = I X

*T M = f (s1; x1); (s2; x2) j*

*s1; s2 2 S and x1; x2 2 X and s2 = T (s1; x1) g*

*`M (s; x) = `(s) where (s; x) 2 SM*

*In the following, we use the term \Kripke structure" to refer to the modi ed* version.

*The transition function T speci es all the possible behaviors of a system.* An FSM can only move from state si to state sj if there is an input x such that sj = T (si; x). We say that the FSM takes a transition from si to sj on input x.

*De nition 5.2.3 (Transition). Let M = (S; X; I; T; `) be a Kripke struc-* ture. There is a transition from state si 2 S to state sj 2 S if:

*9x 2 X : sj = T (si; x) :*

*We write si x*

*sj to indicate a transition from si to sj on input x. If*

*the input does not matter or is implicitly understood, then we just write* si sj.

*We say there is a path from state si to state sj if there is a series of* successive transitions leading from si to sj.

*De nition 5.2.4 (Path). Let M = (S; X; I; T; `) be a Kripke structure.* A path from a state s is an in nite sequence hs1; s2;:: :i of states in S such that:

*s = s1 , and*

*8i 1 : 9x 2 X : si+1 = T (si; x) :*

*We write si k sj to indicate that there is a path of length k between* states si and sj, i.e., there is an in nite path hs1; s2;:: :i such that si = s1 and sj = sk+1. The length indicates the number of transitions required of the FSM to move from state si to state sj.

*Since the transition function is de ned for all (s; x) 2 S 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 nite paths can be thought of as pre xes of in nite paths.

*An in nite path is sometimes called a computation.*

* + 1. *Computation Tree Logic*

*Computation Tree Logic (CTL) [CES86] is a temporal logic used to describe* the speci cation of a nite state machine. There are a number of di erent temporal logics, but we shall only consider CTL.

*Given a state in a Kripke structure M and a CTL speci cation for M ,* then 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 [[ ]].

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

*De nition 5.2.5 (Model Checking). Let M = (S; X; I; T; `) be a Kripke* structure and let be a CTL speci cation for M . We say that M models

*, and write M j= , if I [[ ]].*

*A CTL formula contains a propositional logic part with constants, nega-* tion, conjunction and disjunction. Furthermore, it contains a number of temporal operators each consisting of a path quanti er and a path operator. There are two path quanti ers: E (\there exists an in nite path") and A (\for all in nite paths"). There are ve 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 quanti er and all paths mentioned originate from the current state):

*EX : holds in the next state along some path.* EG : holds on all states along some path.

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

*EU( 1; 2) : On some path, 1 holds on all states until 2 holds4.*

*ER( 1; 2) : On some path, 2 holds until 1 releases 2 (i.e., 2 holds on* the rst state in which 1 holds).

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

*f ::= 0 j 1 j a j :f j f \_ f j f ^ f j EX f j EG f j EU(f; f ) ;* where a is an atomic proposition.

*4We use the notion of strong until where 2 has to hold eventually. There is also a weak until in which 2 does not have to hold if 1 holds on all states along the in nite path.*

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

|  |  |  |
| --- | --- | --- |
| *AX*  *EF AF AG* |  | *:EX :*  *EU(1; )*  *:EG :*  *:EF :* |
| *AU( 1; 2)* |  | *:EU(: 2; : 1 ^ : 2) ^ :EG : 2* |
| *ER( 1; 2)* |  | *:AU(: 1; : 2)* |
| *AR( 1; 2)* |  | *:EU(: 1; : 2)* |

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

*De nition 5.2.7 (Set Transformer). A set transformer is a function*

*: 7! , where is some set.*

*is monotonic if P1 P2 implies (P1) (P2).*

*In model checking of a Kripke structure M = (S; X; I; T; `), we use* set transformers of the type : P(S) 7! P(S). Algorithm 5.3 shows the pseudo-code for the least xed-point operator Lfp and the greatest xed- point operator Gfp, respectively.

*Name: Lfp [ ]*

*Require: must be monotonic.* 1: Q ;

*2: Q0 (Q)*

*3: while Q 6= Q0 do*

*4: Q Q0*

*5: Q0 (Q0)*

*6: return Q*

*Name: Gfp [ ]*

*Require: must be monotonic.* 1: Q S

*2: Q0 (Q)*

*3: while Q 6= Q0 do*

*4: Q Q0*

*5: Q0 (Q0)*

*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 xed-point and greatest*

*xed-point, respectively, for it.*

*Lemma 5.2.8 (Increasing). The sequence of Q's (Q0; Q1; Q2;:: :) obtained* from the Lfp algorithm is increasing. That is, Qi Qi+1 for all i 0.

*Lemma 5.2.9 (Decreasing). The sequence of Q's (Q0; Q1; Q2;:: :) ob-* tained from the Gfp algorithm is decreasing. That is, Qi+1 Qi for all i 0.

*Lemmas 5.2.8 and 5.2.9 follow directly from the de nition of a monotonic* set transformer in De nition 5.2.7.

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

*EF : is potential* AF : is inevitable

*EG : is potentially invariant* AG : is invariant

*De nition 5.2.10 (CTL Semantics). Given a Kripke structure M =* (S; X; I; T; `), the semantics of a CTL formula is a set of states [[ ]] P(S).

|  |  |  |
| --- | --- | --- |
| *[[0]]* | *=* | *;* |
| *[[1]]* | *=* | *S* |
| *[[a]]* | *=* | *f s 2 S j a 2 `(s) g* |
| *[[: ]]* | *=* | *S n [[ ]]* |
| *[[ 1 \_ 2]]* | *=* | *[[ 1]] [ [[ 2]]* |
| *[[ 1 ^ 2]]* | *=* | *[[ 1]] \ [[ 2]]* |
| *[[EX ]]* | *=* | *f s 2 S j 9s0 2 [[ ]] : s s0 g* |
| *[[EG ]]* | *=* |  |

*Gfp Z h[[ ]] \ EX Z i*

*h i*

*[[EU( 1; 2)]] = Lfp Z [[ 2]] [ [[ 1]] \ EX Z*

*where a is an atomic proposition.*

*The set transformer arguments to Lfp and Gfp in De nition 5.2.10 are* monotonic. Lemma 5.2.11 proves it for [[ ]] \ EX Z.

*Lemma 5.2.11. The set transformer (Z) = [[ ]] \ EX Z is monotonic.* Proof. Let P1 P2. The set transformer (Z) is monotonic if (P1)

*(P2). Let s be some state in (P1). Then s is in [[ ]], and there exists a*

*state s0 2 P1 such that s s0. Since P1 P2, then s0 2 P2, and thus* s 2 (P2). ut

*The following is a list of common CTL formulae:* EF : It is possible to reach a state where holds.

*AG(Req ! AF Ack) : All requests will eventually be acknowledged.*

*AG AF : On every path holds in nitely often.*

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

* + 1. *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 them- selves, one is able to get highly eÆcient 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; `). We rede ne it in terms* of characteristic functions instead of sets.

*S : We encode a state as a vector of Boolean variables s = (s1;::: ; sn). We* refer to these variables as state variables.

*X : We encode an input as a vector of Boolean variables x = (x1;::: ; xm).* We refer to these variables as input variables.

*I : We use the characteristic function I(s).* T : T is already a function.

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

*si 2 `(s) if the i'th bit of s 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 ele-* ments of the vector and superscripts to indicate di erent vectors. For state variables, we use the convention that unprimed variables encode the current state while primed variables encode the next state. For example, s0 = T (s; x) indicates a transition on input x from current state s to next state s0.

*In the set version of model checking, each state was labeled with a set* of atomic propositions. The atomic propositions were not speci ed. Now we assume that the atomic propositions are state variables. This means we can change the CTL grammar from De nition 5.2.6 to re ect this. De ni- tion 5.2.12 shows the modi ed CTL grammar.

*De nition 5.2.12 (Modi ed CTL Syntax). A CTL formula can be* generated from the following grammar:

*f ::= 0 j 1 j si j :f j f \_ f j f ^ f j EX f j EG f j EU(f; f ) ;* where si 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* |
| *;* | *0* |
| *S* | *1* |
| *, f g* |  |
| *[* | *\_* |
| *\* | *^* |
| *n* | *^ :* |
| *S n* | *:* |
| *=* | *TAUT( $ )* |
| *6=* | *SAT( )* |
| *2* | *TAUT( ! )* |
| *62* | *SAT( ^ : )* |
|  | *TAUT( ! )* |
| *6* | *SAT( ^ : )* |

*Table 5.1: Correspondence between set and logic notations. On the set side, is an element and and are sets. On the logic side, , and are functions. These correspondences assume that jSj is equal to 2k 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: ; and S become 0 and 1, respectively,

*Q 6= Q0 becomes SAT(Q Q0), and the set transformer should work on* characteristic functions for sets instead of the sets themselves.

*We adapt the other CTL de nitions to characteristic functions as well.*

*De nition 5.2.13 (Modi ed CTL Semantics). Given a Kripke struc-* ture M = (S; X; I; T; `), the semantics of a CTL formula is a set of states [[ ]] P(S) as described in De nition 5.2.10. In terms of characteristic functions, [[ ]] is a Boolean function of the state vector s de ned recursively as follows:

|  |  |  |
| --- | --- | --- |
| *[[0]]* | *=* | *0* |
| *[[1]]* | *=* | *1* |
| *[[si]]* | *=* | *si* |
| *[[: ]]* | *=* | *:[[ ]]* |
| *[[ 1 \_ 2]]* | *=* | *[[ 1]] \_ [[ 2]]* |
| *[[ 1 ^ 2]]* | *=* | *[[ 1]] ^ [[ 2]]* |

*[[EG ]] = Gfp Z h[[ ]] ^ EX Z i* [[EX ]] = 9s0;x : s0 $ T (s; x) ^ [[ ]][s0=s]

*[[EU( 1; 2)]] = Lfp Z h[[ 2]] \_ [[ 1]] ^ EX Z i*

*where [[ ]][s0=s] is a substitution of s0 for s in [[ ]].*

*De nition 5.2.14 (Modi ed Model Checking). Let M = (S; X; I; T; `)* be a Kripke structure and let be a CTL speci cation for M . We say that M models , and write M j= , if TAUT(I ! [[ ]]).

## *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; `) and a CTL speci cation . The following steps are necessary to determine whether M j= :

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

*2. Compute a BED for [[ ]] using De nition 5.2.13.*

*3. Compute TAUT(I ! [[ ]]). The speci cation 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 X 7! S. We use Ti to indicate the function for the i'th state variable:

*n*

*^s $ Ti(s; x) ; (5.1)*

*s0 $ T (s; x) = 0*

*i*

*i=1*

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

*^*

*I = si $ Ii(s1;::: ; si 1; si+1;::: ; sn) : (5.2)* i

*Not all state variables need to be in the conjunction for I. If a state vari-* able is omitted it means that there are initially no restrictions on the state variable. The SMV language [McM93] for describing nite state machines lets the user specify the transition relation and the set of initial states in this functional form. More speci cally, the next and init statements in the ASSIGN section in SMV programs correspond to the Ti and Ii of Equa- tions 5.1 and 5.25.

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

*The second step was to compute a BED for [[ ]] using De nition 5.2.13.* The rst six lines of the de nition are straightforward: The BED is con- structed 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 [s0=s] in [[EX ]] is actually a variable renaming. It can* be done in one traversal of the BED. The quanti cation needs to be handled with care. We deal with it in Section 5.3.1.

*In [[EG ]] and [[EU( 1; 2)]] we need the two xed-point operators. In* line 3 of Lfp (Algorithm 5.3), we compare Q and Q0 to nd out whether we

*5Some versions of SMV allow the use of primed variables in the next statements. This would correspond to Ti(s; x; s0 ) instead of just Ti (s; x) in Equation 5.1. We do not consider such cases.*

*have reached the xed-point. Since the sequence of Q's in Lfp is increasing* (see Lemma 5.2.8), it always holds that Q Q0. Thus we can change the while condition to Q0 6 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 6 Q0. In terms of characteristic functions, these condi- tions become SAT(Q0 ^ :Q) and SAT(Q ^ :Q0), respectively. Section 5.3.2 deals with satis ability checking.

* + 1. *Quanti cation*

*The basic step in our quanti cation algorithm is to eliminate one quanti ed* variable by the following rules:

*9y : f f [0=y] \_ f [1=y] 8y : f f [0=y] ^ f [1=y] (5.3)*

*It is worth noting, that these basic steps can easily be computed by per-* forming 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 quanti er from a formula, we double* the formula size. Since each EX computation involves existential quanti - cation of n state variables and m input variables, we risk increasing the formula size by a factor of 2n+m. We have tried this kind of quanti cation on some examples. The result is always the same: Quanti cation of the rst handful or two of variables only increase the size of the BED slightly. How- ever, each of the remaining quanti cations nearly doubles the BED size. It is our experience that for any reasonably sized problem, this quanti cation method is not suÆcient.

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

*9y : g ^ (y $ f ) g[f =y] ; where y does not occur in f : (5.4)*

*Abdulla et. al. [ABE00] also use quanti cation by substitution in their model* checking algorithm. They call it inlining. One can think of quanti cation by substitution as a special case of pruning which we described in Section 3.2.3. We use the quanti cation by substitution rule in three places: EX com-

*putation, set inclusion, and iterative squaring.*

*EX Computation*

*Consider the EX computation in De nition 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

*quanti cation 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 9s0 : s0 $ T (s; x) ^ [[ ]][s0=s]:

*9s0 : (s0 $ T (s; x)) ^ [[ ]][s0=s]*

*= [[ ]][s0=s][T =s0]*

*= [[ ]][T =s]*

*The QbS algorithm assumes that T is given as the set of BEDs for Ti from* Equation 5.1 and that [[ ]] is also given as a BED. It works in a bottom- up way replacing all primed state variables in [[ ]][s0=s] (corresponding to unprimed state variables in [[ ]]) with their next-state function Ti. 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) QbS(low (u)); QbS(high(u))*

*4: if u is an unprimed variable vertex with var (u) = sj then* 5: return (Tj ^ h) \_ (:Tj ^ l)

*6: else*

*7: return Mk( (u); l; h)*

*Algorithm 5.4: The QbS algorithm for quanti cation by substitution. It com-*

*putes 9s0 : s0 $ T (s; x) ^[[ ]][s0 =s], where T is given as BEDs for T in Equation 5.1 and [[ ]] is given as a BED u.*

*i*

*Quanti cation by substitution works for the quanti cation 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 quanti cation by substitution for the quanti cation 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 ! [[ ]]), i.e.,* whether the initial set of states is a subset of the states characterized by the speci cation. We assume we have BEDs for the Ii functions (as per

*Equation 5.2) and for [[ ]]. In many cases Ii is either a constant or a very* simple function, and we can use this fact to simplify TAUT(I ! [[ ]]).

*Let I be written I0 ^(si $ Ii), where Ii is a function of all state variables* s1;::: ; sn, but not si. Recall that tautology checking corresponds to univer- sal quanti cation of all variables. This means that I ! [[ ]] is a tautology if and only if 8si : I ! [[ ]] is a tautology:

*8si : I ! [[ ]]*

*= 8si : : (I0 ^ (si $ Ii) ^ :[[ ]])*

*= :9si : I0 ^ (si $ Ii) ^ :[[ ]]*

*= :(I0 ^ :[[ ]])[Ii=si]*

*= (I0 ! [[ ]])[Ii=si]*

*The [Ii=si] means a substitution of Ii for si. In the third step we use quanti -* cation by substitution to replace a quanti cation by a substitution. In many cases the Ii functions are quite simple, e.g., a constant. In such situations this method reduces the number of variables and simpli es 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;::: ; 0). Our preprocessing step would simply replace all state variables in [[ ]] 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 xed-point for both the least and greatest xed-point opera- tors [BCL+94]. During reachability analysis we repeatedly square the tran- sition function:

*s0 $ T 2 s; (x; y) = 9s00 : s00 $ T (s; x) ^ s0 $ T (s00; y) :*

*T 2(s; (x; y)) is a new transition function allowing a transition from s to a*

*state s0 on input (x; y) if there is a middle state s00 such that s x*

*s00 and*

*s00 y 0*

*s , where indicates a transition in the old system.*

*Assume that T is written as in Equation 5.1.*

*s0 $ T 2 s; (x; y) = 9s00 : s00 $ T (s; x) ^ s0 $ T (s00; y)*

*= 9s00 : ^s00 $ Ti(s; x)!^ ^s0 $ Ti(s00; y)!*

*i*

*i*

*= ^s0 $ Ti(s00; y)[Tj(s; x)=s00]j ;*

*i*

*i*

*i*

*j*

*i*

*where [Tj(s; x)=s00]j is a substitution of function Tj(s; x) for variable s00 for*

*j j*

*all j. The algorithm is similar to QbS in Algorithm 5.4.*

*In this way we can compute T (2k ) in only k steps. T (2k ) is a new transition*

*function representing all paths in T with a length of exactly 2k, and T (2k )* 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 2k 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 simpli cations [CBM89, BCL+94].

*Algorithm 5.5 shows how to compute [[EF ]] using a least xed-point* method and iterative squaring. EX(2n ) is the EX operator with T (2n ) as transition function. After each iteration in the while loop, Q0 represents the set of states reachable in up to and including 2n 1 steps.

*Name: EF( ) 1: n 0*

*2: Q 0*

*3: Q0 [[ ]]*

*4: while Q 6= Q0 do*

*5: Q Q0*

*6: Q0 Q0 \_ EX(2n ) Q0*

*7: n n +1*

*8: return Q*

*Algorithm 5.5: The EF algorithm. It computes EF with a least xed-point*

*iteration and using iterative squaring. EX transition function.*

*(2n )*

*is the EX operator with T*

*(2n ) as*

*Scope Reduction Rules*

*Our veri cation method performs best when we can exploit the quanti -* cation by substitution rule. Such cases include systems with few or no

*inputs. After performing quanti cation by substitution, we quantify the* inputs variables using the rules below.

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

|  |  |  |
| --- | --- | --- |
| *9x : :f* | *=* | *:8x : f* |
| *9x : f \_ g* | *=* | *(9x : f ) \_ (9x : g)* |
| *9x : f (y) ^ g(x)* | *=* | *f (y) ^ (9x : g(x))* |
| *8x : :f* | *=* | *:9x : f* |
| *8x : f ^ g* | *=* | *(8x : f ) ^ (8x : g)* |
| *8x : f (y) \_ g(x)* | *=* | *f (y) \_ (8x : g(x))* |

*There are two ways to implement scope reduction rules. One way is* to think of the quanti ers 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 quanti er 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 quanti er vertex with three attributes: a quanti er quant 2 f9; 8g, a variable var and BED low . The semantics of a quanti er 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 modi ed to handle the new vertex type. This turns out to be rel- atively simple since the quanti ers distribute over the if-then-else operator (remember that BEDs are assumed to be free):

|  |  |  |
| --- | --- | --- |
| *9x : y ! f; g* | *=* | *y ! 9x : f; 9x : g* |
| *9x : x ! f; g* | *=* | *f \_ g* |
| *8x : y ! f; g* | *=* | *y ! 8x : f; 8x : g* |
| *8x : x ! f; g* | *=* | *f ^ g* |

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

* + 1. *Satis ability Checking*

*There are two places where we need to determine whether a Boolean formula* represented by a BED is satis able. First we need to detect that a xed- point has been reached in the computation of the set of states satisfying a CTL formula. This corresponds to SAT(Q0 ^ :Q) (or SAT(Q ^ :Q0)). The formula Q0 ^ :Q (or Q ^ :Q0) is satis able until we reach the xed-point. Then it is unsatis able. In other words, in all but one case the formula is satis able. A variable assignment satisfying the formula corresponds to a state which has just been added (or removed) from the approximation to the xed-point. It is our experience that SAT-solvers are good at nding a satisfying variable assignment so we suggest using a SAT-solver here. Of course, we do not need to detect the xed-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 satis ability check the rst n times. Another strategy would be to only check for satis ability 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 [[ ]] represented by the CTL speci cation: TAUT(I ! [[ ]]). There are two cases:

*The speci cation holds. This means that I ! [[ ]] is a tautology. We* could use a SAT-solver to prove that the negation of I ! [[ ]] is not satis able. However, it is our experience that most SAT-solvers are not very good at proving a formula to be unsatis able. We can also use BDDs. By using the Up One algorithm, we can convert the BED for I ! [[ ]] to a BDD. This results in the BED 1.

*The speci cation does not hold. A proof is a variable assignment* falsifying I ! [[ ]]. Or equivalent, a variable assignment satisfying

*:(I ! [[ ]]). SAT-solvers are good at nding such variable assign-* ments.

*Of course, in general we do not know beforehand whether the speci cation* 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 xing : 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 xed-point in the rst run as an estimate of the number of iterations in the subsequent runs. Thus we may avoid some of the SAT(Q0 ^ Q) checks in the xed-point algorithms.

*Chapter 6 discusses in more detail how to solve the satis ability 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 2k which is unfortunate.

*Let Vu 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

*Vv $ Vlow (v) op(v) Vhigh(v) ; and for each variable vertex v we create a clause*

*Vv $ var (v) ^ Vhigh(v) \_ :var (v) ^ Vlow(v) :*

*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:

*Vv $ Vl \_ Vh = (:Vv \_ :Vl) ^ (:Vv \_ :Vh) ^ (Vv \_ Vl \_ Vh) :* 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 satis able if and only if the other formula is satis able.

*:x x*



*(0,0)*

*x*

*(0,1)*

*x*

*x*

*(1,1)*

*x*

*(1,0)*

*:x x*

*Figure 5.6: A modulo-4 counter which counts every time the input variable x is true. Each state is labeled with a pair (s1; s0). The initial state (0; 0) is colored dark-gray.*

* + 1. *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 vari-* ables s0 and s1. The value of the counter is the binary number s1s0. The input variable x models the presence and absence of event e: x is true if event e takes place6.

*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 = (s0 $ 0) ^ (s1 $ 0) :*

*We construct the BEDs corresponding to I0 and I1 in Equation 5.2. Both*

*6Normally 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.*



*$*

*^*

*s0*

*s1*

*x*

*1*

*0*



*x*

*s1*

*0*

*1*

*Figure 5.8: The BEDs for the transition function functions T0 (left) and T1 (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:*

*s0 $ T = s0*

*0*

*^ s0*

*1*

*$ (:x ^ s0) \_ (x ^ :s1 ^ s0) \_ (x ^ s1 ^ :s0)*

*$ (:x ^ s1) \_ (x ^ :s1) :*

*We construct the BEDs corresponding to T0 and T1 in Equation 5.1; see* Figure 5.8.

*For the speci cation we choose the liveness property: It should always* be possible for the counter to reach the initial state. In CTL this becomes

*= AG EF (:s0 ^ :s1).*

*We have to compute [[ ]] = [[AG EF (:s0^:s1)]]. We start by computing* [[EF (:s0 ^ :s1)]]:

*[[EF (:s0 ^ :s1)]]*

*= [[EU (1; :s0 ^ :s1)]]*

*= Lfp Z h[[:s0 ^ :s1]] \_ EX Z i*

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

*imations Q0; Q1;:: ::*

*Q0 = 0*

*Q1 = s0\_ s1 \_ EX 0*

*= s0\_ s1*

*Q2 = s0\_ s1 \_ EX Q1*

*= s0\_ s1 \_ 9x : (s0 $ (s1^ x))\_ (x s1)*

*= s0\_ s1 \_ 9x : (s0 $ (s1^ x)) 6 (x $ s1)*

*= s0\_ s1 \_ s0 $ s1*

*= s0 $ s1*

*Q1 is s0\_ s1, which is the result of the BED simpli cations from Section 3.2* applied to :s0 ^ :s1. The Q2 approximation is s0\_ s1 \_ EX Q1. The second line in the Q2 calculation shows the result after quanti cation by substitution by Algorithm 5.4. The third line shows the simpli ed formula. The fourth line shows the formula after quanti cation of the input x. The

*fth and nal line shows the fully simpli ed Q2.*

*To determine whether Q1 is the xed-point, we check SAT(Q2 ^ :Q1).* The variable assignment (s1; s0) = (1; 1) is a witness and thus Q1 is not the

*xed-point.*

*It is possible to relate the xed-point calculations to Figure 5.6. The* function Q0 corresponds to the empty set of states. Q1 corresponds to the singleton set f(0; 0)g. The function Q2 corresponds to f(0; 0); (1; 1)g.

*Q3 = s0\_ s1 \_ 9x : (s0 $ (s1^ x)) $ (x s1)*

*= s0 s1*

*Q4 = s0\_ s1 9x : (s0 $ (s1^ x)) (x s1)*

*= 1*

*Q5 = s0\_ s1 \_ 9x : 1*

*= 1*

*First after calculating Q5 do we detect the xed-point. Both SAT(Q3^:Q2)* and SAT(Q4 ^ :Q3) are true, but SAT(Q5 ^ :Q4) is false and thus Q4 is the xed-point. We have now computed [[EF (:s0 ^ :s1)]] to be 1.

*The function Q3 corresponds to f(0; 0); (1; 1); (1; 0)g, and both Q4 and* Q5 correspond to the set of all states f(0; 0); (1; 1); (1; 0); (0; 1)g.

*We are now ready to compute [[ ]] = [[AG EF (:s0 ^ :s1)]]:*

*[[AG EF (:s0 ^ :s1)]]*

*= [[AG 1]]*

*= [[:EU(1; :1)]]*

*= :Lfp Z EX Z*

*Again we compute a series of approximations Q0; Q1;::: to the least xed-* point:

*Q0 = 0*

*Q1 = 0 \_ EX 0*

*= 0*

*We are already at the xed-point as SAT(Q1 ^ :Q0) = SAT(0) is false.* Thus [[ ]] = [[AG EF (:s0 ^ :s1)]] = :0 = 1.

*To determine whether the speci cation holds, we check TAUT(I ! [[ ]]):*

*TAUT(I ! [[ ]])*

*= TAUT(:s0 ^ :s1 ! 1)*

*= TAUT(1)*

*= 1*

*I ! [[ ]] is a tautology. This means that the speci cation holds for our* modulo-4 counter. This can, of course, trivially been seen from Figure 5.6.

## *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 modi ed version of SMV7, both of which are state- of-the-art in BDD-based model checking. Finally we compare reachability results with FixIt from Adbulla, Bjesse, and E en [ABE00].

*7See* [*http://www.cs.cmu.edu/*](http://www.cs.cmu.edu/) *bwolen.*

*The FixIt results are taken directly from the paper by Abdulla and his* group8. All other experiments are run on a Linux computer with a Pentium Pro 200 MHz processor and 1 gigabyte of main memory.

* + 1. *Multiplier*

*This example comes from the BMC-1.0f distribution9. It is a 16 16 7! 32* bit shift-and-add multiplier. The speci cation 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 nished its computation and the output bits of the two multipliers di er. The multiplier ts 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 unspeci ed initial state and the identity function as the next-state function. This lets us use quanti cation by substitution for

*all quanti cations in the xed-point calculations.*

*Table 5.2 shows the runtimes for verifying that the multiplier satis es* the speci cation. 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 diÆcult output in Table 5.2, the xed-point iteration ac-* counts for only a fraction of the total runtime for our method. It takes less than a minute and almost no memory to calculate the xed-point. By far the most time is spent in proving TAUT(I ! [[ ]]). SAT-solvers gave poor results, so we converted the BED for I ! [[ ]] to a BDD. The FixIt tool uses a SAT-solver to check TAUT(I ! [[ ]]). 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 conver- sion is quite large. Of course, this is expected since the formulae originate from multiplier circuits which are known to be diÆcult 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

*xed-point computations. The runtimes for output 9 to 12 are dominated*

*8From 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.*

*9See* [*http://www.cs.cmu.edu/*](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 veri cation could not be completed with 800 MB of memory.*

*by the BDD to BED conversion of I ! [[ ]]. The size of the BED for I ! [[ ]]* 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 rst 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 xed- point calculations. Table 5.3 shows the runtimes for calculating the xed- 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 xed-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 nd a satisfying assignment.

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

100000

Seconds (logscale)

FixIt

BED

Output

10000

1000

100

10

1

0 2 4 6 8 10 12

*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.*

*di erence is when we prove TAUT(I ! [[ ]]). Using BED to BDD conver-* sion as with the correct design, we now get poorer results because I ! [[ ]] is not a tautology and the nal BDD is not necessarily small. However, using a SAT-solver, we get much better results. In many cases, the SAT-solver is able to nd 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 squar- ing 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 nd a bug in the \correct" speci cation of the multiplier* for the two most signi cant outputs. Iterative squaring allowed us to quickly compute the xed-points, and Sato instantly found the errors. The total runtimes to nd 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 nding bugs in a system.

*We constructed shift-and-add multipliers of di erent sizes and veri ed* that they always terminate, i.e., we checked \AF done". The number of iterations needed to reach the xed-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 xed-point calculation in verifying the cor- rectness 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 signi cantly faster and we are able to handle much larger designs. We cannot compare with FixIt as it does not handle AF properties.

* + 1. *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 les. The contents of one of the register les is rotated at each step while the other le stays the same. The width of a register is log2 R, where R is the number of registers in the register le.

*The correctness of the barrel shifter is proven by showing that if two* registers from the les 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 les will eventually become equal. The number of iterations for the

*xed-point calculation is equal to the size of the register le. 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 rst 3 outputs are una ected by the bug. The rest of the outputs are erroneous. The veri cation of output 30 was no completed in 10 minutes, how- ever, 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 mul- tiplier; 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 di erent sizes always terminate, i.e., we check \AF done". The number of iterations to reach the xed-point is equal to the size of the multiplier. A dash indicates that the veri cation 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.

## *Model Checking of lfp-CTL*

*Symbolic model checking using xed-points involves quanti cation of both* state and input variables. A na ve 0/1 expansion as in Equation 5.3 does not work in practice. The formulae blow up in size after few quanti cations. The model checking method presented so far in this chapter uses quanti-

*cation by substitution to quantify out the state variables. It has proven* e ective 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 quanti ca-* tion 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 xed-points and hence we restrict ourselves to a xed-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 xed-point methods, we can hope to handle

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Size* | *BED* | *NuSMV* | *Bwolen* | *FixIt* |  | *Size* | *BED* | *NuSMV* | *Bwolen* |
| *2* | *0.1* | *0.1* | *1.0* | *0.1* |  | *2* | *0.2* | *0.1* | *1.0* |
| *4* | *0.3* | *0.2* | *2.5* | *0.1* |  | *4* | *0.5* | *0.2* | *2.1* |
| *6* | *0.4* | *609* | *-* | *0.2* |  | *6* | *0.7* | *521* | *-* |
| *8* | *0.4* | *-* | *-* | *0.5* |  | *8* | *0.9* | *-* | *-* |
| *10* | *0.6* | *-* | *-* | *1.1* |  | *10* | *1.2* | *-* | *-* |
| *20* | *1.9* | *-* | *-* | *14* |  | *20* | *3.2* | *-* | *-* |
| *30* | *4.0* | *-* | *-* | *52* |  | *30* | *5.9* | *-* | *-* |
| *40* | *8.0* | *-* | *-* | *231* |  | *40* | *11* | *-* | *-* |
| *50* | *13* | *- - 502* | | | *50* | | *18* | *- -* | |
| *60* | *19* | *- - ?* | | | *60* | | *28* | *- -* | |
| *70* | *30* | *- - ?* | | | *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 TAUT(I ! [[ ]]). A dash indicates that the veri cation could not be completed with 800 MB of memory.*

*designs which neither of those methods can handle.*

* + 1. *Theory*

*We use the same setup as described in Section 5.2, i.e., we use Kripke struc-* tures to represent nite state machines. Properties of systems are still mod- eled 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 opera- tors. The subset is called lfp-CTL and is de ned in De nition 5.5.1. The remaining temporal operators are next state X, future F and until U. Def- inition 5.5.2 gives the semantics of lfp-CTL.

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

*f ::= 0 j 1 j si j :si j f \_ f j f ^ f j EX f j EF f j EU(f; f ) j* AX f j AF f j AU(f; f ) ;

*where si is any state variable.*

*De nition 5.5.2 (lfp-CTL Semantics). Given a Kripke structure M =* (S; X; I; T; `), the semantics of an lfp-CTL formula is a set of states [[ ]] P(S). In terms of characteristic functions, [[ ]] is a Boolean function of the state vector s de ned recursively as follows:

|  |  |  |
| --- | --- | --- |
| *[[0]]* | *=* | *0* |
| *[[1]]* | *=* | *1* |
| *[[si]]* | *=* | *si* |
| *[[:si]]* | *=* | *:si* |
| *[[ 1 \_ 2]]* | *=* | *[[ 1]] \_ [[ 2]]* |
| *[[ 1 ^ 2]]* | *=* | *[[ 1]] ^ [[ 2]]* |

*[[EX ]] = 9s0;x : (s0 $ T (s; x) ^ [[ ]][s0=s]* [[AX ]] = :[[EX : ]]

*[[EF ]] = Lfp Z h[[ ]] \_ EX Z i*

*[[AF ]] = Lfp Z h[[ ]] \_ AX Z i*

*[[EU( 1; 2)]] = Lfp Z h[[ 2]] \_ [[ 1]] ^ EX Z i*

*[[AU( 1; 2)]] = Lfp Z h[[ 2]] \_ [[ 1]] ^ AX Z i*

*where [[ ]][s0=s] is a substitution of s0 for s in [[ ]].*

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

*Lemma 5.5.3. Let be a monotonic set transformer. Let k(;) be k ap-* plications of to ;: ( (::: (;) :: :)). Then k(;) k+1(;).

*Proof. (by induction) The lemma holds for k = 0: ; (;). Assume it* holds for k = j 1: j 1(;) j(;). As is monotonic, we can apply on each side: ( j 1(;)) ( j(;)) After regrouping we get j(;) j+1(;), which shows that the lemma holds for k = j. ut

*De nition 5.5.4 (k-CTL). [[ ]]k is the set of states represented by CTL* formula in which we make exactly k iterations (k applications of the set transformer ) in each xed-point computation.

*The following two theorems state that for lfp-CTL we can determine an* under-approximation to [[ ]], and if we know the diameter of the system M , we can compute [[ ]] exactly.

*Theorem 5.5.5 (Under-approximation). Let be an lfp-CTL formula.* Then, for all non-negative integers k,

*[[ ]]k [[ ]]k+1 [[ ]] :*

*Proof. We rst prove [[ ]]k [[ ]]k+1 by induction over the recursive depth of* the semantics in De nition 5.5.2. We use set notation in the argumentation. The four base cases 0, 1, si and :si trivially hold. The disjunction case

*[[ 1\_ 2]]k is equal to [[ 1]]k [[[ 2]]k, which per induction hypothesis is a subset* of [[ 1]]k+1 [ [[ 2]]k+1, which is equal to [[ 1 \_ 2]]k+1. A similar arguments is valid for the conjunction case. For [[EX ]]k we argue that it is the set of all states s such that there exists an s0 where s s0 and s0 2 [[ ]]k. Per induction hypothesis [[ ]]k [[ ]]k+1, and thus s 2 [[EX ]]k+1. A similar argument holds for [[AX ]]k.

*All the remaining cases involve the least xed-point operator with a*

*monotonic set transformer . We prove the inclusion for [[EF ]]k; the other* cases are handled in a similar way. Let 1 be the set transformer [[ ]]k [EX Z and 2 the set transformer [[ ]]k+1 [ EX Z. We prove by induction that

*k(;) k+1(;). The base case k = 0 holds trivially. For k = i, assume*

*1 2*

*i 1(;) i(;). Let s be some state in 1( i 1(;)) = i 1(;) [ EX i 1(;).*

*1 2 1 1 1*

*Remember that 2( i(;)) = i(;) [ EX i(;). If s 2 i 1(;), then per*

*2 2 2 1*

*induction hypothesis s 2 2( i(;)). Otherwise s 2 EX i 1(;), in which*

*2 1*

*case, per induction hypothesis, s 2 EX i(;). In either case we have that*

*2*

*k(;) k+1(;) and thus [[EF ]]*

*1 2 k*

*[[EF ]]k+1.*

*We now prove [[ ]]k [[ ]], however, we only consider the case where*

*has an Lfp-construction at the outermost level. Let s be some state in [[ ]]k.* Then it has been found within k iterations in Lfp. Let d be the number of

*iterations to reach the xed-point. If k d, then s 2 [[ ]]. Otherwise k < d.* Because of Lemma 5.5.3, s 2 d(;), where is the set transformer for the

*Lfp-construction. But d(;) is just another name for [[ ]], so s 2 [[ ]]. tu*

*Theorem 5.5.6 (Completeness). Let be an lfp-CTL formula. Then* there exists a non-negative number d such that:

*[[ ]]d = [[ ]] :*

*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* [[ ]]k [[ ]]k+1. For each Lfp-computation, each step either adds at least one new state or we are at a xed-point. Since the state space is nite there exists

*some nite d such that: [[ ]]0 [[ ]]1 ::: [[ ]]d = [[ ]]d+1 = [[ ]]d+2 = :: :.*

*Now we prove that [[ ]]d = [[ ]]. Let s be some state in [[ ]]d. Then s is* also in [[ ]]d+i for all i > 0 and thus more iterations in the least xed-point calculations do not result in any new states. We must therefore have reached

*the xed-point in all Lfp-computations. Hence [[ ]]d = [[ ]]. tu*

*Unfortunately, in practice it is diÆcult to nd the diameter of a system* short of actually performing a xed-point calculation. It is also diÆcult to give useful bounds on the diameter. Without a tight bound on the diameter, our method is not complete in practice.

*Instead of computing [[ ]]k, we compute a related formula [ ]k, where [ ]k* is de ned as:

*De nition 5.5.7. Let be a lfp-CTL formula and let k be a non-negative* integer. Then [ ]k is a function of state and input variables such that

*9x : [ ]k $ [[ ]]k*

*For a given and k, more than one function [ ]k satisfy De nition 5.5.7.* However, for our purpose this does not matter. We use [ ]k to indicate one such function.

*De nition 5.5.8 ( -renaming). Let be a function of input variables* xi;::: ; xi+j, then is a renaming of the input variables such that is a function of xk;::: ; xk+j where xk;::: ; xk+j are new, fresh variable vectors.

*We use x to indicate that extra input variables may have been added* through -renaming.

*We now introduce the notion of s-equality and note that [ ]k and [[ ]]k* are s-equal.

*De nition 5.5.9 (s-equality). Let 1 and 2 be formulae of state (s) and* input (x ) variables. Then =s is the equivalence relation:

*1 =s 2 9x : 1 $ 9x : 2* We say 1 and 2 are s-equal if they relate through =s.

*Observation 5.5.10 shows a recursive way to compute [ ]k. It is related*

*to De nition 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 [ ]k). Let be an lfp-CTL formula* and let k be a non-negative integer. The function [ ]k can be recursively computed as follows:

|  |  |  |
| --- | --- | --- |
| *[si]k* | *=s* | *si* |
| *[:si]k* | *=s* | *:si* |
| *[ 1 \_ 2]k* | *=s* | *[ 1]k \_ [ 2]k* |
| *[ 1 ^ 2]k* | *=s* | *[ 1]k ^ [ 2]k* |

*[EX ] = 9s0 : (s0 $ T (s; x) ^ [ ] (s0)*

*k s k*

*[EF ]k =s Lfpk Z h[ ]k \_ [EX Z]k i* [AX ]k =s :9s0 : s0 $ T (s; x) ^ :9x : [ ]k(s0)

*[AF ]k =s Lfpk Z h[ ]k \_ [AX Z]k i*

*[EU( 1; 2)]k =s Lfpk Z h[ 2]k \_ ( [ 1]k ^ [EX Z]k) i*

*h i*

*[AU( 1; 2)]k =s Lfpk Z [ 2]k \_ ( [ 1]k ^ [AX Z]k)*

*where [ ]k(s0) is a shift of variables from s to s0 in [ ]k and Lfpk indicates* the result of a least xed-point algorithm after the rst k iterations.

*Theorem 5.5.11. Let be an lfp-CTL formula and M = (S; X; I; T; `) a* Kripke structure. Then, for all non-negative integers k, if I is a singleton set,

*SAT (I ^ [[ ]]k) ! M j=*

*Proof. In general, if I is a subset of [[ ]]k, then M j= . Since I is a singleton* set, it is enough to prove the existence of an element in the intersection of

*I and [[ ]]k. tu*

*Lemma 5.5.12. Let be an lfp-CTL formula and M = (S; X; I; T; `) a* Kripke structure. Then, for all non-negative integers k, if I is a singleton set,

*(SAT (I ^ [[ ]]k) ! M j= )*

*! SAT I ^ [[ ]]k+1 ! M j=*

*Proof. Follows from Theorems 5.5.5 and 5.5.11. tu*

*Theorem 5.5.13. Let be an lfp-CTL formula and M = (S; X; I; T; `) a* Kripke structure. Then, for all non-negative integers k,

*SAT (I ^ [[ ]]k) = SAT (I ^ [ ]k)*

*Proof. Apply the de nition of [ ]k, move the quanti er outward, and note* that SAT ( ) corresponds to an existential quanti cation of all variables in

*:*

*SAT (I ^ [[ ]]k) = SAT (I ^ 9x : [ ]k)*

*= SAT (9x : I ^ [ ]k)*

*= SAT (I ^ [ ]k)*

*ut*

*The restriction in Theorem 5.5.11 that I has to be a singleton set is* to prevent an alternation of quanti ers. The typical check is whether I is a subset of [[ ]]k. In terms of characteristic functions we check 8s : I ! [[ ]]k. However, since we compute [ ]k and not [[ ]]k, we would introduce an

*existential quanti er within the scope of the universal quanti er. By limiting*

*ourselves to exactly one initial state, it is enough to check whether I and* [[ ]]k have a state in common. It is possible to overcome this restriction by constructing a new system with a new single initial state ~i and transitions from ~i to all states in the old set of initial states I. Using the AX operator we go one step backward and only include ~i if all states in I were in [[ ]]. Unfortunately, the computation of [AX ] contains quanti cations so unless we nd some smart way of handling [AX ], we have just pushed the problem from a restriction of I to AX computation.

* + 1. *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; ; k)*

*1: Compute [ ]k using Observation 5.5.10* 2: if SAT(I ^ [ ]k) then

*3: return \M models "*

*4: else if k diameter of M then* 5: return \M does not model " 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; `) is a model for an lfp-CTL formula . The number of iterations in the xed-point calculations is k.*

*Computing [ ]k*

*We use BEDs to represent the Boolean formulae. Each state variable cor-* responds to a BED variable. Boolean connectives (negation, conjunction, disjunction) correspond to operator vertices. -renaming corresponds to variable substitution. We use quanti cation by substitution for quanti ca- tion of state variables.

*The AX operator causes trouble as it requires quanti cation of all input* variables to compute [AX ]k. We cannot use quanti cation by substitu- tion, so we are stuck with a na ve 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 nd errors in AG properties10.

*Computing SAT(I ^ [ ]k)*

*Recall that we only have one initial state. Since we have a conjunction be-* tween this state and [ ]k, we can compute SAT(I ^ [ ]k) as SAT([ ]k(I)), where [ ]k(I) means a substitution of all state variables in [ ]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 sat- is ability.

## *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 repre- sent sets of states by characteristic functions. The data structure of Binary Decision Diagrams turns out to be a very eÆcient representation for char- acteristic 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 nding errors with short counterexamples. The diameter of

*10If AG does not hold, then there is a state such that : holds. In other words, EF : 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 veri cation method in practice.

*The work most closely related to ours is by Abdulla, Bjesse and E en.* 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 xed-point iteration, and like us they make use of the quanti cation by substitution rule. They use St almarck's patented method [SS98] to determine satis ability of Boolean functions. While related, their method and ours di er in a number of ways: In our method the quanti cation by substitution rule is extensively used at three di erent places while they only use it during xed-point calcula- tion. We have heuristics for choosing di erent SAT-procedure depending on the expected result of the satis ability check. Candidates are various SAT-solvers or an explicit BED to BDD conversion. They use St almarck's method as the only SAT-procedure used. BEDs are always locally reduced and we identify further important simpli cation 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 ex-* amples. Both logics do not require a path quanti er 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 A , where is a path formula.

*The three temporal logics have di erent 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. St almarck's method also works on a non-CNF representation. Chapter 6 discusses how to do satis ability 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 nite 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 veri cation depends on the prop- erty, and not on the system. This is opposed to symbolic model checking where the number of BDD variables depends on the system.

## *Conclusion*

*We have presented a BED-based CTL model checking method based on the* classical xed-point iterations. Quanti cation is often the Achilles heel in CTL xed-point iterations but by using quanti cation by substitution we are in some cases able to deal e ectively 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 quanti cation by substitution rule.

*We have shown how the quanti cation by substitution rule can also help* simplify the nal set inclusion problem of model checking and help perform eÆcient iterative squaring. Our proposed method combines SAT-solvers and BED to BDD conversions to perform satis ability 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 stan- dard BDD-based model checking methods. Furthermore, we were able to

*nd a previously undetected bug in the speci cation 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 ! [[ ]] 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 ! [[ ]] to a BDD, our method will bene t from a good variable ordering heuristic. The other problem is the order in which we quantify the variables in the [[EX ]] computation. This is interesting especially in cases where we cannot use the quanti cation by substitution rule. Finally, it would be worth looking into

*nding a way around the quanti cation problem with AX in lfp-CTL.*

*Chapter 6*

# *Satis ability*

*In this chapter we show how to determine satis ability of a formula repre-* sented 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].

## *Introduction*

*In Chapter 3 we have dealt with the tautology problem for BEDs. Two at* combinational circuits are equivalent if f $ g is a tautology, where f and g are the functions modeling the outputs of the two circuits. In case of an error, f $ g is not a tautology. In other words, if :(f $ g) is satis able then there is an error.

*In Chapter 5 we determine whether I ! [[ ]] is a tautology. If it is,* then the CTL property holds for the system. Otherwise :(I ! [[ ]]) is satis able, and the property does not hold.

*In both cases, tautology is associated with correct behavior while satis-*

*ability is associated with errors.*

*Given a BED for a formula, one way of proving satis ability is to con-* vert the BED to a BDD. If the resulting BDD is 0 (a contradiction), then the formula is not satis able. Otherwise the formula is indeed satis able. Table 6.1 illustrates it for both satis ability (SAT) and tautology (TAUT). Recall from Section 1.2 that there is the following relation between the two:

*SAT = :TAUT : :*

*Converting a BED into a BDD allows us to determine satis ability. How-* ever, we obtain more information than just a \yes, the formula is satis able"

*131*

|  |  |  |
| --- | --- | --- |
| *Formula* | *SAT* | *TAUT* |
| *Tautology* | *yes* | *yes* |
| *Contradiction* | *no* | *no* |
| *Neither* | *yes* | *no* |

*Table 6.1: The table shows the result of solving the satis ability (SAT) and tautol- ogy (TAUT) problem for three di erent 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 satis able" 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 satis ability 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 eÆcient in proving satis ability. 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.

## *Satis ability 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 di erent 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*

*1: if is the empty set of clauses then* 2: return 1

*3: else if contains the empty clause then* 4: return 0

*5: else if a unit clause l occurs in then* 6: return DP(assign(l; ))

*7: else*

*8: l choose-literal( )*

*9: return DP(assign(l; )) \_ DP(assign(:l; ))*

*Algorithm 6.1: The basic version of Davis-Putnam. The function assign(l; ) applies the truth value of literal l to the CNF formula . The function choose- literal( ) picks a literal for DP to split on.*

## *Satis ability Using Boolean Expression Dia-* grams

*The basis of Davis-Putnam is a splitting on literals. In BEDs we can obtain* the same e ect 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 satis able if either the low child or the high child (or both) represent a satis able 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* satis able. 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 contains the empty clause (line 3) becomes a test for the terminal 0. We cannot nd 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 = 1 then*

*2: return 1*

*3: else if u = 0 then* 4: return 0

*5: else*

*6: x choose-variable(u) 7: u0 Up One(x; u)*

*8: if u0 is a variable x vertex then*

*9: return BedSat low (u0) \_ BedSat high(u0) 10: else*

*11: return BedSat u0*

*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 di erent heuristics for picking good variable orderings for Up One. The rst 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 di erent branches. We have chosen a simple implementation which does a depth- rst traversal of the BED and picks the rst 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 rst 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 num- bers 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 dur- ing 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 2n.

*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 nish 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 satis able 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 nd a satisfying assignment in the easy part of the state space then we never need to revisit the diÆcult parts. For unsatis able 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 di erent sequence of variables to split on, the rewriting rules collapse the diÆcult part of the state space.

## *Experimental Results*

*To see how well BedSat works in practice, we compare it to other tech-* niques 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 satis a- bility 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 in- put 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 St almarck's method as it is targeted toward proving tautology and for

*2n 4*



*2n 1*

*2n 2*

*2n 3*

*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*

*n 2 n 4*

*state space which has already been examined is 2 +2 .*

*2n*

*non-tautology formulae, St almarck's method is not complete in practice.*

*Table 6.2 shows the ISCAS'85 results. The rst ten rows represent*

*475 unsatis able 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 cir- cuits, 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) unsatis able formulae. In our sim- ple 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 ve rows of Table 6.2 show the results for the erroneous circuits.* Here there are 340 functions in total out of which 267 are unsatis able and

*73 are satis able. 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 ve are faulty in the erroneous version. BedSat has no problem detecting the errors in the ve faulty outputs. In the correct version, about 149 seconds were spent on proving those ve outputs to be unsatis able. Another example is c1908 where, in the correct case, BedSat spends all its time (242 seconds) on one unsatis able output. In the erroneous version the diÆcult output has an error and the corresponding function becomes satis able. BedSat nds a satisfying assignment instantaneously (0.1 seconds). This indicates that BedSat is not very good at handling unsatis able 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 satis ability of problems arising in veri cation of the ISCAS'85 benchmarks using di erent approaches. In the*

*\Result" column, \U" indicates unsatis able problems while \S/U" indicates both satis able and unsatis able 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 satis ability problems for the model checking experiments in Ta-* ble 5.2 ( rst nine rows) and Table 5.4 (last nine rows). The numbers 10,

*20 and 30 indicate the output bit we are considering. The word \ nal"* indicates the satis ability problem for the nal check of the speci cation. The word \last lp" indicates the satis ability problem for the last iteration in the xed-point computation (where we detect that we have reached the

*xed-point). The word \second last fp" indicates the satis ability problem* for the previous iteration in the xed-point computation. The result column indicates whether the satis ability problem is satis able (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 satis able and the unsatis able 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.

## *Related Work*

*The satis ability 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 satis ability.

*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. St almarck's method also works on a non-CNF representation.

## *Conclusion*

*In this chapter we have presented BedSat as an algorithm for solving the* satis ability 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*

*10 final*

*10 last fp*

*10 second last fp*

*20 final*

*20 last fp*

*20 second last fp*

*30 final*

*30 last fp*

*30 second last fp bug 10 final*

*bug 10 last fp*

*bug 10 second last fp bug 20 final*

*bug 20 last fp*

*bug 20 second last fp bug 30 final*

*bug 30 last fp*

*bug 30 second last fp*

*Result Up One Up All Sato Grasp BedSat*

*U 13.1 43.5 - - -*

*U 10.3 - 0.1 0.1 0.2*

*S - - 0.1 0.1 0.1*

*? - - - - -*

*U - - 0.1 0.1 0.1*

*S - - 0.5 40.9 0.5*

*S - - 0.3 0.6 0.2*

*U - - 0.1 0.2 0.2*

*S - - 0.6 1.4 0.5*

*S 13.0 - 6.7 0.1 0.1*

*U 9.9 - 0.1 0.1 0.2*

*S - - 0.1 0.1 0.1*

*S - - 113 - 0.3*

*U - - 0.1 0.1 0.1*

*S - - 0.5 499 0.5*

*S - - 0.3 0.6 0.2*

*U - - 0.1 0.2 0.2*

*S - - 0.6 1.5 0.5*

*Table 6.3: Runtimes in seconds for determining satis ability of problems arising in model checking of multipliers using di erent approaches. In the \Result" column,*

*\U" indicates unsatis able problems while \S" indicates satis able 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 ex-* amples 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 unsatis able problems. Implementation of premature backtracking may also help on the performance since BedSat could backtrack out of diÆcult parts of the state space and either nd a satisfying assignment elsewhere or let the rewriting rules and di erent split sequences handle the more diÆcult parts.

*Chapter 7*

# *Extending Boolean* Expression Diagrams

*A Boolean Expression Diagram is an extension of a Binary Decision Dia-* gram 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 struc-* ture? 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 connec- tives 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.

*141*

*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 eld var , one operator eld op, and two BED elds low and high available per vertex. The low and high elds must point to valid BED vertices as the elds are used in traversal of the BED1. The op eld is an identi cation 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 eld should contain an identi cation of the vertex type. The var eld 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 de ned 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 quanti -* cation and substitution. Quanti cation, 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 Quanti cation*

*The rst new vertex type we introduce is one for existential quanti cation.* We use the following attributes:

*var : The variable to be existentially quanti ed.*

*low : The function in which the variable is quanti ed.* high : Not used2.

*An existential quanti cation vertex v denotes the Boolean function:* f v = 9var (v) : f low(v) :

*1We could use the elds for other purposes, but that would require more complicated*

*traversal algorithms and we have opted not to do it.*

*2Just 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 di erent variables and let f and g be functions* represented by BEDs. The terminal cases for existential quanti cation are:

*9x : 0 = 0*

*9x : 1 = 1*

*Existential quanti cation distributes over if-then-else in the following way:* 9x : x ! f; g = g \_ f ; x 62 sup(f ) [ sup(g) (7.1)

*9x : y ! f; g = y ! (9x : f ); (9x : g)*

*The requirement that x is not a free variable of f or g in 9x : x ! f; g is not* a problem because we assume BEDs are always free as per De nition 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 na ve quanti er* expansion in Equation 5.3:

*9x : x ! f; g*

*= (x ! f; g)0 \_ (x ! f; g)1*

*= (0 ! f; g) \_ (1 ! f; g)*

*= g \_ f*

*and*

*9x : y ! f; g*

*= (y ! f; g)0 \_ (y ! f; g)1*

*= (y ! f0; g0) \_ (y ! f1; g1)*

*= (y ^ f0) \_ (:y ^ g0) \_ (y ^ f1) \_ (:y ^ g1)*

*= y ^ (f0 \_ f1) \_ :y ^ (g0 \_ g1)*

*= y ! (9x : f ); (9x : g)*

*where a subscript 0 (1) indicates the negative (positive) co-factor with re-* spect to x.

*7.2.2 Universal Quanti cation*

*The universal quanti er is the dual of the existential quanti er. We add it* to the data structure in a similar way. We use the following attributes:

*var : The variable to be universally quanti ed.*

*low : The function in which the variable is quanti ed.* high : Not used.

*A universal quanti cation vertex v denotes the Boolean function:*

*f v = 8var (v) : f low(v) :*

*Let x and y be two di erent variables and let f and g be functions* represented by BEDs. The terminal cases for universal quanti cation are:

*8x : 0 = 0*

*8x : 1 = 1*

*Universal quanti cation distributes over if-then-else in the following way:*

*8x : x ! f; g = g ^ f ; x 62 sup(f ) [ sup(g) (7.2)*

*8x : y ! f; g = y ! (8x : f ); (8x : g)*

*The proof of 7.2 follows along the same lines as the corresponding proof* for existential quanti cation.

*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 [[EX ]]. Back then we simply reconstructed the whole BED, but with a new set of variables. This worked ne, 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 di erent 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 di erent variables. It would be more memory eÆcient to have just one copy of f , and then have di erent 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 di erent variables and let f , g and h be functions* represented by BEDs. The terminal cases for substitution are:

*0[f =x] = 0*

*1[f =x] = 1*

*Substitution distributes over if-then-else in the following way:*

|  |  |  |  |
| --- | --- | --- | --- |
| *(x ! f; g)[h=x]* | *= (h ^ f ) \_ (:h ^ g)* | *;* | *x 62 sup(f ) [ sup(g) [ sup(h)* |
| *(y ! f; g)[h=x]* | *= y ! f [h=x]; g[h=x]* |  | *(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 ^ f ) \_ (:h ^ g) corresponds to h ! f; g. However, we write (h ^ f ) \_ (:h ^ g) to indicate that that is way we construct the BED.

*The proof of 7.3 is straightforward. The rst 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: quanti cation of one variable* or substitution of one variable with a function. In many situations it is interesting to perform quanti cation of a vector of variables or substitutions of a vector of variables with a vector of functions.

*Vector Existential Quanti cation*

*In the one variable version of existential quanti cation, we use the var eld* 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 eld contains the BED for the conjunction.

*var : Not used.*

*low : The function in which the variables are quanti ed.*

*high : A conjunction of the variable to be quanti ed. (A conjunction of zero* elements is 1.)

*An existential quanti cation vertex v denotes the Boolean function:* f v = 9 x 2 sup(high(v)) : f low(v) :

*Let x be a vector of variables. Let x be one of those variable, and let y* be a variable not in x . Let f and g be functions represented by BEDs. We use ; to denote the empty set of variables (a vector of dimension zero). The terminal cases for existential quanti cation are:

*9x : 0 = 0*

*9x : 1 = 1*

*9; : f = f*

*Existential quanti cation distributes over if-then-else in the following way:* 9x : x ! f; g = 9x n x : g \_ f ; x 62 sup(f ) [ sup(g)

*9x : y ! f; g = y ! (9x : f ); (9x : g)*

*where x n x is the vector of all the variables in x excluding the variable*

*x. The BED in the 9x : x ! f; g case looks like this: The top vertex v* has low (v) = x ! f; g and high(v) = x ! h; 0 3. The resulting vertex v0 has low (v0) = g \_ f and high(v0) = h. The BED h is a conjunction of the variables x n x. See Figure 7.1.

*Vector Universal Quanti cation*

*The vector version of universal quanti cation resembles the vector version* of existential quanti cation.

*var : Not used.*

*low : The function in which the variables are quanti ed.*

*3Since 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.*

*!*

*v*

*9*

*x*

*x*

*v 0*

*9*

*\_*

*h*

*g f 0 h g f*

*Figure 7.1: The BED v for 9x : x ! f; g and the BED v0 for 9x n x : g \_ f . The BEDs show how the vector existential quanti cation operator distributes over if-then-else.*

*high : A conjunction of the variable to be quanti ed. (A conjunction of zero* elements is 1.)

*A universal quanti cation vertex v denotes the Boolean function:* f v = 8 x 2 sup(high(v)) : f low(v) :

*Let x be a vector of variables. Let x be one of those variable, and let y* be a variable not in x . Let f and g be functions represented by BEDs. We use ; to denote the empty set of variables (a vector of dimension zero). The terminal cases for universal quanti cation are:

*8x : 0 = 0*

*8x : 1 = 1*

*8; : f = f*

*Universal quanti cation distributes over if-then-else in the following way:* 8x : x ! f; g = 8x n x : g ^ f ; x 62 sup(f ) [ sup(g)

*8x : y ! f; g = y ! (8x : f ); (8x : g)*

*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 corre- sponding element in the variable vector. This means we need two vectors and a function in which to perform the substitutions. To t 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 rst element in a map list.

*gx gy*

*map x*

*map y*

*map y*

*map x*

*gy gx*

*f f*

*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)[h =x ] ;*

*where [h =x ] is the map of variables x to functions h in the high(v) map list.*

*Let x be a vector of variables. Let x be one of those variable, and let y* be a variable not in x . Let f and g be functions represented by BEDs. We use ; to denote the empty map list. The terminal cases for substitution are:

|  |  |  |
| --- | --- | --- |
| *0[h =x ]* | *=* | *0* |
| *1[h =x ]* | *=* | *1* |
| *f [;]* | *=* | *f* |

*Substitution distributes over if-then-else in the following way:*

*(x ! f; g)[h =x ] = (hx ^ f ) \_ (:hx ^ g) [h n hx = x n x]*

*(y ! f; g)[h =x ] = y ! f [h =x ]; g[h =x ]*

*where hx is the function in h to be substituted for x, and assuming that in* the rst equation x 62 sup(f ) [ sup(g).

*Vector Existential Quanti cation and Substitution*

*In symbolic model checking, we compute [[EX ]] as 9s0;x : T (s; x; s0) ^* [[ ]][s0=s]. BDD-based model checkers often have one specialized function for performing the vector quanti cation and the conjunction in one single step. We now show how to combine the vector quanti cation and the vector substitution in one new operator type in BEDs4. 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 di erent 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 quanti ed and even variables should be replaced by the variable with one higher number.

*We use the following elds in an ESUB operator vertex:* var : Not used.

*low : The function on which the ESUB operates.*

*high : Not used.*

*4The idea of combining the two is due to Henrik Reif Andersen.*

*An ESUB vertex v denotes the Boolean function:*

*f v = 9x 0 : f low(v) [x 0=x ] :*

*where x is a vector of even variables and x 0 is a vector of odd variables.*

*Let x be an even variable and x0 the corresponding odd variable. Let f* and g be functions represented by BEDs. The terminal cases for ESUB are:

*ESUB 0 = 0 (7.4)*

*ESUB 1 = 1*

*ESUB distributes over if-then-else in the following way:*

|  |  |  |
| --- | --- | --- |
| *ESUB x0 ! f; g* | *= ESUB g \_ ESUB f ;* | *x0 62 sup(f ) [ sup(g)* |
| *ESUB x ! f; g* | *= x0 ! ESUB f; ESUB g* | *(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( ; l; h)*

*1: if there exists a ( ; l; h) vertex then* 2: return that vertex

*3: else if is a variable and l = h then* 4: return l

*5: else if 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(:; l; ) or Mk(:; h; )*

*7: else if is an ESUB operator and l is a terminal then* 8: return l

*9: else*

*10: return new vertex ( ; l; h)*

*Algorithm 7.3: The Mk algorithm modi ed to handle the terminal case of ESUB. Line (7) and (8) are the new ones.*

*Algorithm 7.4 shows the pseudo-code for the modi ed 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 modi ed 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 One0(x; u)*

*Require: The memorization table M is initialized to empty prior to the*

*rst 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) Up One0(x; low (u))

*5: (hl; hh) Up One0(x; high(u)) 6: if u is an ESUB vertex then*

*7: rl Mk(ESUB; ll; ll)*

*8: rh Mk(ESUB; lh; lh)* 9: if x is even then

*10: rl rh Mk(x + 1; rl; rh) 11: else*

*12: rl rh Mk(\_; rl; rh) 13: else*

*14: rl Mk( (u); ll; hl)*

*15: rh Mk( (u); lh; hh)* 16: insert ((x; u); (rl; rh)) in M 17: return (rl; rh)

*Algorithm 7.4: The Up One0 algorithm modi ed 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 overa set of variables X = fx1;::: ; xng,*

*Name: Up All(u)*

*Require: The memorization table M is initialized to empty prior to the*

*rst 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( (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( (u); low (l); low (h)))*

*20: tmp2 Up All(Mk( (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( (u); low (l); h))*

*24: tmp2 Up All(Mk( (u); high (l); h)) 25: r Mk(var (l); tmp1; tmp2)*

*26: else*

*27: tmp1 Up All(Mk( (u); l; low (h)))*

*28: tmp2 Up All(Mk( (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 modi ed to handle the ESUB operator. The new lines are (8) through (17).*

*the two constants 0; 1 2 B and the usual operators ^, \_, :, x ! y; z (if-* then-else) etc. A literal is either a variable x or its negation :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 be a product that contains only positive literals.*

*We denote by c*

*X*

*the minterm obtained by adding to the negative literals*

*formed over all of the variables occurring in f but not in . is a p-cut of f*

*X*

*if c*

*X*

*j= f . It is minimal if there is no product Æ such that Æc*

*j= f . We*

*denote by (f ) the set of the minimal p-cuts of the formula f and 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 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 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 implicants5 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 posi-* tive 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 qual- itative analysis of fault trees, as it identi es 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 P complete problem [Pap94]. Therefore, unless NP=coNP=P, there

*5An implicant for function f is a product over the variables in f such that j= f . The implicant is prime if it has the property that for all shorter implicants , j= f .*

*do not exist eÆcient (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 the- orems. The rst one establishes that minterms with more than k positive literals are useless for computing k(f ). The second theorem gives a re- cursive principle for computing 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:

*k(f ) = 1(f \ minterms+(X));*

*k*

*where minterms+(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.

*k*

*Theorem 7.3.2 (Dutuit & Rauzy [DR97]). Let f = x ! f1; f0 be a* Boolean formula with f1 and f0 not depending on x. Then, k(f ) can be obtained as the union of two sets k(f ) = v: 1 [ 0 where 0 = k(f0),

*1 = k 1(f1 \_ f0) n 0, v:P = fv ^ j 2 P g and n denotes set di erence.*

*We exploit this fact not only to compute 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* suÆcient 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 ner 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 c , where is a k-truncated

*X*

*minimal p-cut of f . In each step of the new algorithm, the Up One trans-* formation 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 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:*

|  |  |  |
| --- | --- | --- |
| *PC(0)[k; L]* | *=* | *0* |
| *PC(1)[k; ;]* | *=* | *1* |
| *PC(1)[k; x:L]* | *=* | *:x ^ PC(1)[k; L]* |

*where x:L denotes the union of the sets fxg and L (x 62 L). PC distributes* over if-then-else in the following way:

*PC(x ! f; g)[0; x:L] = :x ^ PC(g)[0; L]*

*PC(x ! f; g)[k; x:L] = x ! S; T (k > 0)*

*T = PC(g)[k; L]*

*S = PC(f \_ g)[k 1; L] ^ :T*

*PC(x ! f; g)[k; y:L] = :y ^ PC(x ! f; g)[k; L] (x 62 L)*

*To calculate the minimal truncated p-cuts we use either Up All (cor-* responding 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 c*

*for a k-truncated p-cut is equal to*

*P n X*

*k*

*i=0*

*i*

*, where n is the number of variables in X. This number is bounded*

*by O(nk). Each k-truncated p-cut uses at most k variable vertices. Thus* O(nk) 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(nk).

*up*

*P*

*P*



*F*

*PC*

*up x*  *up x*

1 3

*F*



*PC PC PC PC*

*x*0 *x*1 *x*2 *x*3 *xn x*0 *x*2 *xn*

*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 diÆcult 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 mul- tiplier [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 = x1x2 and y = y1y2 be two 32-bit numbers where x1 and y1 represent the 16 most signi cant bits and x2 and y2 the 16 least signi cant bits. A 32-bit multiplication of x and y can be done by use of 16-bit multiplications of x1, x2, y1, and y2 in the following way:

*z = x1x2 32 y1y2 = (x1 16 y1) (216)2 + (x1 16 y2 + x2 16 y1) 216 + x2 16 y2*

*Here n represents n-bit multiplication while alone represents a multipli-* cation 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

*z*

*+*

*x1 y1 x1 y2*

*x2 y1*

*x2 y2*

*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* veri cation 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 veri ed to be equivalent by pulling* two variables (256gat and 290gat) up to the root. Each of the pairs of larger multipliers is veri ed 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 di erent 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

*z*

*x0 := x0 :=*

*x0 := x0 :=*

*y0 :=*

*y0 :=*

*+*

*x1 y1*

*x0 y0*

*y2 x2*

*Figure 7.8: A 2n-bit multiplier created from one n-bit multiplier using substi- tution. x0 and y0 are the formal parameters for the multiplication cell which are replaced with the actual parameters x1, x2, y1, and y2.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *n* | *Ops* | *Subst* | *Ntotal* | *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* | *1:05 106* | *29.5* |
| *512* | *56 401* | *2 793 472* | *4:19 106* | *119* |
| *1024* | *109 643* | *11 192 080* | *14:3 106* | *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. Ntotal is the total number of vertices used in the veri cation. CPU is the veri cation time measured in seconds on a Sun Ultra- SPARC 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 nd 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.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Name* | *1* | *No.*  *2* | *of p-cuts*  *3 4* | | *1* | *Run*  *2* | *time [sec]*  *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 les. The standard method depends on the variable ordering, and using improved heuristics to determine a good initial variable ordering will de nitely improve the performance. However, the Up One method will also bene t 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 exis- tential and universal quanti ers in their XBDDs [JPHS91]. Hett, Drechsler, and Becker add existential quanti ers to obtain a new method for BDD construction [HDB96, HDB97].

*Most BDD implementations have algorithms for doing substitution and* quanti cation. 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

*rst 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 quanti cation and sub- stitution algorithms which allow substitution and quanti cation of vectors. The ESUB operator combines two di erent operations in the compu- tation of [[EX ]]: the existential quanti cation and the substitution. In BDD-based model checking, existential quanti cation and Apply are of- ten combined. Combinations of this kind are very e ective 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 quanti cation vertices and of substitution ver- tices. Furthermore, we have presented an operator vertex, ESUB, combining existential quanti cation 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 modi cations 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 veri cation. We have selected a number of di erent problems domains for investigation.

*The rst problem domain was veri cation of combinational circuits. We* considered both circuits described in a at way and circuits described in a hierarchical way.

*From a at 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.

*St almarck : Reasoning-based method. As opposed to the other two meth-* ods, St almarck's method does not convert or change the formula it works on. In terms of speed, St almarck'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 veri cation:

*161*

*Use BEDs as BDDs with Up All. The rewriting rules form a prepro-* cessing step which helps speed up the veri cation 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 speci c 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 veri ed using Up One.

*Circuits described in a hierarchical way may be converted to at circuits* and then veri ed 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 simi-* lar hierarchical structures. The advantage is that instead of working with representations of the functionality of the circuits, we work with represen- tations 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 veri ed 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 pre-* sented a method for CTL model checking based on xed-point iterations. We use quanti cation by substitution for the quanti cation whenever pos- sible. Quanti cation by substitution works well with BEDs. Quanti cation 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 re- duction rules to press the quanti ers as far down as possible in the formulae. Then we perform the quanti cation using Up One.

*Symbolic model checking is typically done using the BDD data structure* where equivalence and satis ability 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*

*8.1. FUTURE DIRECTIONS 163*

*with few or no input variables. In such examples we can fully exploit quanti-*

*cation 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* quanti cation 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 nd 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 satis ability 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 satis ability checking. We have compared a simple implementation of BedSat with state-of-the- art SAT-solvers. On satis ability 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 quanti cation and substitution.

## *8.1 Future Directions*

*In this dissertation we have explored ways for doing formal veri cation us-* ing 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 bene cial 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 rst on a number of variables. Then switch to* Up All to nish the BED to BDD conversion. We have sug- gested Up One-minimizing in Section 3.2.3. This may be a start- ing 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 back-* tracking. As a result it sometimes gets stuck { especially when working on unsatis able problem instances. We expect that the addition of pre- mature 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 conver-* sion 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 xed-point. After each iteration* in the xed-point calculation we need to determine whether we have reached the xed-point or we should continue with another iteration. At the moment we convert the BEDs to either CNF or BDDs. How- ever, it might be worth using SAT-solvers (CNF conversion) in the beginning and then switch to BDD conversion near the xed-point. This would require some metric which tells us how close we are to the

*xed-point. One possible metric is the number of states in the set* di erence between two successive xed-point approximations in the

*xed-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 xed-point algo- rithms.

*Appendix A*

# *The BED Tool*

*NAME bed | tool for manipulating Boolean formulae as Boolean Ex-* pression Diagrams.

*SYNOPSIS*

*bed [-h] [-b m] [-c n] [-f script- le] [ lename] 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/*](http://www.it-c.dk/research/bed/)

*OPTIONS*

*Options may appear in any order as long as they appear before* the lename.

*165*

*-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-*

*le.*

*lename*

*Formula description, see le format below.*

*COMMANDS*

*The bed tool is used to manipulate Boolean formulae repre-* sented 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 di erent ways to do this: by us- ing 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 satis ability of a function repre- sented by a BED. This can be done with the bedsat command. A number of other commands are available to obtain informa- tion 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

*le.*

*upone input-list output-list*

*Takes each variable at a time (starting from the rst* 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 com- mand only works if the BED is transformed into a BDD

*rst. anynonsat node*

*Returns a non-satisfying assignment for Boolean func-*

*tion represented by node. This only works if the BED* is transformed into a BDD rst.

*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 rst.

*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 satis able* function (result is 1), or it is unsatis able (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 de ned 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

*expr ! ID*

*j (expr)*

*j expr binop expr*

*j expr < var > expr* j not expr

*j exists ID . expr j forall ID . expr*

*j expr [ ID := expr ]*

*binop ! and j or j biimp j nand j nor j xor j imp j limp j nimp j nlimp*

*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 impli- cation. The expression expr1 < var > expr2 is the if-then-else operator where expr1 denotes the low-child and expr2 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 op-* tions:

*support*

*value is either left or right. This option con-* trols whether the support command make a left-

*rst (low- rst) or right- rst (high- rst) 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 identi ers (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 alpha-* betically 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 when-* ever 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. De- fault 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 [ > lename]*

*Outputs the BEDs rooted at nodes in node-list to the*

*le lename 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/*](http://www.research.att.com/sw/tools/graphviz/)

*read lename*

*Reads a le containing a description of Boolean formu-* lae in the format described below. Any previously read circuit is discarded.

*write [ lename ]*

*Writes the BED to the le lename or to TMP.bed if no*

*lename is speci ed.* source lename

*Reads and executes a script of commands. The com-*

*mands must be separated by semicolons.* dump node - node

*Prints all nodes between the two nodes.*

*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 semi- colons.

*stat [ bed j hash j 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 j 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 speci ed, 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 nodes 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 \* de- noting 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 nally fanout( node ) which returns the Depth Fanout or- dering of inputs of node.

*BED FILE FORMAT*

*The bed tool supports a simple le format for storing BEDs.* The format is meant to be computer-friendly, not easy to use for humans.

*A le in this format consists of three parts: inputs, assign-* ments, 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 le format is:

|  |  |  |
| --- | --- | --- |
| *BED le* | *!* | *inputs assignments outputs* |
| *inputs* | *!* | *inputs input-list* |
| *input-list* | *!* | *input input-list j input* |
| *input* | *!* | *ID* |
| *assignments* | *!* | *assign assign-list* |
| *assign-list* | *!* | *assign assign-list j assign* |
| *assign* | *!* | *var-assign j op-assign j 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 j and j biimp j nand j nor j or j xor j* |
|  |  | *imp j limp jnimp j nlimp j ESUB* |
| *misc-op* | *!* | *:= j ? j !* |
| *outputs* | *!* | *outputs output-list* |
| *output-list* | *!* | *ID NO output-list j ID NO* |

*An ID is a series of one or more letters, numbers, and under-* scores, or any string in single quotes. A NO is a non-negative integer.

*Each assign de nes a BED vertex. The rst NO is a unique* identi er for the vertex. A vertex must be de ned earlier in the le than a vertex which refers to it. The terminal vertices

*0 and 1 are implicitly de ned and have unique identi ers 0 and* 1, respectively.

*var-assign represents a variable vertex. The rst NO is a* unique vertex identi er. The ID is the variable name. The last two NO's are the identi ers of the low and high children.

*op-assign represents an operator vertex. The rst NO is a*

*unique vertex identi er. The last two NO's are the identi ers* of the low and high children.

*misc-assign represents a vertex in the extended version of BEDs* described in Chapter 7. The rst NO is a unique vertex iden- ti er. The ID is a variable eld. The two last NO's are the identi ers of the low and high children.

*The operators imp, limp, nimp, nlimp, and biimp are impli-* cation, left implication, negated implication, negated left im- plication, and biimplication. The operators :=, ?, ! and ESUB correspond to substitution, existential quanti cation, universal quanti cation and ESUB. The rest of the operators have stan- dard 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) quanti cation assignment denotes a Boolean function in which an input is existentially (univer- sally) quanti ed. 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 speci ed as an ID and a NO. The ID is the* name of the output. The NO is the unique identi er for a vertex in the BED.

*A full-adder can be speci ed 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 di erent 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 veri ed 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 di erent 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 di erent value* for this assignment:

*eval s1 [ ci ] eval s2 [ ci ]*

*The rst command results in a 0, the second in a 1.*

*EXAMPLE 2*

*Consider verifying that two multi-output combinational cir-* cuits are equivalent. Assume that the two circuits are described in the le circuits.bed in the BED le format. Each output in circuits.bed is a biimplication of two corresponding out- puts from the original circuits. In other words, our veri cation task is to prove all outputs of circuits.bed to be tautologies.

*Write a script- le, script.com, containing the following com-* mands:

*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 rst command iterates through all outputs in circuits.bed. For each output it rst sets the variable or- dering 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 tau- tologies among the outputs. In this way we can see whether the veri cation 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 rst 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. E en. 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 satis ability 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 dia-* grams. Information and Computation. (To appear). 2, 2.2, 2.3, 2, 7

*[AH97] H. R. Andersen and H. Hulgaard. Boolean expression dia-* grams. 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 hard-* ware veri cation 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. Veri cation of arithmetic func-* tions 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

*177*

*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 veri cation. 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: 1020 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 Veri cation (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 combina-* tional 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 repre-* sented as formulas. Technical Report CS-1999-102, Chalmers University of Technology, Sweden, June 1999. 3.2.3

*[Bra93] D. Brand. Veri cation 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. EÆcient 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. Heuris-* tics to compute variable orderings for eÆcient manipulation of ordered binary decision diagrams. In Proc. ACM/IEEE De- sign 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 or-* dered 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: En-* abling technologies for formal veri cation. In Proc. Inter- national Conf. Computer-Aided Design (ICCAD), pages 236{ 243, November 1995. 2.4

*[CBM89] O. Coudert, C. Berthet, and J.C. Madre. Veri cation of syn-* chronous sequential machines using symbolic execution. In Proceedings of the International Workshop on Automatic Ver- i cation Methods for Finite State Systems, volume 407 of Lec- ture 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 Veri er. In N. Halbwachs and D. Peled, editors, Proceedings Eleventh Conference on Computer-Aided Veri cation (CAV'99), volume 1633 of Lec- ture 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 veri-*

*cation of nite-state concurrent systems using temporal logic*

*speci cations. 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 dia-* grams 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 quanti ed Boolean formulae. In Pro- ceedings of the 15th National Conference on Arti cial Intel- ligence (AAAI-98), pages 262{267, Menlo Park, July 1998.

*AAAI Press. 5.6*

*[CHP93] P.-Y. Chung, I. N. Hajj, and J. H. Patel. EÆcient 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. Inter- national 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 applica- tion to technology mapping. In Proc. ACM/IEEE Design Au- tomation 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 veri cation. In IFI WG 10.5 Advanced Re- search Working Conference on Correct Hardware Design and Veri cation Methods, Frankfurt, Germany, October 1995. 7

*[DBR97a] R. Drechsler, B. Becker, and S. Ruppertz. The K\*BMD: A* veri cation data structure. IEEE Design and Test of Comput- ers, 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 pro-* gram 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 quan-* ti cation theory. Journal of the ACM, 7:201{215, 1960. 5.6,

*6.2, 6.5*

*[DR97] Y. Dutuit and A. Rauzy. Exact and truncated computa-* tions 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 di- agrams. In Lydersen, Hansen, and Sandtorv, editors, Pro- ceedings of European Safety and Reliability Association Con- ference, 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. EÆcient representation and manipulation of switching functions based on ordered kronecker functional de- cision 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 im-* provements of boolean comparison method based on binary de- cision 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 Automa-* tion (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. Veri cation of arithmetic circuits by comparing* two similar circuits. In Computer Aided Veri cation (CAV), Lecture Notes in Computer Science, pages 159{168. Springer-

*Verlag, 1996. 1.3, 3.1*

*[GK00] M. Ganai and A. Kuehlmann. On-the- y compression of logi-* cal circuits. In Proc. of International Workshop on Logic Syn- thesis, 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 veri cation. In Proc. International Workshop on Logic Synthesis, 1997. 3.6

*[GM94] J. Gergov and C. Meinel. EÆcient 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, De-* partment 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 Arti cial 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 veri cation 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 eÆcient con-* struction of BDDs by reordering based synthesis. In IEEE European Design & Test Conference, 1997. 7, 7.5

*[HO82] C. M. Ho mann 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. Combi-* national logic-level veri cation using boolean expression di- agrams. 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 di- agrams. 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 veri cation* 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. Ex-* tended BDD's: Trading o canonicity for structure in veri ca- tion 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 Confer- ence (DAC), volume 34, pages 263{268, 1997. 3.6

*[KP94] W. Kunz and D. K. Pradhan. Recursive learning: A new impli-* cation technique for eÆcient solutions to CAD problems { test, veri cation, and optimization. IEEE Transactions on Com- puter 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 frame-* work for logic veri cation in a synthesis environment. IEEE Transactions on Computer Aided Design, 15(1):20{32, Jan- uary 1996. 10, 3.6

*[KSR92] U. Kebschull, E. Schubert, and W. Rosenstiel. Multilevel logic* synthesis based on functional decision diagrams. In Proc. Eu- ropean Conference on Design Automation (EDAC), pages 43{ 47, 1992. 7, 3.6

*[Kun93] W. Kunz. HANNIBAL: An eÆcient tool for logic veri ca-* tion 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 sys-* tems. Talk at Carnegie Mellon University, November 1999. For more information, visit Dr. Leveson's website at [http://sunnyday.mit.edu.](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 transfor- mation and function decomposition. IEEE Transactions on Computer Aided Design, 13(8):959{975, August 1994. 7, 7

*[Mat96] Y. Matsunaga. An eÆcient equivalence checker for combi-* national 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 com-* binatorial 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. M ller and J. Lichtenberg. Difference decision diagrams.* Master's thesis, Department of Information Technology, Tech- nical University of Denmark, Building 344, DK-2800 Lyngby,

*Denmark, August 1998. 2.4, 7*

*[MLAH99] J. M ller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard.* Di erence decision diagrams. In Proceedings 13th Interna- tional 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 satis ability. IEEE Transactions on Computers, 48, 1999. 14, 5.3.2, 23

*[MTJ+97] R. Mukherjee, K. Takayama, J. Jain, M. Fujita, J. A. Abra-* ham, and D. S. Fussell. Flover: Filtering oriented combina- tional veri cation 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 veri cation 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- rst manipula-* tion of very large binary-decision diagrams. In Proc. Interna- tional 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: Ver-* i cation 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* St almarck's algorithm. Talk at seminar given at Cambridge, March 1998. 3.6

*[Rin99] J. T. Rintanen. Improvements to the evaluation of quanti-*

*ed boolean formulae. In D. Thomas, editor, Proceedings of* the 16th International Joint Conference on Arti cial Intelli- gence (IJCAI-99-Vol2), pages 1192{1197. Morgan Kaufmann Publishers, August 1999. 5.6

*[Rud93] R. Rudell. Dynamic variable ordering for ordered binary deci-* sion 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 synthe-* sis. 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 manipula-* tion 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. Jour-* nal of Arti cial Intelligence Research (JAIR), 1:309{314, Jan- uary 1994. 5.6, 6.5, 8.1

*[Sht00] O. Shtrichman. Tuning SAT checkers for bounded model* checking. In Computer Aided Veri cation (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 satis ability problems. In P. Rosenbloom and

*P. Szolovits, editors, Proc. Tenth National Conference on Ar-* ti cial Intelligence, pages 440{446, Menlo Park, California, 1992. American Association for Arti cial 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/*](http://vlsi.colorado.edu/) *fabio. 10*

*[SS98] M. Sheeran and G. St almarck. A tutorial on St almarck'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 Com-* puter 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 dia-* grams and the Davis-Putnam procedure. In J.P. Jouannaud, editor, 1st International Conference on Constraints in Com- putational Logics, volume 845 of Lecture Notes in Computer Science, September 1994. 3.6

*[vE97] C.A.J. van Eijk. Formal Methods for the Veri cation 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 Au- tomation 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 veri cation 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 Bi-* nary Decision Diagrams. Kluwer Academic Publishers, 1996. 7, 7

*[WAH01] P. F. Williams, H. R. Andersen, and H. Hulgaard. Satis ability* 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. Com-* bining decision diagrams and SAT procedures for eÆcient sym- bolic model checking. In Computer Aided Veri cation (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 In- ternational Conference on Electronics, Circuits and Systems (ICECS), September 1999. 1.5, 4

*[Wil00] P. F. Williams. Formal Veri cation Based on Boolean Expres-* sion 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. Nikolska a, 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-eÆcient BDD construction by working set control. In Asian-Paci c Design Automation Conference (AS-

*PDAC), pages 423{432, February 1998. 3.5.1, 7.4.1*

*[Zha97] H. Zhang. SATO: An eÆcient propositional prover. In William* McCune, editor, Proceedings of the 14th International Confer- ence on Automated deduction, volume 1249 of Lecture Notes in Arti cial Intelligence, pages 272{275, Berlin, July 1997.

*Springer-Verlag. 15, 5. 3.2, 23*

# *Index*

*apply, 23, 45*

*atomic propositions, 94*

*balancing, 41*

*BDD, 18, 28*

*BED, 13*

*algorithms, 20*

*free, 14*

*path, 16*

*reduced, 14*

*size, 16*

*support, 16*

*tool, 165*

*binary decision diagram, see BDD*

*boolean expression diagram, see BED bounded model checking, 91*

*build, 84*

*cache, 23, 135*

*cell, 79*

*characteristic function, 6, 35, 80, 86,*

*99{101, 122*

*CNF, 73, 109, 132*

*combinational circuit, 33, 77*

*hierarchical, 77*

*computation, 95*

*computation tree logic, see CTL conjunctive normal form, see CNF connective, 4*

*container cell, 79*

*contradiction, 5*

*CTL, 95, 128*

*CTL\*, 128*

*cut, 81*

*cut-relation, 81*

*Davis-Putnam, 132, 133*

*decision diagrams, 28*

*decomposition, 28*

*depth, 48*

*ESUB operator, 149*

*existential quanti cation, 142 ESUB operator, 149 vector, 145*

*fault tree, 151*

*nite state machine, see FSM FixIt, 114{116, 118*

*formal veri cation, 3*

*formal world, 3*

*FSM, 93, 95, 121*

*garbage collection, 18 Grasp, 92, 109, 127*

*hardness, 54*

*initial state, 102 input cut, 81*

*input relation, 81*

*instantiation, 80*

*interfaces, 8*

*ISCAS'85 benchmark, 43, 57*

*iterative squaring, 105*

*Kripke structure, 93, 121*

*lfp-CTL, 120*

*LGSynth'91 benchmark, 63 literal, 153*

*logic, 4*

*logic cell, 79*

*LTL, 128*

*189*

*minimal p-cut, see p-cut minimizing, 41*

*minterm, 153*

*model checking, 95, 101*

*monotonic, 97, 122*

*multiplier, 115, 156*

*negation normal form, 121 negative Davio, 28*

*NuSMV, 114, 115, 117, 118, 120*

*operator sets, 36*

*output cut, 81*

*output relation, 81*

*p-cut, 151, 153, 154, 157*

*path, 16, 81, 95*

*PC operator, 151, 157 positive Davio, 28 prime implicant, 153*

*product, 153*

*proof, 4*

*propagate, 84*

*property, 4*

*propositional logic, 4, 33, 50*

*pruning, 42*

*QBF, 5, 129*

*quanti cation by substitution, 103 quanti ed boolean formulae, see QBF quanti er, 5, 103*

*reactive system, 94*

*real world, 3*

*recursive learning, 62, 73*

*Reed-Muller, 28*

*reliability, 8*

*rewriting, 37*

*safety, 8*

*satis ability, 5, 108*

*satis ability problem, 5, 20*

*Sato, 92, 109, 114, 117, 127*

*scope reduction, 106 set inclusion, 104*

*set transformer, 97, 122*

*Shannon, 28*

*simpli cations, 36*

*SMV, 102, 110*

*speci cation, 4, 95*

*St almarck's method, 42, 50, 66*

*STE, 129*

*strong until, 96*

*substitution, 144, 156*

*vector, 147*

*symbolic trajectory evaluation, see STE system, 4*

*tautology, 5*

*tautology problem, 5, 20, 35, 131*

*temporal logic, 95, 128*

*transition, 94*

*transition function, 94, 102*

*transition relation, 94*

*universal quanti cation, 143*

*vector, 146*

*unsatis ability, 5*

*up, 25*

*up all, 23, 45*

*up one, 22, 45*

*variable ordering, 45*

*Depth Fanout, 49 Fanin, 48*

*weak until, 96*