ACHIEVING SCALABLE HARDWARE VERIFICATION WITH SYMBOLIC SIMULATION

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Valeria Bertacco

August 2003

© Copyright by Valeria Bertacco 2003
All Rights Reserved

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

Kunle Olukotun
(Principal Adviser)

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

David Dill

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

Mark Horowitz

Approved for the University Committee on Graduate Studies.

Abstract

In the past 40 years, electronic systems have become pervasive in modern society. Digital integrated circuits (ICs) are at the heart of the large majority of these systems. These digital ICs are complex systems containing millions of interconnected transistors in a very small area. Moreover, the underlying semiconductor fabrication technology used to fabricate these ICs allows doubling the number of transistors in the same area approximately every 18 months.

The design of digital systems is a complex and time consuming process that progresses through various phases and levels of abstraction, and relies heavily on CAD (Computer-Aided Design) software tools. Within this context, ensuring the correctness of these digital systems is a major consideration, especially because the cost of failures is becoming increasingly high. One of the most famous recent examples of its importance is the Intel, Inc. Pentium's flaw in the floating point divide unit of 1994 that eventually forced Intel to replace all the Pentium chips that were already in the market. In many cases, the possibility of failure is even unacceptable, examples of these applications are: transportation systems, medical applications and financial systems. Even though guaranteeing the correctness of a design is such a central aspect in its development, current verification methodologies are still inadequate to tackle the complex systems that are being developed nowadays. Hardware design companies try to compensate for mediocre CAD tools by dedicating the majority of their resources involved in a design to verification, yet are still unable to guarantee correct functionality over the entire design space.

Logic simulation is the most widely accepted method for ensuring the correctness of digital ICs in industry because of its scalability, flexibility and predictable run-time behavior. This technique is

based on verifying a digital system by providing sequences of binary values for each of the inputs of the system and checking that the corresponding outputs are correct, based on what the design team expected or described in a specification document. However, because of its inherent approach, this validation technique usually can visit only a small fraction of all the possible configurations of a system - also called the state space - and thus the discovery of bugs heavily relies on the expertise of the designer of the test stimuli to select a few crucial configurations to verify. Symbolic simulation is another verification method that is attracting increasing interest because it allows the verification engineer to explore all, or a major portion, of a circuit's state space without the need to design time-consuming test stimuli. However, this approach poses a high demand on the resources of the simulating host, and in particular, on the memory system, because of the complexity of the algorithms involved and their unpredictable run-time behavior. Thus, the scalability of this approach has been the main limiting factor to its mainstream deployment and so far its scope has been limited to small systems.

This thesis presents new symbolic simulation based approaches to the verification problem that radically improve scalability. We present two new techniques that narrow the performance gap between the complexity of digital systems that are being developed and the limited ability to verify them. The first technique, Cycle-Based Symbolic Simulation, is a unique combination of formal methods and logic simulation that can stimulate a circuit with a very large number of input combinations and sequences in parallel. The key concept is the use of a parametric form to represent the set of states visited during simulation. This approach maintains a high degree of scalability, in line with current cycle-based logic simulation techniques, while achieving better efficiency. To better exploit the use of parameterization in improving the memory profile of simulation, the second technique, Disjoint Support Decomposition Based Symbolic Simulation, exploits the disjoint support decomposition properties of the state functions. We develop a new algorithm that exposes the disjoint decomposition properties of a Boolean function by restructuring its BDD representation. The new algorithm is very efficient in the sense that it has worst-case complexity that is only quadratic in the size of the initial BDD, while previous algorithms had exponential complexity in the size of

the function's support. We deployed this algorithm to find the disjoint support decomposition of the state functions in symbolic simulation. By restructuring the next-state functions using their disjoint support components, it is possible to gain better insight about the role of each input variable. Consequently, the next-state functions can be transformed into a simpler parametric form without sacrificing simulation accuracy. Both of these techniques have been tested on the ISCAS benchmark suite. The results show that the first technique can simulate very large trace sets in parallel, maintaining a simulation speed and memory profile that are much closer to logic simulation. The second technique is effective in reducing the memory requirements of symbolic simulation while maintaining exact state exploration.

Acknowledgements

I would like to first thank my graduate advisor Kunle Olukotun. Throughout these years, he has always been prompt and available in supporting whatever direction of research and of life I decided to pursue. In our technical interactions, he would always go straight to the results of my work and challenge me on their practical contribution to the quality of verification for industrial scale digital designs. David Dill has been the person I could always go to for bouncing ideas and have illuminating technical discussions. When my ideas could survive his dissecting analysis, I knew I could publish them. On a personal level, I always admired his bluntness that would eliminate a lot of useless conversation in our interactions. Thanks also to Mark Horowitz for being part of my defense committee and reviewing this thesis even though it is not central to his research area.

My years in Synopsys have played a central role in shaping my understanding of design verification as an industrial challenge first and a research area later. My colleagues have been crucial in providing me with invaluable opportunities: Ghulam Nurie, Swami Venkat and the marketing team of the Vera Group allowed me to interact with customers in meetings that have always been enlightening in my quest towards an understanding of the needs of the hardware designers. Pei-Hsin Ho, my manager in the Advanced Technology Group of Synopsys, gave me the chance to be part of a high-profile technical team and take part in seminars and technical conferences, all while never losing sight of the objective of providing solutions for the design industry. Most of all, he showed me how to efficiently achieve technology transfer, taking academic research and deploying it in software solution for the hardware design community. I would like to thank my other colleagues in the Advanced Technology Group, in particular: Stephen Edwards, Thomas Shiple, James Kukula,

David Cyrluk, Tony Ma, Kevin Harer, Jerry Taylor, Randy Harr and Robert Damiano.

I spent the past year at Stanford completing my PhD work. During this time I shared my office with John Davis. John has created a very positive work environment for me, he has always provided good advice and been very helpful. He has been crucial especially during the preparation of my oral defense talk for which he provided countless bits of advice, asked me all the most difficult questions and forced me to rehearse it until it would flow seamlessly, by which point he could give the talk himself. I would also like to thank all the people that supported me by making available all those resources that are involved in putting together a thesis. My thanks go especially to: Charlie Orgish, Darlene Hadding, Lance Hammond and Azita Emami-Neyestanak. My undergraduate advisor, Maurizio Damiani, first introduced me to research and to the area of Computer Aided Design for integrated circuits. I would like to thank him for the numerous interactions and collaborations that lasted long after my undergraduate studies and spurred many of the publications that led to this research work. The material presented in Chapters 4 and 5 has been shaped by many months of intense discussions with him.

On a personal level, I would like to thank my parents for teaching me the first concepts of mathematics and logic and for introducing me early in my life to pursuing both education and industry experience, contrary to the Italian tradition of completing all the studies before gaining any work experience. I also want to thank my family for supporting my choices in my path through life. My brother Livio provided all sort of technical support and advice and solved many system crashes, most often connecting from some remote location around Europe. Finally, I thank all my friends in the Stanford community who provided me with enthusiastic social entertainment during my years at Stanford.

As this work comes closer to completion, I look forward to new research in the years to come that will hopefully both have practical use and be intellectually stimulating. Thus, I see this dissertation more as a stepping stone in my research work than as the end of my efforts.

Contents

Abstract **v**

Acknowledgements **ix**

1 Introduction **1**

1.1 Functional validation 2

1.2 Formal verification 3

1.2.1 Symbolic simulation 4

1.3 Contributions of the thesis 5

1.4 Organization of the thesis 6

2 Design and verification of digital systems **9**

2.1 The design flow 10

2.2 RTL verification 14

2.3 Boolean variables and functions and their representation 17

2.3.1 Binary Decision Diagrams 18

2.4 Models for design verification 21

2.4.1 Structural network model 21

2.4.2 State diagrams 23

2.4.3 Mathematical model of Finite State Machines 25

2.5 Functional validation 26

2.6 Formal verification 31
2.6.1 Symbolic Finite State Machine traversal 32
2.7 Symbolic Simulation 35
2.7.1 The algorithm 37
2.7.2 The challenge in symbolic simulation 41

3 Cycle-Based Symbolic Simulation 43

3.1 Parametric transformations 43
3.2 Parameterizations in symbolic simulation 46
3.3 The CBSS algorithm 47
3.4 The parameterization phase 49
3.4.1 Using functional dependencies 50
3.4.2 How to classify the components of the state vector 53
3.4.3 The `remap` function 57
3.5 Implementation and complexity 59
3.6 Experimental results 61
3.7 Conclusion 65

4 Disjoint Support Decompositions 67

4.1 Introduction 68
4.2 Related work on Disjoint Support Decompositions 69
4.3 Terminology 71
4.3.1 Decomposition trees. 73
4.4 The unique maximal Disjoint Support Decomposition 73
4.4.1 Decomposition by prime functions. 75
4.4.2 A characterization of F/K_F. 78
4.4.3 The normal Decomposition Tree 86
4.5 On the decomposability of Boolean functions 92

5 A novel algorithm for Disjoint Support Decompositions **95**

5.1 Building the decomposition bottom-up 96

5.2 Case 1. Neither A_{10} nor A_{11} is constant and $A_{10} \neq \overline{A_{11}}$ 99

5.3 Case 2. Exactly one of A_{10}, A_{11} is constant 103

5.4 Case 3. $A_{10} = \overline{A_{11}}$ and A_{10} is not a constant 106

5.5 New decompositions 108

5.6 Putting it all together: The DSD procedure 117

5.6.1 Inherited decompositions 120

5.6.2 New decompositions 125

5.7 Complexity analysis and considerations 127

5.8 Experiments on the decomposability of industrial testbenches 129

5.9 Conclusion 136

6 Exact Parameterizations for Symbolic Simulation **137**

6.1 Re-encoding the state function 138

6.2 Reduction at Free Points 140

6.3 Elimination of Prime functions 143

6.4 Removal of non-dominant variables 146

6.5 DSD-SS Implementation 150

6.6 Experimental results 151

6.7 Summary 155

7 Conclusion **157**

7.1 Parameterized approaches in symbolic simulation 157

7.2 Disjoint support decompositions 158

7.3 The future of this work 159

Bibliography **161**

List of Tables

3.1 Cycle Based Symbolic Simulation results 62

5.1 Disjoint Support Decomposition results 131

6.1 Disjoint Support Decomposition-based simulation results 154

List of Figures

2.1 Design flow of a digital system 11
2.2 Approaches to verification 16
2.3 Binary Decision Diagrams 19
2.4 Graphic symbols for some basic logic gates 22
2.5 Structural network model 22
2.6 Network model of a 3-bits up/down counter with reset 23
2.7 State diagram of a 3-bit up/down counter 24
2.8 State diagram of a 1-hot encoded 3-bit counter 25
2.9 Compiled logic simulator 27
2.10 Logic simulation - pseudocode 29
2.11 FSM state traversal - pseudocode 35
2.12 Logic and symbolic simulation 36
2.13 Symbolic simulation algorithm - pseudocode 38
2.14 Symbolic simulation for Example 2.8 - Initialization phase 38
2.15 Symbolic simulation for Example 2.8 - Simulation Step 2 40
2.16 Iterative model of symbolic simulation 40

3.1 Parameterization of the state vector during symbolic simulation 44
3.2 Three steps of symbolic simulation for the counter of Example 2.2 and possible parameterizations of the reached state sets 46

3.3 Cycle-Based symbolic simulation flow 49
3.4 The CBSS algorithm - pseudocode 50
3.5 The parameterized frontier subset $\mathbf{PS}_{@\mathbf{k}}$ 51
3.6 `parameterize` function - pseudocode 53
3.7 Classifying simple and complex variables - pseudocode 55
3.8 Classifying shared variables - pseudocode 57

4.1 Decompositions for Example 4.1 68
4.2 A decomposition tree for Example 4.3. 74
4.3 Decomposition representation of the function of Example 4.7 88
4.4 Decomposition tree for Example 4.8. 89

5.1 PRIME decomposition. 111
5.2 Function for Example 5.6. 115
5.3 Two functions and the construction of their $Max(G,H)$ tree. 127

6.1 A decomposed state vector for a small design 139
6.2 The parameterized frontier set $\mathbf{PS}_{@\mathbf{k}}$ 140
6.3 A vector function and its free points 141
6.4 Free points elimination for Example 6.1 143
6.5 General case for prime function elimination: (a) before and (b) after 145
6.6 Prime elimination for Example 6.2. 146
6.7 Non-dominant variable removal for Example 6.4 150

7.1 Trade-offs of in the breadth vs. scalability plane 159

Chapter 1

Introduction

In the past decade, the semiconductor industry has experienced a challenging evolution in the complexity of digital integrated circuit (IC) designs: increasing integration density and die size has made it possible to design chips with hundreds of millions of transistors. At the same time, the growing importance of getting products to market quickly has increased the pressure on design teams to deliver new products and new technologies in a short time span: typical development times are less than two years. In this fast evolving landscape, ensuring that the digital ICs are functionally correct is crucial: an error in the design's functionality can delay product deployment by months. Moreover, ICs are embedded in many safety critical applications, where a design flaw can lead to the loss of life.

Due to the importance of design correctness, a significant fraction of engineering development time and resources are devoted to it. Design verification involves checking that the initial functional design of a circuit is correct against the specifications. It consists of a whole set of activities aimed at acquiring a reasonable certainty level that a circuit will function correctly, under the assumption that no manufacturing fault is present. Validating the functionality of digital circuits and systems is an increasingly difficult task. Multiple chip design projects are reporting that approximately 70% of their design time is spent in verification. This is due to the growing complexity of the designs that has not been accompanied by improvements in functional verification techniques.

Part of this high resource allocation is due to the fact that verification methodologies are still very experimental, there is almost no standard approach or methodology in any area pertaining to verification, and the whole process is still largely manual. The cause of this high investment cost can be attributed to the high complexity of the task at hand, but also to the lack of support from the design automation industry. While on the design synthesis front, they have made available tools that can, at least partially, support the complexity and the challenges of such highly integrated designs, on the verification front there has been almost a complete lack of support. The only widely deployed tools for verification are logic simulators. Such simulators are a key tool for the verification team to gain insight in the actual functionality of the design under test; nonetheless, they cannot be used to guarantee the general correctness of any aspect of a design, and thus, their usefulness as push-button verification tools is still limited.

1.1 Functional validation

Designers normally try to ensure correctness by developing multiple set of tests to stimulate the digital design and by inspecting the results of the simulations. These techniques are the only ones available today that can cope with the complexity and scale of current digital ICs; at the same time they have significant limitations.

Today, logic simulation is the mainstream approach for the validation of large synchronous systems because of its scalability : CPU time is proportional to the design size and test length. Simulation is also flexible: practical cycle-based simulators allow for circuits with multiple clocks and the ability to mix cycle-based and event-based simulation. Unfortunately, the fraction of the design space which can be explored by simulation is miniscule, especially for large designs. Only one state and one input combination of the design under test are visited during each simulation cycle. Moreover the test stimuli must be hand crafted by the designer to cover those areas of the design that she wishes to validate. For a large, complex system, it is impossible to test or simulate all possible inputs or sequences of inputs. One measure of the quality of verification for a design

that is commonly used in industry is state coverage. State coverage counts how many different configurations of a system have been visited, and thus verified, by the simulation. When the size of the design space, or total number of reachable configurations, is known, the state coverage can be expressed as a fraction of this size. Furthermore, simulation inputs are usually based on the design specification and thus are only aimed at verifying that the design performs all the primary activities indicated in the specification document. However, it is often the case that complex systems manifest unforeseen behavior for corner case situations that were not planned in the specification. Most often, designers are unaware of behavior that results as a by-product of the interactions among different modules and that was unaccounted for in the specification document. Thus these cases do not get checked, while they may as well have negative consequences on the overall behavior of the system.

Overall, designers are discovering that their simulation-based verification approaches are inadequate as ICs become more complex through increased size, more aggressive pipelining, and greater use of concurrency.

1.2 Formal verification

In its broadest meaning formal verification consists of proving formally that the implementation of a digital IC is compatible with its specification. In a formal verification approach, the desired functionality of the system needs to be completely specified, then a formal model of the system needs to be constructed – the implementation – and finally, formal reasoning is used to show that this formal model satisfies the specification. Formal verification techniques have the potential of providing more general results than traditional validation methods: it is possible for instance to guarantee that a specific property holds for a design under all possible input stimuli. Due to the complexity of constructing a complete specification and of formally proving the compatibility between implementation and specification, this approach is infeasible for state-of-the-art hardware designs.

Traditionally formal methods have been mainly explored in academic research settings and only

applied to problems of very limited size. However, the recent "verification crisis" – that is the inability of current validation techniques to provide sufficient confidence in the correctness of the design – has spurred increased interest for this approach to verification which has led to new algorithmic solutions and to new approaches that compromise on the completeness of the verification in order to reduce its complexity. In particular, the past ten years have seen efforts in developing commercial formal verification tools; however, so far these tools have not shown the required robustness to be included in the industry mainstream verification methodology, they have only been applied to experimental projects. One of the main limitations shown by these first attempts are in the complexity of the algorithms involved in formal verification: usually their demand for computing resources far exceeds the resources available at most design sites. Another limitation has been the amount of engineering effort that is required for providing a complete formal specification of the design.

As a consequence, formal techniques have only be applied to very simple designs that do not represent the complexity of the digital ICs developed in industry.

1.2.1 Symbolic simulation

Symbolic simulation is a promising approach to formal verification. The key idea is to simulate the design using Boolean symbolic variables instead of constant binary values at the combinational inputs of the circuit's model. During simulation, the approach derives Boolean expressions based on the initial symbolic variables and the functionality of each of the circuit components. At the end of each simulation step, we obtain a set of Boolean expression representing implicitly all configurations – or set of states – that are reachable by the circuits in one clock cycle with an appropriate set of inputs. Thus, this approach allows the complete behavior of a design in a specific state to be verified with a single simulation step, under all possible inputs simultaneously. Thus, it has the potential of 1) verifying many configurations of the design in parallel and providing much better coverage than traditional logic simulation. 2) providing the ability to prove time-bound properties of the design.

The problem with this approach is that it requires extensive manipulation of Boolean expressions, which in turn, often exhaust the memory resources of the host computer even on designs of limited complexity.

1.3 Contributions of the thesis

This thesis addresses the robustness and scalability limitations of symbolic simulation and presents two algorithms that dramatically reduce the memory requirements compared to current techniques.

The first technique, called Cycle-Based Symbolic Simulation, simplifies the Boolean expressions involved in symbolic simulation while trying to maximize the range that they span. The resulting simulator maximizes the level of parallelism achievable with a limited amount of memory. This technique performs very well from a scalability standpoint, and achieves a high level of parallelism in terms of test vectors run through the simulator, while maintaining a low memory profile. We found, however, that a better parameterization technique was needed in order to support the complete state exploration that is typical of symbolic simulation. To this end, we introduced an efficient algorithm that exposes the disjunctive support decomposition properties of a Boolean function.

The disjoint support decomposition of a scalar function $F : \mathcal{B}^m \rightarrow \mathcal{B}$, consists of finding other, simpler functions G and H such that: $F(x_1, \cdots, x_m) = G(H(x_1, \cdots, x_h), x_{h+1}, \ldots, x_m)$. An exact solution to this problem that had exponential complexity in the number of variables of the function, was proposed in the late '50s. The algorithm we present in this thesis takes as input a binary decision diagram (BDD) representation of a Boolean function and restructures it in its disjoint decomposition components. The worst case complexity of the algorithm we propose is only quadratic in the size of this representation, making it well suited for use with complex functions.

This algorithm is applied to transform and simplify the Boolean expressions involved in symbolic simulation so that the simulation requires fewer memory resources while producing the same original quality of results. Experimental results show that these solutions provide often more than 10

orders of magnitude better performance (in test vectors per second) than a logic simulator, while, at the same time, improve the scale of symbolic simulation by handling up to thousands more symbolic variables.

1.4 Organization of the thesis

In order to provide the context for our work, we present a quick overview of the steps involved in the design cycle of a digital IC in Chapter 2. The chapter also presents the models of digital systems used in verification and the main algorithms for both traditional simulation and formal verification.

We then present the first technique mentioned in the previous section, *Cycle-Based Symbolic Simulation* in Chapter 3. The chapter provides a formal presentation of the algorithm and simulation results that compare the performance of CBSS to that of a logic simulator.

In the following two chapters, we move away from the main topic of verification to focus on a core topic of Boolean algebra, the theory of Disjoint Support Decompositions (DSD). This theory is the central idea behind our second symbolic simulation technique. We decided to introduce it only at this point, so that the reader has a chance of seeing the type of approach that we take at verification with our first technique, before diving into core theoretical material. The first of these two chapters introduces the main results of the *Disjunctive Support Decomposition theory* of Boolean functions, the second presents our novel algorithm for decomposing the BDD of a function in its disjoint support components. Chapter 5 discusses the algorithm in detail and shows results on the decomposability of many Boolean functions involved in indutrial benchmarks.

This theory and novel algorithm are then deployed in Chapter 6 to present a new symbolic simulation algorithm based on the DSD properties of the state vector functions of simulation. This approach is called *Disjoint Support Decomposition based Symbolic Simulation*. The results compare this approach to a plain symbolic simulator. We conclude the thesis with Chapter 7, where we provide a discussion of the methods presented in the thesis and some directions for future research.

Each of the chapters starts with a presentation of our objectives for the chapter and a review of

the previous research developed on its specific topic. The central part covers a formal presentation of the material. When a chapter presents a new algorithm, we conclude presenting simulation results obtained by implementing the algorithm and testing it on indutrial testbenches.

Chapter 2

Design and verification of digital systems

Before diving into the discussion of the various verification techniques, we are going to review how digital ICs are developed. During its development, a digital design goes through multiple transformations from the original set of specifications to the final product. Each of these transformations corresponds, coarsely, to a different description of the system, which is incrementally more detailed and which has its own specific semantics and set of primitives. This chapter provides a high level overview of this design flow in the first two sections. We then review the mathematical background (Section 2.3) and cover the basic circuit structure and finite state machine definitions (Section 2.4) that are required to present the core algorithms involved in verification.

The remaining sections presents the algorithms that are at the core of the current technology in design verification. Section 2.5 covers the approach of *compiled level logic simulation*. This technique was first introduced in the late 80's and it is still today the industry's mainstream verification approach. Section 2.6 provides and overview of formal verification and of a few of its more successful solutions; within this context Section 2.7 focuses on providing a more detailed presentation of *symbolic simulation*, since this technique will be at the basis of the novel solutions introduced by this thesis.

2.1 The design flow

Figure 2.1 presents a conceptual design flow from specifications to final product. The flow in the figure shows a top-down approach that is very simplfied: as we discuss later in this section, the reality of an industrial development is much more complex, involving many iterations through various portions of this flow, until the final design converges to a form that meets the specification requirements of functionality, area, timing, power and cost. The design specifications are generally presented as a document describing a set of functionalities that the final solution will have to provide and a set constraints that it must satisfy. In this context, the *functional design* is the initial process of deriving a potential and realizable solution from this design specifications and requirements. This is sometimes referred to as modeling and includes such activities as hardware/software tradeoffs and a micro-architecture design.

Because of the large scale of the problem, the development of a functional design is usually carried out using a hierarchical approach, so that a single designer can concentrate on a portion of the model at any given time. Thus, the architectural description provides a partition of the design in distinct modules, each of which contributes a specific functionality to the overall design. These modules have well defined input/output interfaces and protocols for communicating with the other components of the design. Among the results of this design phase is a high level functional description, often a software program in C or similar programming language, that simulates the behavior of the design with the accuracy of one clock cycle and reflects the module partition. It is used for performance analysis and also as a reference model to verify the behavior of the more detailed designs developed in the following stages.

From the functional design model, the hardware design team proceeds to the *Register Transfer Level (RTL) design* phase. During this phase, the architectural description is further refined: memory element and functional components of each model are designed using an Hardware Description Languages (HDL). This phase also sees the development of the clocking system of the design and architectural trade-offs such as speed/power.

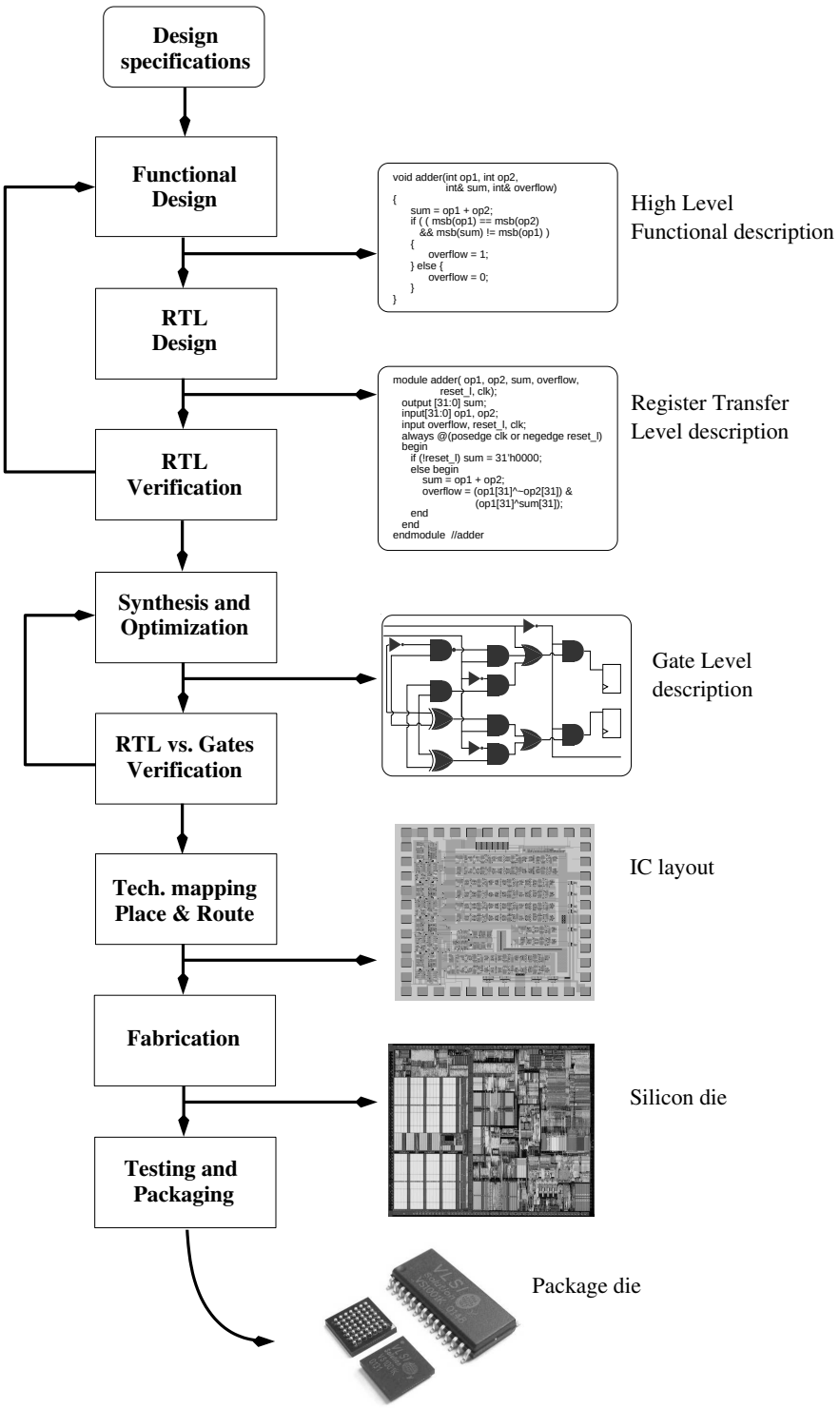

Figure 2.1: Design flow of a digital system

With the RTL design, the functional design of our digital system ends and its verification begins. *RTL verification* consists of acquiring a reasonable confidence that a circuit will function correctly, under the assumption that no manufacturing fault is present. The underlying motivation is to remove all possible design errors before proceeding to the expensive chip manufacturing. Each time functional errors are found the model needs to be modified to reflect the proper behavior. During RTL verification, the verification team develops various techniques and numerous suites of tests to check that the design behavior corresponds to the initial specifications. When that is not the case, the functional design model needs to be modified to provide the correct behavior specified and the RTL design updated consequently. It is also possible that the RTL verification phase reveals incongruences or overlooked aspects in the original set of specifications and this latter one needs to be updated instead.

In the diagram of Figure 2.1, RTL verification appears as one isolated phase of the design flow. However, in practical designs, the verification of the RTL model is carried on in parallel with the other design activities and it often lasts until chip layout. An overview of the verification methodologies that are common in today's industrial developments is presented in the next section.

The next design phase consists of the *Synthesis and optimization* of the RTL design. The overall result of this phase is to generate a detailed model of a circuits which is optimized based on the design constraints. For instance a design could be optimized for power consumption or the size of its final realization (IC area) or for the ease of testability of the final product. The detailed model produced at this point describes the design in terms of its basic logic components, such as *AND*, *OR*, *NOT* or *XOR* and memory elements. Optimizing the netlist, or gate level description, for constraints such as timing and power requirements is an increasingly challenging activity for current developments and it usually involves multiple iterations of trial-and-error attempts before it converges to a solution that satisfies both these requirements. Such optimizations may in turn introduce functional errors that require additional RTL verification.

While all the design phases, up to this point, have minimal support from Computer Aided Design (CAD) software tools and are almost entirely hand crafted by the design and verification team,

starting from synthesis and optimization, most of the activities are semi-automatic or at least heavily supported by CAD tools. Automating the RTL verification phase, is the next challenge that the CAD industry is facing in providing full support for digital systems development.

The synthesized model needs to be verified. The objective of *RTL versus gates verification*, or equivalency checking, is to guarantee that no errors have been introduced during the synthesis phase. It is an automatic activity requiring minimal human interaction that compares the pre-synthesis RTL description to the post-synthesis gate level description in order to guarantee the functional equivalence of the two models.

At this point, it is possible to proceed to *technology mapping* and *placement and routing*. The result is a description of the circuit in terms of a geometrical layout used for the fabrication process. Finally the design is *fabricated*, and the microchips are *tested and packaged*.

This design flow is obviously a very ideal, conceptual case. For instance, usually there are many iterations of synthesis, due to changes in the specification or to the discovery of flaws during RTL verification. Each of the new synthesized version of the design needs to be put through again all the subsequent design phases. One of the main challenges faced by design teams, for instance, is in satisfying the ever increasing market pressure to produce designs with faster and faster clock cycles. These tight timing specifications force engineering teams to push the limits of their designs by optimizing them at every level: architectural, in the components choice and sizing, and in placement and routing. Achieving timing closure, that is, developing a design that satisfies the timing constraints set in the specifications while still operating correctly and consistently, most often requires optimizations that go beyond the abilities of automatic synthesis tools and forces the engineers to intervene manually, at least in some critical portions of the design. Often, it is only possible to check if a design has met the specification requirements after the final layout has been produced. If these requirements are not met, the engineering team comes up with alternative optimizations or architectural changes and creates a new model that needs to go through the complete design flow.

2.2 RTL verification

As we observed in the previous section, the correctness of a digital circuit is a major consideration in the design of digital systems. Given the extremely high and increasing costs of manufacturing microchips, the consequences of flaws going unnoticed in system designs until after the production phase, would be very expensive. At the same time, RTL verification is still one the most challenging activities in digital system development: as of today, it is still carried on mostly with ad-hoc tests, scripts and often even tools developed by the design and verification teams specifically for the current design. In the best case, these verification infrastructure development can be amortized among a family of designs with similar architecture and functionality. Moreover, verification methodology still lacks any standard or even a commonly accepted approach, with the consequence that each hardware engineering team has its own distinct verification practices which often change with subsequent designs by the same team, due to the insufficient "correctness confidence level" that any of the current approaches provide. Given this scenario, it is easy to see why many digital IC development teams report that more than 70% of the design time and engineering resources are spent in verification, and why verification is thus the bottleneck in the time-to-market for integrated circuit development [5].

The workhorse of the industrial approach to verification is *functional validation*. The functional model of a design is simulated with meaningful input stimuli and the output is checked for the expected behavior. The model used for simulation is the RTL description The simulation involves applying patterns of test data at the inputs of the model, then using the simulation software or machine to compute the simulated values at the outputs and finally checking the correctness of the values obtained.

Validation is generally carried on at two levels: module level and chip level. The first verifies each module of the design independently. It involves producing entire suites of *stand alone tests*, each of which checks the proper behavior of one specific aspect or functionality of that module. Each test includes a set of input patterns to stimulate the module and a portion that verifies that the

output of the module corresponds to what is expected. The design of these tests is generally very time consuming, since each of them has to be handcrafted by the verification engineering team. Moreover their reusability is very limited because they are specific to each module. In general, the verification team develops a separate test suite for each functionality described in the original design specification document. Recently, a few CAD tools have become available to support functional validation: they mainly provide more powerful and compact language primitives to describe the test patterns and to check the outputs of the module, thus saving some test development time [34, 37, 5].

During chip level validation, the design is verified as a whole. Often this is done after a fair confidence is obtained about the correctness of each single module, and the focus is mainly in verifying the proper interaction between modules. This phase, while more compute intensive, has the advantage of being carried on in a more automatic fashion. In fact, input test patterns are often randomly generated, with the only constraint of being compatible with what the specification document define to be the proper input format to the design. During chip level validation it is usually possible to simulate both the RTL and a high level description of the design simultaneously and check that the outputs of the two systems and the values stored in their memory elements match one to one, at the end of each clock cycle.

The quality of all these verification efforts is usually analytically evaluated in terms of coverage: a measure of the fraction of the design that has been verified [43, 50]. Functional validation can provide only partial coverage because of its approach; the objective is thus to maximize coverage for the design under test. Various measures of coverage are in use: for instance *line coverage* counts the lines of the RTL description that have been activated during simulation. Another type is *state coverage* which measures the number of all the possible configurations of a design that have been simulated, that is, validated. This measure is particularly valuable when an estimate of the size of the total state space of the design is available: in this situation the designer can use state coverage to quantify the fraction of the design that she has verified.

With the increasing complexity of industrial designs, the fraction of the design space that the functional validation approach can explore is becoming vanishingly small, and it is showing more

and more that it is an inadequate solution to the verification problem. Since only one state and one input combination of the design under test are visited during each step of simulation, it is obvious that neither of the above approaches can keep up with the exponential growth in circuit complexity.

Because of the limitations of functional validation, new alternative techniques have received increasing interest. The common trait of these techniques is the attempt to provide some type of mathematical proof that a design is correct, thus guaranteeing that some aspect or property of the circuit behavior holds under every circumstance, and thus its validity is not limited only to the set of test patterns that have been checked. These techniques go under the name of *formal verification* and have been studied mostly in academic research settings for the past 25 years. Formal verification constitutes a major paradigm shift in solving the verification problem. As Figure 2.2 shows, with logic simulation we probe the system with a few handcrafted stimuli, while with formal verification we show the correctness of a design by providing analytical proofs that the system is compatible with each of the specifications. Compared to a functional validation approach, this is equivalent to simulating a design with all possible input stimuli and thus to providing 100% coverage.

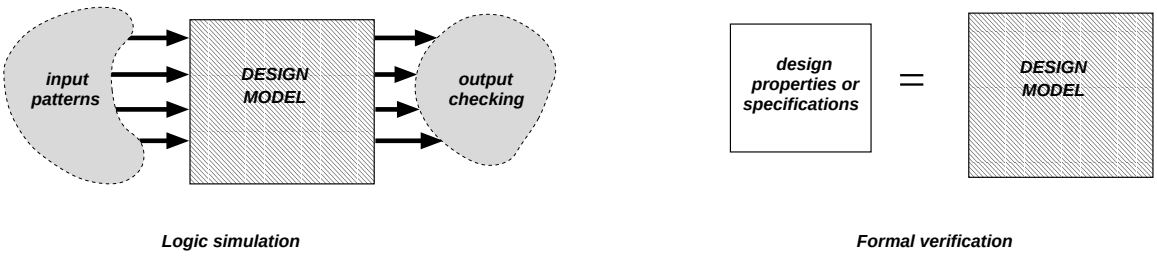

Figure 2.2: Approaches to verification

It is obvious that the promise of such thorough verification, makes formal verification a very appealing approach. While on one hand the solution to the verification problem seems to lie with formal verification approaches, on the other hand, these techniques have been unable to tackle industrial designs due to the complexity of the underlying algorithms, and thus have been applicable only to smaller components. They have been used in industrial development projects only at an experimental level, and so far they generally have not been part of the mainstream verification

methodology.

We now overview some of the fundamental methods for validation and verification to set the stage for the new techniques presented in this thesis. Before diving into this, we briefly review some mathematical concepts and the models and abstractions of digital systems used by these techniques.

2.3 Boolean variables and functions and their representation

We review here a few basic notions on Boolean algebra to set the stage for the following presentation.

Let $\mathcal{B}$ denote the Boolean set $\{0,1\}$. A symbolic variable is a variable defined over $\mathcal{B}$. A logic function is a mapping $\mathbf{F}:\mathcal{B}^n \rightarrow \mathcal{B}^m$. Hereafter, lower-case and upper-case letters will denote logic variables and functions, respectively. We will be mostly concerned with **scalar** functions $F(x_1,\cdots,x_n):\mathcal{B}^n \rightarrow \mathcal{B}$. We use boldface to indicate vector-valued functions. The i^{th} component of a vector function $\mathbf{F}$ is indicated by F_i.

The *1-cofactor* of a function F w.r.t. a variable x_i is the function F_{x_i} obtained by substituting 1 for x_i in F. Similarly, the *0-cofactor*, $F_{\overline{x_i}}$, is obtained by substituting 0 for x_i in F.

Definition 2.1. *Let $F:\mathcal{B}^n \rightarrow \mathcal{B}$ denote a non-constant Boolean function of n variables $x_1,\cdots,x_n$. We say that F* **depends** *on x_i if $F_{x_i} \neq F_{\overline{x_i}}$. We call* **support** *of F, indicated by $\mathcal{S}(F)$, the set of Boolean variables F depends on. In the most general case when F is a multiple output function, we say that $\mathbf{F}:\mathcal{B}^m \rightarrow \mathcal{B}^n$ depends on a variable x_i, if at least one of its components F_i depends on it.*

The **size** of $\mathcal{S}(F)$ is the number of its elements, and it is indicated by $|\mathcal{S}(F)|$. Two functions F,G are said to have **disjoint support** if they share no support variables, *i.e.* $\mathcal{S}(F)\cap\mathcal{S}(G)=\emptyset$.

Definition 2.2. *The* **range** *of a function $\mathbf{F}:\mathcal{B}^n \rightarrow \mathcal{B}^m$ is the set of m-tuples that can be asserted by $\mathbf{F}$, and it will be denoted by $\mathcal{R}(\mathbf{F})$:*

$$\mathcal{R}(\mathbf{F}) = \{y \in \mathcal{B}^m | \exists x \in \mathcal{B}^n, \mathbf{F}(x) = y\}$$

For scalar functions the range reduces to $\mathcal{R}(F) = \mathcal{B}$ *for all except the two constant functions* 0 *and* 1.

An operation between Boolean functions that will be needed in the following presentation is that of *generalized cofactor*:

Definition 2.3. *Given two functions* F *and* G*, the generalized cofactor of* F *w.r.t.* G *is the function* F_G *such that for each input combination satisfying* G *the outputs of* F *and* F_G *are identical.*

Notice that in general there are multiple possible functions F_G satisfying the definition of generalized cofactor. Moreover, if F and G have disjoint supports, than one possible solution for F_G is the function F itself.

A special class of functions that will be used frequently is that of *characteristic functions*. They are scalar functions that represent sets implicitly: They are asserted if and only if their input value belongs to the set represented.

Definition 2.4. *Given a set* $\mathcal{V} \subset \mathcal{B}^n$*, whose elements are Boolean vectors, its characteristic function* $\chi_V(x) : \mathcal{B}^n \to \mathcal{B}$ *is defined as:*

$$\chi_V(x) = \begin{cases} 1 & \textit{when} \quad x \in \mathcal{V} \\ 0 & \textit{otherwise} \end{cases} \tag{2.1}$$

When sets are represented by their characteristic function, the operations of set intersection, union and complementation correspond to *AND*, *OR* and *NOT* respectively, on their corresponding functions. This observation will be very useful in the following presentation.

2.3.1 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) [17, 19] are a compact and efficient way of representing and manipulating symbolic Boolean functions. Because of this, BDDs are a key component of all symbolic techniques for verification. They form a canonical representation, making the testing of functional properties such as satisfiability and equivalence straightforward.

BDDs are rooted *directed acyclic graphs* that satisfy a few restrictions for canonicity and compactness. Each path from root to leaves, in the graph, correspond to an evaluation of the Boolean function for a specific assignment of its input variables.

Example 2.1. *Figure 2.3.a represents the BDD for the function* $F = (\overline{x} + \overline{y})pq$. *Given any assignment to the four input variables it is possible to find the value of the function by following the corresponding path from the root* F *to a leaf. At each node, the* 0 *edge (dashed) is chosen is the variable has a value* 0*, similarly for the* 1 *edge.*

Figure 2.3.b represents the BDD for the function $G = w \oplus x \oplus y \oplus z$. *Observe that the number of BDD nodes needed to represent XOR functions with BDDs, is* $2 \cdot \#vars$. *At the same time, other canonical representations, such as truth tables or sum of minterms require a number of terms that is exponential with respect to the number of variables in the function's support.*

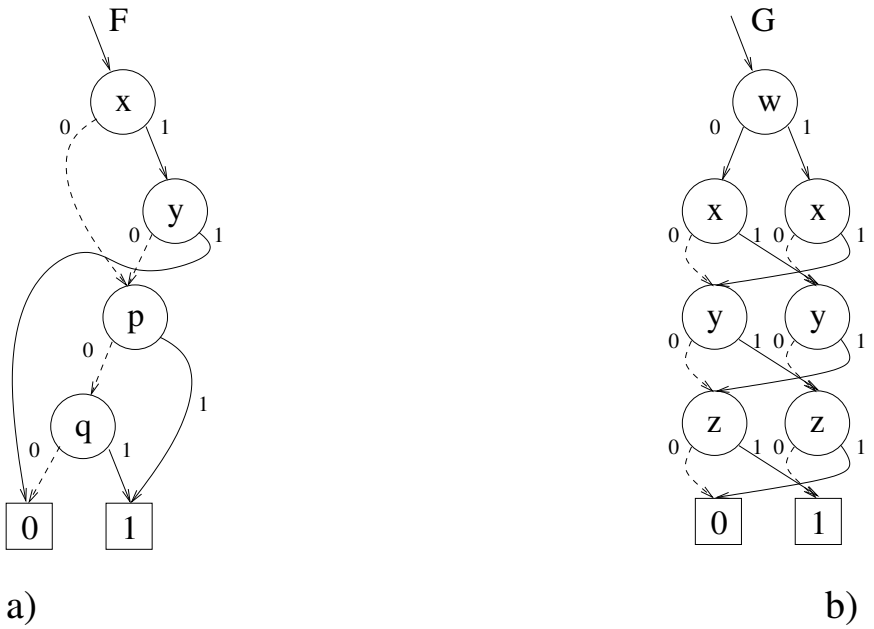

Figure 2.3: Binary Decision Diagrams

For a given ordering of the variables, it was shown in [17] that a function has a unique BDD representation. Therefore, checking the identity of two functions reduces to checking for BDD identity, which is accomplished in constant time.

The following definition formalizes the structure of BDDs:

Definition 2.5. *A BDD is a DAG with two sink nodes labeled "0" and "1" representing the Boolean functions* **0** *and* **1**. *Each non-sink node is labeled with a Boolean variable x_i and has two out-edges labeled 0 and 1. Each non-sink node represents the Boolean function $\overline{x_i}F_0 + x_iF_1$, where F_0 and F_1 are the cofactors w.r.t. x and are represented by the BDDs rooted at the 0 and 1 edges respectively.*

Moreover, a BDD satisfies two additional constraints:

1. *There is a complete (but otherwise arbitrary) ordering of the input variables and every path from source to sink in the BDD visits the input variables according to this ordering.*

2. *Each node represents a distinct logic function, that is, there is no duplicate representation of the same function.*

A common optimization in implementing BDDs is the use of *complement edges* [13]. A complement edge indicates that the connected function is to be interpreted as the complement of the ordinary function. When using complement edges, BDDs have only one sink node "1", whereas the sink node "0" is represented as the complement of "1".

Boolean operations can be easily implemented as graph algorithms on the BDD data structure by simple recursive routines making Boolean function manipulation straightforward when using a BDD representation.

A critical aspect that contributes to the wide acceptance of BDDs for representing Boolean functions is that in most applications the amount of memory required for BDDs remains manageable. The number of nodes that are part of a BDD, also called the BDD size, is proportional to the amount of memory required, and thus the peak BDD size is a commonly used measure to estimate the amount of memory required by a specific computation involving Boolean expressions. However, the variable order chosen can affect the size of a BDD. It has been shown that for some type of functions the size of a BDD can vary from linear to exponential based on the variable order. A lot of research has been done in finding algorithms that can provide a good variable order. While finding the optimal order is an intractable problem, many heuristics have been suggested to find sufficiently

good orders, from static approaches based on the underlying logic network structure in [52, 32], to dynamic techniques that change the variable order whenever the size of the BDD grows beyond a threshold [58, 12].

Binary Decision Diagrams are used extensively in symbolic simulation, one of the more successful formal verification methods. The most critical drawback of this method is its high demand on memory resources, which are mostly used for BDD representation and manipulation. This thesis introduces novel techniques that transform the Boolean functions involved in symbolic simulations through parameterization. The result of the parameterization is to produce new functions that have a more compact BDD representation, while preserving the same results of the original symbolic exploration. The reduced size of the BDDs involved in simulation translates to a lower demand of memory resources, and thus it increases the size of IC designs that can be effectively tackled by this formal verification approach.

2.4 Models for design verification

The verification techniques that we present in this thesis rely on a structural gate-level network description of the digital system, generally obtained from the logic synthesis phase of the design process. In the most general case, such networks are sequential, meaning that they contain storage elements like flipflops or banks of registers. Such circuits store state information about the system, thus the output at any point in time depends not only on the current input but also on historical values of the input. State transition models are a common abstraction to describe the functionality of a design. In this section we review both their graph representation and the corresponding mathematical model.

2.4.1 Structural network model

A digital circuit can be modeled as a network of ideal combinational logic gates and a set of memory elements to store the circuit state. The combinational logic gates we use are: *AND*, *OR*, *NOT* or

XOR. Figure 2.4 reproduces the graphic symbol we use for each of these types.

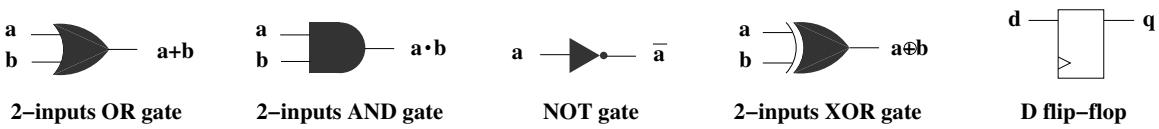

Figure 2.4: Graphic symbols for some basic logic gates

A synchronous sequential network has a set of primary inputs and a set of primary outputs. We make the assumption that the combinational logic elements are ideal, that is that there is no delay in the propagation of the value across the combinational portion of the network. Figure 2.5 represents such a model for a general network.

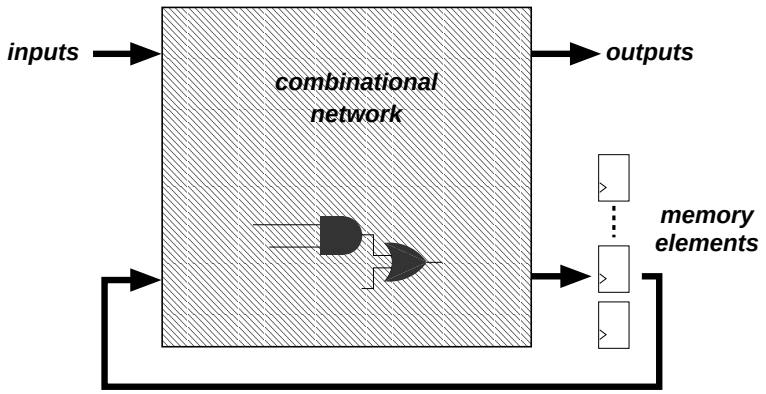

Figure 2.5: Structural network model

We also assume that there is a single clock signal to latch all the memory elements. In the most general case where a design has multiple clocks, the system can still be modeled by an equivalent network with a single global clock and appropriate logic transformations to the inputs of the memory elements.

Example 2.2. *Figure 2.6 is an example of a structural network model for a 3-bits up-down counter with reset. The inputs to the system are the* reset *and the* count *signals. The outputs are 3 bits representing the current value of the counter. The clock input is assumed implicitly. This system*

has four memory elements that store the current counter value and if the counter is counting up or down. At each clock tick the system updates the values of the counter if the count signal is high. The value is incremented until it reaches the maximum value seven, after, it is decremented down to zero. Whenever the reset signal is held high the counter is reset to zero.

The dotted perimeter in the figure indicates the combinational portion of the circuit's schematic.

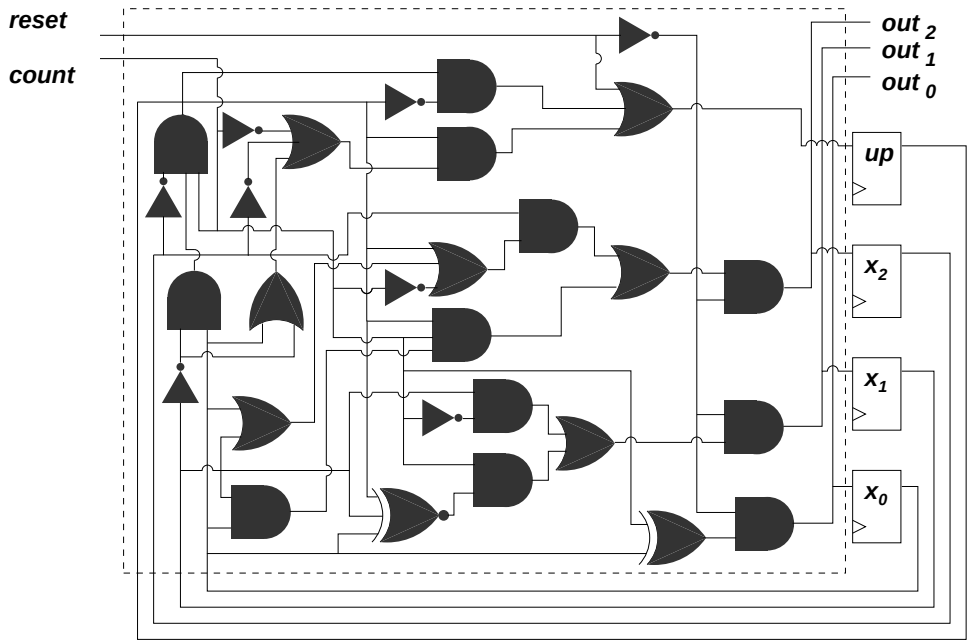

Figure 2.6: Network model of a 3-bits up/down counter with reset

2.4.2 State diagrams

A representation that can be used to describe the functional behavior of a sequential digital system is a Finite State Machine model.

Such model can be represented through state diagrams. A *state diagram* is a labeled directed graph where each node represents a possible configuration of the circuit. The arcs connecting the nodes represent changes from one state to the next and are annotated by the input which would cause such transition in a single clock cycle. State diagrams present only the functionality of the design,

while the details of the implementation are not considered and any implementation satisfying this state diagram will perform the function described. State diagrams also contain the required outputs at each state and/or at each transition. In a Mealy state diagram, the outputs are associated to each transition arc, while in a Moore state diagram outputs are specified with the nodes/states of the diagram. The initial state is marked in a distinct way to indicate the starting configuration of the system.

Example 2.3. *Figure 2.7 represents the Moore state diagram corresponding to the counter of Example 2.2. Each state indicates the value stored in the three flip-flops x_0, x_1, x_2 in bold and in the up/down flip-flop under it. All the arcs are marked with the input signal required to perform that transition. Notice also that the initial state is indicated with a double circle.*

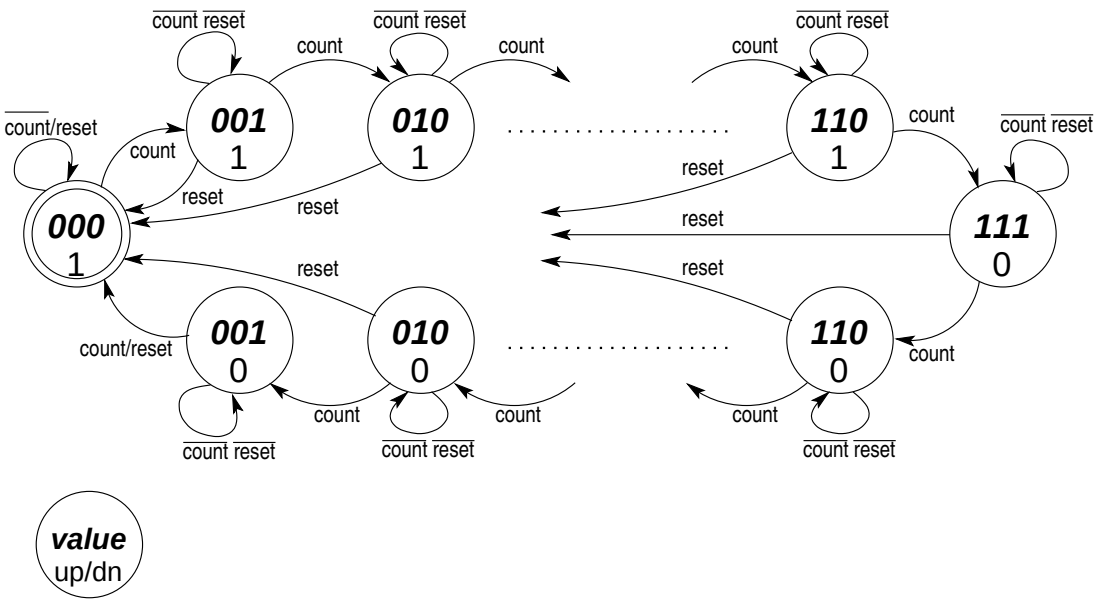

Figure 2.7: State diagram of a 3-bit up/down counter

In the most general case the number of configurations, or different states a system can be in, is much smaller than the number of possible values that its memory elements can assume.

Example 2.4. *Figure 2.8 represents the Finite State Machine of a 3-bits counter 1-hot encoded. Notice that even if the state is encoded using three bits, only the three configurations* $001, 010, 100$

are possible for the circuit. Such configurations are said to be reachable *from the Initial State. The remaining five configuration* $000, 011, 101, 110, 111$ *are said to be* unreachable*, since the circuit will never be in any of these states during normal operation.*

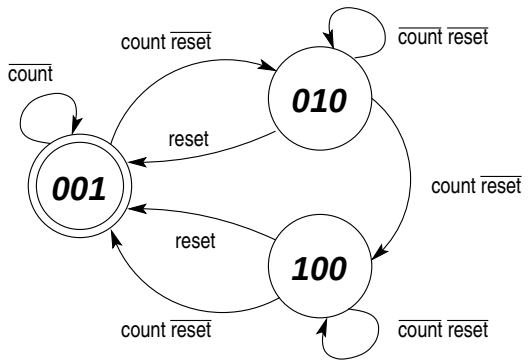

Figure 2.8: State diagram of a 1-hot encoded 3-bit counter

2.4.3 Mathematical model of Finite State Machines

An alternative way of describing a Finite State Machine is through a mathematical description of the set of states and the rules to perform transitions between states. In mathematical terms, a completely specified, deterministic Finite State Machine (FSM) is defined by a 6-tuple:

$$\mathcal{M} = (I, O, S, \delta, S_0, \lambda)$$

where:

- I is an ordered set $(i_1, \cdots, i_m)$ of Boolean input symbols,
- O is an ordered set $(o_1, \ldots, o_p)$ of Boolean output symbols,
- S is an ordered set $(s_1, \ldots, s_n)$ of Boolean state symbols,
- δ is the next-state function: $\delta : S \times I : \mathcal{B}^{n+m} \rightarrow S : \mathcal{B}^n$,
- λ is the output function $\lambda : S \times I : \mathcal{B}^{n+m} \rightarrow O : \mathcal{B}^p$,
- and S_0 is an *initial assignment* of the state symbols.

The definition above is for a Mealy type FSM, for a Moore type FSM the output function λ simplifies to: $\lambda : S : \mathcal{B}^n \rightarrow O : \mathcal{B}^p$.

Example 2.5. *The mathematical description of the FSM of Example 2.4 is the following:*

- $I = \{count, reset\}$,
- $O = \{x_0, x_1, x_2\}$,
- $S = \{001, 010, 100\}$,
- $\delta =$

count reset	*001*	*010*	*100*
0 0	*001*	*010*	*100*
0 1	*001*	*001*	*001*
1 0	*010*	*100*	*001*
1 1	*010*	*001*	*001*

- $\lambda = \{001 \rightarrow 001, 010 \rightarrow 010, 100 \rightarrow 100\}$,
- $S_0 = \{001\}$.

While the state diagram representation is often much more intuitive, the mathematical model gives us a mean of having a formal description of a FSM or, equivalently, of the behavior of a sequential system. The formal mathematical description is also much more compact, making it possible to describe even very complex systems for which a state diagram would be unmanageable.

2.5 Functional validation

The most common approach to functional validation involves the use of a logic simulator software. A commonly deployed architecture is based on the levelized compiled code logic simulator approach by Barzilai and Hansen [4, 33, 66].

Their algorithm starts from a gate level description of a digital system and chooses an order for the gates based on their distance from the primary inputs – in fact, any order compatible with

this partial ordering is valid. The name levelized of the algorithm is due precisely to this initial ordering by levels of the gates. The algorithm then builds an internal representation in assembly language where each gate corresponds to a single assembly instruction. The order of the gates and, equivalently, of the instructions, guarantees that the values for the instruction's inputs are ready when the program counter reaches that specific instruction. This assembly block constitutes the internal representation of the circuit in the simulator.

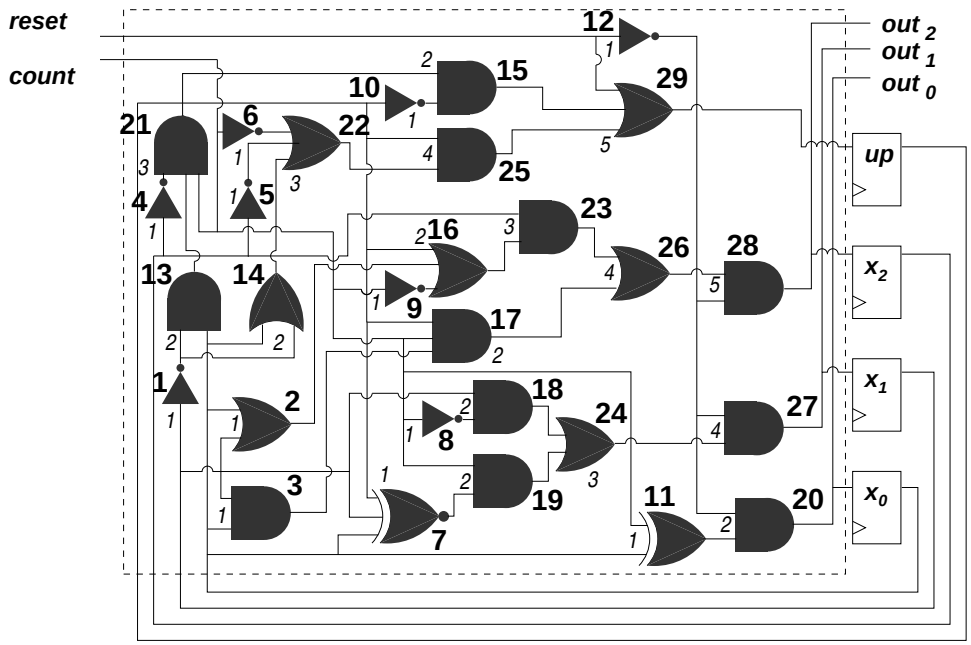

Figure 2.9: Compiled logic simulator

Example 2.6. *Figure 2.9 reproduces the gate level representation of the counter we used in Example 2.2. Each combinational gate has been assigned a level number (in italic in the picture) based on its distance from the inputs of the design. Subsequently, gates have been numbered sequentially (bold numbers) compatibly with this partial order. From this diagram it is possible to write the corresponding assembly block:*

```
1. r1 = NOT(x1)
2. r2 = OR(x0, x1)
```

```
3. r3 = AND(x0, x1)
.. ..
11. r11 = XOR(x0, count)
12. r12 = NOT(reset)
13. r13 = AND(r1, x0)
.. ..
27. r27 = AND(r24, r12)
28. r28 = AND(r26, r12)
.. ..
```

Note that there is a 1-to-1 correspondence between each instruction in the assembly block and each gate in the logic network.

The assembly compiler can then take care of mapping the virtual registers of the source code to the physical registers set available on the specific simulating host.

Multiple inputs gates can be easily handled by composing their functionality through multiple operations. For instance, with reference to Example 2.6, the 3-input *XNOR* of gate **7**, can be translated as:

```
7.     r7tmp = XOR(up, x1)
7bis. r7     = XNOR(r7tmp, x0)
```

At this point, simulation is performed by providing an input test vector, executing the assembly block and reading the output values computed. Such output values can be written to a separate file to be further inspected later to verify the correctness of the results:

Notice that, in first approximation, each of the assembly instructions can be executed in one CPU clock cycle of the host computer, thus providing a very high performance simulation. Moreover, this algorithm scales linearly with the length of the test vector and with the circuit complexity. The high performance and linear scalability of logic simulation are the properties that make this approach to functional validation widely accepted in industry.

```
Logic_Simulator(network_model) {
   assign(present_state_signals, reset_state_pattern);
   while (input_pattern != empty) {
      assign(input_signals, input_pattern);
      CIRCUIT_ASSEMBLY;
      output_values = read(output_signals);
      state_values  = read(next_state_signals);
      write_simulation_output(output_values);
      assign (present_state_signals, state_values);
      next input_pattern;
   }
}
```

Figure 2.10: Logic simulation - pseudocode

The model just described is called a cycle-based simulator, since values are simulated on a cycle by cycle basis. Another family of simulators are event-driven simulators: the key difference is that each gate is simulated only when there is a change of the values at its inputs. This alternative scheduling approach makes possible to achieve a finer time granularity in the simulation, and to simulate events that occur between clock cycles.

Various commercial tools are available that use one or both of the approaches described above, and that have proven to have the robustness and scalability to handle the complexity of designs being developed today. Such commercial tools are also very flexible: Practical cycle-based simulators allow for circuits with multiple clocks and the ability to mix cycle-based and event-based simulation to optimize performance [31]. When deployed in a digital system development context, simulators constitute the core engine of the functional validation process. However, the development of meaningful test sequences is the bulk of the time spent in verification. Generally the test stimuli are organized so that distinct sequences cover different aspects of the design functionalities. Each test sequence needs to be hand crafted by verification engineers. The simulated output values then are checked again by visual inspection. Both these activities require an increasing amount of engineering resources.

As mentioned before, some support in such development is available from specialized programming languages that make it possible for the verification engineer to use powerful primitives to create stimuli for the design, and to create procedures to automatically check the correctness of the output values [34, 49]. These test programs are then compiled and executed side by side with the simulation, exchanging data with it at every time step. Another module that is often run in parallel to the simulator or as a post-processing tool is a coverage engine: it collects analytical data on the portions of the circuit that has been exercised.

Since designs are developed and changed on a daily basis, it is typical to make use of verification farms – thousands of computers running logic simulators – where the test suites are run every day for weeks at a time.

Another common validation methodology approach in industry is pseudo-random simulation. Pseudo-random simulation is mostly used to provide chip level validation and to complement Stand Alone Testing validation at the module level. This approach involves running logic simulation with stimulus generated randomly, but within specific constraints. For instance, a constraint could specify that the reset sequence is only initiated 1% of time. Or it could specify some high level flow of the randomly generated test, while leaving the specific vectors to be randomly determined [2, 26, 69]. The major advantage of pseudo-random simulation is that the burden on the engineering team for test development is greatly reduced. However, since there is a very limited control on the direction of the design state exploration, it is hard to achieve a high coverage with this approach and to avoid producing just many similar redundant tests that have limited incremental usefulness.

Pseudo-random simulation is also often run using emulators, which conceptually are hardware implementations of logic simulators. Usually they use configurable hardware architectures, based on FPGAs (Floating Point Gate Arrays) or specialized reconfigurable components that are configured to reproduce the gate level description of the design to be validated [56, 35, 25]. While emulators can perform one to two order of magnitude faster than software based simulation, they constitute a very expensive solution because of the high raw cost of acquiring them and the time consuming process of configuring them for a specific design, which usually requires several weeks

of engineering effort. Because of these reasons, they are mostly used for IC designs with a large market.

Even if design houses put so much effort in developing tests for their designs and in maximizing the amount of simulation in order to achieve thorough coverage, simulation can only stimulate a small portion of the entire design and thus can potentially miss a subtle design error that might only surface trouble under a particular set of rare conditions.

2.6 Formal verification

On the other side of the verification spectrum are formal verification techniques. These methods have the potential to provide a quantum leap in the coverage achievable on a design, thus improving significantly the quality of verification. Formal verification attempts to establish universal properties about the design, independent of any particular set of inputs. By doing so, the possibility of letting corner situations go untested in a design is removed. A formal verification system uses rigorous, formalized reasoning to prove statements that are valid for all feasible input sequences. Formal verification techniques promise to complement simulation because they can generalize and abstract the behavior of the design.

Almost all verification techniques can be roughly classified in one of two categories: model-based or proof-theoretic. *Model-based techniques* usually rely on a brute-force exploration of the whole solution space using symbolic techniques and finite state machines representations. The main successful results of these methods are based on *symbolic state traversal* algorithms which allow the full exploration of digital systems up to a few hundreds latches. At the root of state traversal approaches is some type of implicit or explicit representation of all the states of a systems that have been visited up to a certain step of the traversal. Since there is an exponential relationship between the number of states and the number of memory elements in a system, it is easy to see how the complexity of these algorithms grows exponentially with the number of memory elements in a system. This problem is called the *state explosion problem* and it's the main reason for the very

limited applicability of the method. At the same time, the approach has the advantage of being fully automatic.

An alternative approach that belongs to the model based category is *symbolic simulation*. This method verifies a set of scalar tests with a single symbolic vector. Symbolic functions are assigned to the inputs and propagated through the circuit to the outputs. This method has the advantage that large input spaces can be covered *in parallel* with a single symbolic sweep of the circuit. Again, the bottleneck of this approach lies in the explosion of symbolic functions representations.

Symbolic approaches are also at the base of equivalency checking, another verification technique. In equivalence checking, the goal is to prove that two different network models provide the same functionality. In recent years this problem has found solutions that are scalable to industrial size circuits, thus achieving full industrial acceptance. Although checking the equivalence of two circuits has the advantage that it does not require to face the state explosion problem, nevertheless, there is hope that symbolic techniques will be the basis for viable industrial-level solutions to formal verification.

The other family of techniques, *proof-theoretic methods*, are based on abstractions and hierarchical methods to prove the correctness of a system [39, 42]. Verification in this framework uses theorem prover software to provide support in reasoning and deriving proofs about the specifications and the developed model of a design. They use a variety of logic representations. The design complexity that a theorem prover can handle is unlimited. However, currently available theorem provers require significant human guidance: even with a state-of-the-art theorem prover, proving that a model satisfies a specification is a very hand-driven process. Thus, this approach is still impractical for most industrial applications.

2.6.1 Symbolic Finite State Machine traversal

One approach used in formal verification is to focus on a property of a circuit and prove that this property holds for any configuration of the circuit that is reachable from its initial state. For instance, such property could specify that if the system is properly initialized, it never deadlocks. Or, in the

case of pipelined microprocessor, one property could be that any issued instruction completes within a finite number of clock cycles. The proof of properties such as these, require, first of all, to construct a global state graph representing the combined behavior of all the components of the system. After this, each state of the graph needs to be inspected to check if the property holds for that state. Many problems in formal hardware verification are based on reachable state computation of Finite State Machines (FSMs). A *reachable state* is just one that is reachable for some input sequence from a given set of possible initial states (see Example 2.4). This type of computation uses a symbolic breadth-first approach to visit all reachable states, also called reachability analysis. This approach, described below, has been published in seminal papers by Madre, Coudert and Berthet [27] and later in [64, 21].

In the context of FSMs, reachable states computations are based on implicit traversal of the state diagram (Section 2.4.2). The key step of the traversal is in computing the *image* of a given set of states in the diagram, that is, computing the set of states that can be reached from the present state with one single transition (following one edge in the diagram).

Example 2.7. *Consider the state diagram of Figure 2.7. The image of the one state set* $\{000-1\}$ *is* $\{000-1, 001-1\}$ *since there is one edge connecting the state* $000-1$ *to both these states. The image of the set* $\{110-1, 111-0\}$ *is* $\{000-1, 110-1, 111-0, 110-0\}$.

Definition 2.6. *Given a FSM* $\mathcal{M}$ *and a set of states R, its image is the set of states that can be reached by one step of the state machine. With reference to the model definition of Section 2.4.3, the image is:*

$$Img(\mathcal{M}, R) = \{s' | s' = \delta(s, i), s \in R, i \in I\}$$

It is also possible to convert the next state function $\delta()$ into a *transition relation* $TR(s, s')$, which holds when there is some input i such that $\delta(s, x) = s'$. This relation is defined by existentially

quantifying the inputs from $\delta()$:

$$TR(s,s') = \exists i \left[\bigwedge_{k=1}^{n} \delta_k(s,i) \equiv s'_k \right]$$

where δ_k represents the transition function for the kth bit. The transition relation can be represented by a corresponding characteristic function – see Definition 2.4 – $\chi_T R$ which equals 1 when $TR(s,s')$ holds.

We can then define the image of a pair $\langle \mathcal{M}, R \rangle$ using characteristic functions. Given a set of states R with characteristic function χ_R, its *image under transition relation TR* is the set *Img* having characteristic function:

$$\chi_{Img}(s') = \exists s \ \ (\chi_{TR}(s,s') \cdot \chi_R(s))$$

As we mentioned before, image computation is the key step of reachability analysis, the operation of finding the set of all the states of the FSM that are reachable. Reachability analysis involves determining the set of states that can be reached by a FSM, after an arbitrary number of transitions, given that it starts from an initial state S_0. The set of reachable states can be computed by a *symbolic breadth-first traversal* where all operations are performed on the characteristic functions. During each iteration, the procedure starts from a set of newly encountered states `from` and performs an image computation on the set to determine the new set of states `to` that can be reached in one transition from the states in `from`. The states to include in the `from` set are simply the states in the `to` set of the previous step that have not already been used in a previous image computation. Since the FSM that is being traversed has a finite number of states and transitions, these iterations will eventually reach a point where no new states are encountered. That point is called the *fixpoint*. The final accumulated set of states `reached` represents the set of all reachable states from the initial state S_0. In practice, all the sets involved in the computation are represented by their characteristic functions.

```
Machine_Traversal( FSM M ) {
   from = new = reached = initial_state;
   while (new ≠ ∅) {
      to = Img(transition_relation, from);
      new = To \ reached;
      reached = reached ∪ new;
      from = new;
   }
   return reached;
}
```

Figure 2.11: FSM state traversal - pseudocode

In general, the number of iterations required to achieve this fixpoint could be linear in the number of states of the FSM, and thus exponential in the number of memory elements of the system.

Symbolic traversal is at the core of the *symbolic model checking* approach to verification. The basic idea underlying this method is to use BDDs (Section 2.3.1) to represent all the functions involved in the validation and the set of states that have been visited during the exploration. The primary limitation of this approach is that the BDDs that need to be constructed can grow extremely large, exhausting the memory resources of the simulation host machine and/or causing severe performance degradation. Moreover, each image computation operation can take too long. The solution (exact or approximate) to these bottlenecks is still the subject of intense current research, in particular various solutions have been proposed that try to contain the size of the BDDs involved [57, 24] and to reduce the complexity of performing the image computation operation [23, 54].

Finally, another limitation of symbolic traversal is that it is not very informative from a design debugging standpoint: If a bug is found, it is nontrivial to construct an input trace that exposes it.

2.7 Symbolic Simulation

An alternative approach to symbolic state traversal is symbolic simulation. As described in Section 2.5, a logic simulator uses a gate-level representation of a circuit and perform the simulation by manipulating the Boolean scalar values, 0 and 1. Symbolic simulation differ from logic simulation

because it builds Boolean expressions rather than scalar values, as a result of circuit simulation. Consider the two *OR* gates in Figure 2.12.

Figure 2.12: Logic and symbolic simulation

On the left side, in performing logic simulation, the two input values 0 and 1 are evaluated and the result of the simulation produces a value 1 for the output node of the gate. On the right side of the figure, we perform a symbolic simulation of the same gate. The inputs are the two symbolic variable a and b, and the result placed at the output node is the Boolean expression $a+b$.

This approach is very powerful is two ways. First, at the completion of the symbolic simulation we have a Boolean expression that represents the full functionality of the circuit (in this example, a tiny one-gate circuit). This expression can be compared and verified against a formal specification of the desired outputs. Because of the quickly increasing complexity of these expressions, this comparison is only feasible for very small designs. Second, symbolic simulation can be seen as a way of applying multiple test vectors in parallel to a logic simulator, each symbolic variable representing implicitly both scalar values 0 and 1 and thus multiplying by a factor of two the number of equivalent vectors that are being simulated. For instance, the symbolic simulation of Figure 2.12 is implicitly applying four test vectors in parallel corresponding to $\{a=0,b=0\}$, $\{a=1,b=0\}$, $\{a=0,b=1\}$, $\{a=1,b=1\}$. This use of symbolic simulation is interesting also because it can be easily integrated in a logic simulation methodology where the amount of parallelism in the test vector can be tuned to the resources of the host.

2.7.1 The algorithm

The key idea of symbolic simulation consists of the use of mathematical techniques for letting symbols represent arbitrary input values for a circuit. One of the first works in this area is by King in 1976 [48], where he proposes a method of symbolic execution to verify software programs. More recently, in 1987, Bryant introduced in [20] a method for the symbolic simulation of CMOS designs. The algorithm presented in that work uses BDDs – see the previous Section 2.3.1 – as the underlying mathematical technique to represent the values associated with the nodes of the circuit.

In symbolic simulation, the state space of a synchronous circuit is explored iteratively by means symbolic expressions. The iterative exploration is performed with reference to a gate level description of the digital design. At each step of simulation each input signal and present state signal is assigned a Boolean expression; these expressions can be generally complex or extremely simple, such as simple Boolean variables or even constant values. The simulation proceeds by deriving the appropriate Boolean expression for each internal signal of the combinational portion of the network, based on the expressions at the inputs of each logic gate and the functionality of the gate. It is straightforward to see an analogy with the logic simulation approach described in Section 2.5, where we would operate on the same gate level description model for the design, but the inputs would be assigned to constant values instead of Boolean expressions and the internal operations would be in the binary domain.

In detail, the algorithm operates as follows: At time step 0, the gate level network model is initialized with the initial assignment S_0 for the each of the state signals and with a set of Boolean variables $IN_{@0} = \{i_{1@0}, \cdots, i_{m@0}\}$ for the combinational input signals. symbols. During each time step, the Boolean expressions corresponding to the primary outputs and the next state signals are computed in terms of the expressions at the inputs of the network. To do this, a Boolean expression is computed at each gate's output node based on the gate's functionality. Gates are evaluated in an order compatible with their distance from the input nodes, similarly to what is done in compiled level logic simulation. At the end of each step, Boolean expression are obtained for the primary outputs. The expressions computed for the memory elements' inputs are fed back to the state inputs

of the circuit for the next step of simulation.

```
Symbolic_Simulator(network_model) {
   assign(present_state_signals, reset_state_pattern);
   for (step = 0; step < MAX_SIMULATION_STEPS; step+1 ) {
      input_symbols = create_boolean_variables (m, step);
      assign(input_signals, input_symbols);
      foreach (gate) in (combinational_netlist) {
         compute_boolean_expression (gate);
      }
      output_symbols = read(output_signals);
      state_symbols  = read(next_state_signals);
      check_simulation_output(output_symbols);
      assign (present_state_signals, state_symbols);
   }
}
```

Figure 2.13: Symbolic simulation algorithm - pseudocode

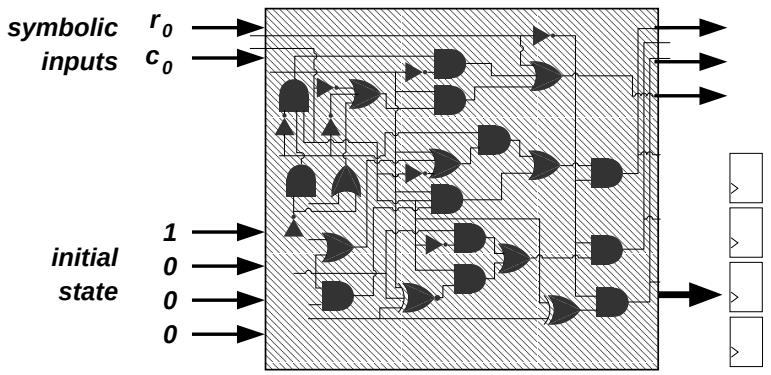

Figure 2.14: Symbolic simulation for Example 2.8 - Initialization phase

Example 2.8. *We want to symbolically simulate the counter circuit of Example 2.2. To set up the simulation we configure the state lines with the initial state* $\{000-1\}$ *as indicated in Figure 2.7. Then, we use two symbolic variables* r_0 *and* c_0 *for the two input lines.*

At this point the simulation proceeds computing a symbolic expression for each gate output node. Using the labels of Figure 2.9, we show some of the expressions for the first step of simulation:

1. 1 2. 0 ...

11.	c_0	12.	$\overline{r_0}$	13.	0
...		19.	0	20.	$\overline{r_0}c_0$

At the end of the first step, the expressions for outputs and flip-flop's inputs are:

$$\begin{aligned} out_0 &= x_0 = \overline{r_0}c_0 \\ out_1 &= x_1 = out_2 = x_2 = 0 \\ up &= 1 \end{aligned}$$

The expressions computed for the memory elements are used to set the state lines for the next simulation step, while the input lines will be set with new symbolic variables as indicated in Figure 2.15. At completion of the second simulation step, we obtain the following expressions:

$$\begin{aligned} out_0 &= x_0 = ((\overline{r_0}c_0) \oplus c_1)\overline{r_1} \\ out_1 &= x_1 = \overline{r_0 r_1}c_0 c_1 \\ out_2 &= x_2 = 0 \\ up &= 1 \end{aligned}$$

Note how the expressions involved in the simulation become increasingly complex at each time step.

Notice that new Boolean variables are created at every simulation step, one for each of the combinational primary inputs of the network. Thus, the expression obtained for the outputs and the state signals at the end of each step k will be functions of variables in $\{IN_{@0}, \cdots, IN_{@k}\}$. The vector of Boolean functions obtained for the state symbols $\mathbf{ST}_{@\mathbf{k}} : \mathcal{B}^{mk} \rightarrow \mathcal{B}^n$ represents all the states that can be visited by the circuit at step k. The state symbols $\mathbf{ST}_{@\mathbf{k}}$ represent the states at step k in implicit form, that is, any distinct assignment to the symbolic variables $\{IN_{@0}, \cdots, IN_{@k}\}$ will evaluate the state expressions to a valid state vector that is reachable in k steps from the initial state S_0. Viceversa, each state that is k steps away from the initial state corresponds to a least one

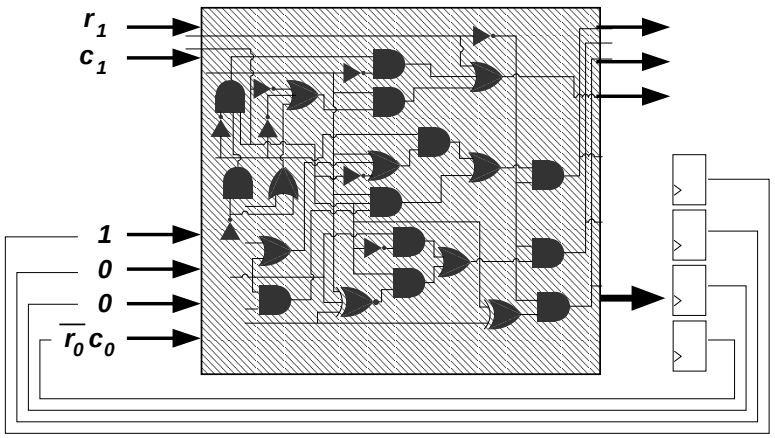

Figure 2.15: Symbolic simulation for Example 2.8 - Simulation Step 2

evaluation of the symbolic variables.

The procedure just described is equivalent to propagating the symbolic expressions through a time-unrolled version of the circuit, where the combinational portion is duplicated as many times as there are simulation steps. Figure 2.16 shows this iterative model and input and output nodes for the symbolic variables and expressions, respectively.

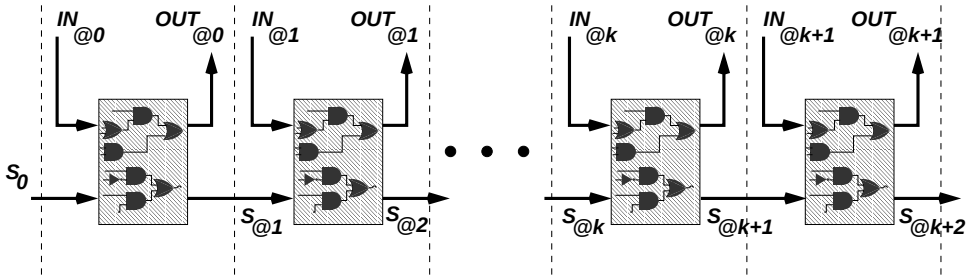

Figure 2.16: Iterative model of symbolic simulation

Design errors are found by checking at every step that the functions $\mathbf{OUT}_{@\mathbf{k}} : \{in_{@0}, \cdots, in_{@k}\} \rightarrow \mathcal{B}^p$ represents a set of legal values for the outputs of the circuit. When an illegal output combination is found, $\mathbf{OUT}_{@\mathbf{k}}$ reports all the possible input combinations that expose it. To generate a test sequence that exposes the error, is sufficient to find an evaluation of all the symbolic variables that simultaneously satisfy the error condition. If the correct output vector at time k is given by the

Boolean vector $C \in \mathcal{B}^p$, all the assignments that satisfy the expression Err:

$$Err = \bigwedge_{i=1}^{p} (\mathbf{OUT}_{@\mathbf{k},i} = C_i)$$

are valid test sequences that expose the design error. This approach can be easily generalized to verify more complex properties where the output signals have to satisfy complex relations expressed in terms of the input symbols.

2.7.2 The challenge in symbolic simulation

While theoretically the simulation can proceed indefinitely, the representation of the Boolean expressions involved eventually requires memory resources beyond those available in the simulation host. Similarly to the state traversal algorithm, the bottleneck of this approach lies in the explosion of the BDD representations. Various techniques have been suggested to approximate the functions represented in order to contain the size of the BDDs within reasonable limits.

In [67], Wilson presents a technique that imposes a hard limit on the size of the BDDs involved in the simulation. Whenever this limit is reached, one or more of the symbolic variables are evaluated to a constant, so that the Boolean expressions, and consequently, their BDD representations, can be simplified. The simulation step is then re-simulated with the other constant value for each of the simplified symbolic variable, until complete expressions are obtained for the outputs of the network.

A different approach is chosen by Bergmann in [6], where the design is abstracted, so that its size can be within reach of a simulator. Different abstractions are chosen based on coverage holes – areas of the design that had not been explored yet – with the objective that each new abstraction will improve the current coverage.

In [36], Bertacco *et al.* attempt to overcome the limitations of the single techniques presented in this chapter by using collaborative engines: symbolic simulation interacts with logic simulation in achieving the most coverage of the design within boundaries of time and memory usage, while

symbolic state traversal is used with abstraction techniques to prove some portions of the design's state space as non reachable, and thus prove that could they cannot be covered by the simulation engines. The result is an integrated software tool that supports the designer in "classifying" the state space of the IC design into reachable and unreachable and produces efficient and compact tests to visit the reachable portion of a design.

Even if some of these efforts provide a major contribution in making the symbolic simulation approach much more attractive for use in industrial settings, there is still a lot to be conquered and the functional verification of digital systems remains a challenge for every hardware engineering team. The core observation underlying the work in this thesis is that symbolic simulation traverses the states of a digital system carrying across each simulation step much more information than it is needed to verify the system. In fact, it uses complex Boolean expressions to describe each of the states that can be visited during each simulation step and how they relate to the input symbolic variables. However, we need much less information in order to achieve our objective to be able to identify which states can be visited at each time step. In fact, for this latter goal, any encoding of the set of states reached is sufficient. Thus, the remainder of this thesis develops techniques and theoretical work to discover new efficient encodings of the state sets involved in simulation. These new encodings, or parameterization, have more compact representations than the original ones, and thus allow for a memory efficient symbolic simulation, that presents much better robustness and scalability characteristics.

Chapter 3

Cycle-Based Symbolic Simulation

This chapter introduces our first technique to address the robustness and scalability limitations of the traditional symbolic simulation approach. We present an algorithm that can be applied to much more complex designs using a bounded amount of memory resources. The main focus of this algorithm is to achieve as much breadth of traversal as possible while maintaining the advantages of logic simulation, namely scalability and limited memory requirements. In the best situation, this approach achieves the same breadth of traversal as a pure symbolic simulation algorithm, but the breadth of the traversal can be reduced if that is required to minimize memory usage. Thus, Cycle-Based Symbolic Simulation, or CBSS, can be viewed as a hybrid approach that exploits the tradeoffs between symbolic search and logic simulation. Before diving into the presentation of this new algorithm, we discuss the motivation for this direction of work, namely the use of parameterization in symbolic simulation and its advantages for a reduced memory profile.

3.1 Parametric transformations

The central observation underlying the work of this thesis is that the expressions involved in a symbolic simulation exploration carry more information than the algorithm uses. At the end of each step, the Boolean expressions representing the state signals are fed back to the sequential inputs of

the gate level network and used for the next simulation step. As we pointed out in Section 2.7.1, at the end of a generic step k, these expressions represent implicitly all the states that are reachable by the design in k steps from an initial state S_0. We observe now, that this implicit description of the set of states $\mathbf{S_{@k}}$ is most often redundant. Since the only information that needs to be transfered across simulation steps is the set of states that have been reached in the previous step, it is generally possible to define a more compact encoding of this set description, that is, a new *parameterization* of the state set. We can then use this new implicit description for the next simulation step. Figure 3.1 shows where the transformation takes place in the simulation flow.

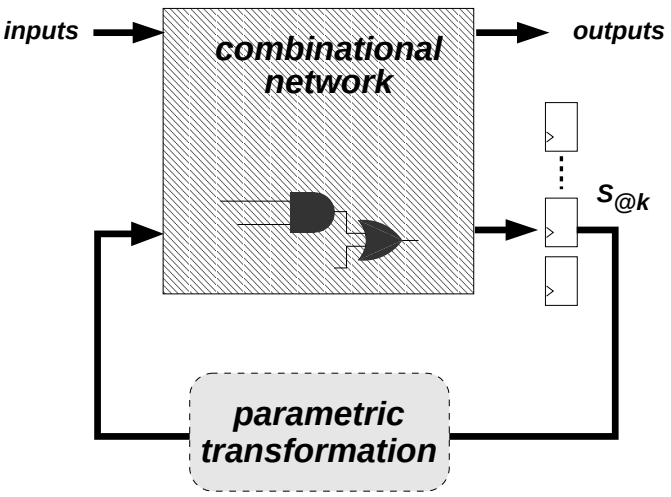

Figure 3.1: Parameterization of the state vector during symbolic simulation

Consequently, if we can define a new encoding that uses compact BDDs and transforms the expressions defining the set of reached states at the end of every simulation step based on this new parameterization, then we can maintain a low memory profile across the process and thus achieve better scalability and robustness in simulation.

Example 3.1. *Consider once again the counter of Example 2.2. When we perform the first step of symbolic simulation on this design, with reference to Figure 3.2 - step 1, we obtain the following*

vector of Boolean expressions for the next state functions:

$$\begin{aligned} up &= 1 \\ x_2 &= 0 \\ x_1 &= 0 \\ x_0 &= \overline{r_0} \cdot c_0 \end{aligned}$$

By varying the values associated with each of the variables in the expressions, that is, performing all the assignments $\{00, 01, 10, 11\}$ *for the pair of variables* r_0 *and* c_0*, we obtain an explicit list of all the states that can be reached in one step of symbolic simulation. For this example, such state set is* $\{1000, 1001\}$*. It's easy to see that this set can be more simply encoded as* $\{100p_0\}$*, where* p_0 *is a new Boolean parameter. Note that the new parameterization uses only one Boolean variable instead of two.*

We can now use this new simpler representation of the state set for the second step of simulation and obtain the expressions reported in Figure 3.2 - step 2 after simulating the combinational portion of the network. The new state expressions depend now on three variables: r_1*,* c_1 *and* p_0*, the parameter. Again, by evaluating the expressions for each possible assignment to the Boolean variables, we only obtain three distinct states:* $\{1000, 1001, 1010\}$*. These three states can be more efficiently encoded using only two parameters as:*

$$\begin{aligned} up &= 1 \\ x_2 &= 0 \\ x_1 &= p_0 \\ x_0 &= \overline{p_0} \cdot p_1 \end{aligned}$$

As even this small example shows, most often symbolic simulation produces expressions that are not an efficient encoding of the state set spanned by the traversal. In order to exploit the compact

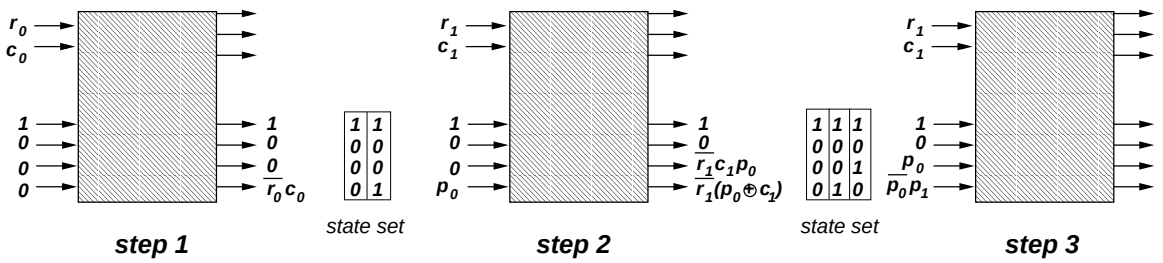

Figure 3.2: Three steps of symbolic simulation for the counter of Example 2.2 and possible parameterizations of the reached state sets

memory representations allowed by parameterization, we need to find an efficient algorithm that can discover good parameterizations automatically.

3.2 Parameterizations in symbolic simulation

Cycle-Based Symbolic Simulation is a hybrid approach in the sense that the values that are propagated through the network can be both symbolic expressions or constant Boolean values. Section 2.3.1 showed that BDDs can be used to represent both efficiently.

Our algorithm adds a *parameterization phase* at the end of each simulation step to basic symbolic simulation, as indicated in Figure 3.1. This parameterization transforms the state vector BDDs into compact BDDs that use only a small amount of memory resources. It is possible that the set of parametric BDDs produced spans only a subset of the original state set. This under-approximation may occur as a trade-off between accuracy of the traversal (that is, producing an exact parameterization) and complexity of the expressions produced (which we want to keep at a minimum). However, even when we settle for representing a subset of the state set, this set is chosen to maximize the amount of states represented for the amount of memory used.

Previous work has used parameterization techniques in connection with FSM traversal or symbolic simulation to reduce the memory requirements of the algorithms. Often, user interaction is required to suggest a relation among different signals of the systems under verification which can

be exploited for parameterizing the simulation. For instance, in [38], the authors exploit the dependencies among state variables to simplify the traversal of a FSM. such dependencies need to be suggested by the designer and are verified for correctness during the simulation of the system. [41] presents a range of techniques to parameterize relations provided by the user. The work in [65] automatically discovers dependencies among state variables during FSM traversal. Detecting such dependencies requires checking all the state variables at each step of the traversal and transforming both the reached set and the transition relation accordingly during every step of the traversal.

Another research direction related to symbolic simulation and parameterizations focuses on partitioning the search exploration based on circuit related constraints and then performing multiple simulation for each element of the partition. In this context, parameterization techniques have been used to express the conditions of each subcase of the partitioned constraints. In particular, Jain *et al.* considered in [40] a variety of Boolean formula representations for the constraints and proposed a method to obtain parametric solutions. Aagaard *et al.* [1] introduced an alternative method where the case splitting on the constraints is based on Shannon decomposition.

In contrast, the focus of the algorithm presented here is to be fully automatic in a symbolic simulation context and to introduce a parameterization that is efficiently computed and produces very compact results in terms of memory requirements. We compare our approach to logic simulation and show that while each simulation step is more time consuming, since it manipulates Boolean expressions instead of constant values, it also produces the equivalent of multiple logic simulations test vector in a single pass. Experimental results report a quantitative analysis of the performance of this new approach to logic simulation.

3.3 The CBSS algorithm

Cycle-Based Symbolic Simulation is initialized by setting the state of the circuit to the initial constant vector S_0 (see Section 2.4.3 for a definition of S_0). Each of the combinational input signals is

assigned a distinct symbolic variable $IN_{@0} = \{i_{1@0}, \cdots, i_{m@0}\}$. The simulation proceeds by computing the Boolean expressions corresponding to each node in the combinational portion of the network, as in the basic symbolic simulation algorithm. At the end of a simulation step, the expressions representing the next-state functions undergo a parametric transformation. During this parameterization, a minimal number of inputs could be set to constants. The objective of the selection is to maximize the breadth of the traversal, while keeping the representation of the state set compact through use of Boolean expressions with a small BDD.

During the simulation, we do not compute a `reached` set as in symbolic state traversal (Section 2.6.1). This computation is one of the main causes of reduced scalability of symbolic state traversal. Its main advantage is to maintain a history of states previously visited in the traversal, which is central to 1) discover when all the reachable states have been visited and the traversal is complete and to 2) select a set of states to use in the next simulation step, possibly with a compact representation. However, the simulation approach we present here targets circuits whose size is beyond the capability of symbolic state traversal. In general we don't expect to complete the simulation within a few hundreds steps, as it is generally the case for the type of designs that symbolic state traversal approaches. Moreover, we use a novel parameterization algorithm that does not require `reached` set information.

After parameterization, the newly generated functions are used as present state for the next state of simulation. Figure 3.3 shows how the algorithm just described corresponds to the iterative model for symbolic simulation. Notice that now we have added two new blocks to those in Figure 2.16. The outputs of the *parametric transformation (PAR-TRF)* block are: 1) the parametrized state vector that is fed to the present state in the next step of simulation and 2) a set of parametric equations that relate the newly created parameters $p_{@k}$ to the set of combinational inputs $\{IN_{@0}, \cdots, IN_{@k}\}$.

The parametric representation of frontier sets that we adopted can be constructed and manipulated very efficiently. The selection of which inputs to tie and to what value is based on the ease of construction of this representation. Alternatively, the value selection can be left to the user or to the tool: by evaluating to constant symbolic variables selectively, it is possible to symbolically simulate

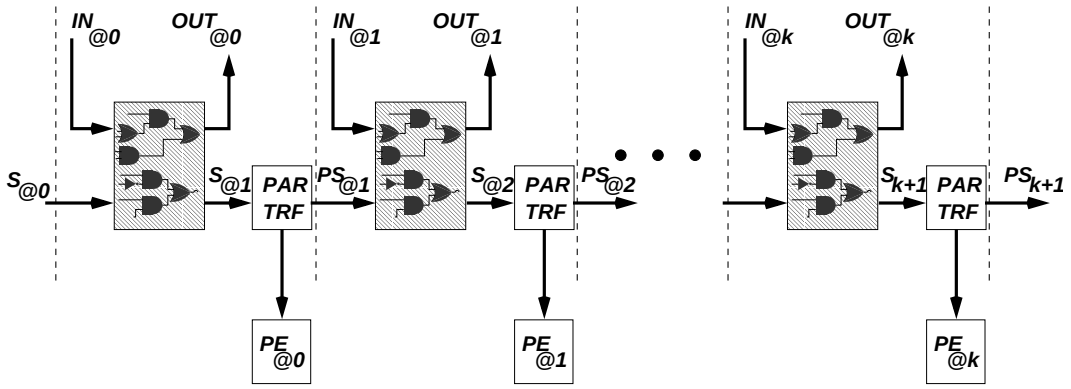

Figure 3.3: Cycle-Based symbolic simulation flow

any neighborhood of an input trace generated by the test bench.

3.4 The parameterization phase

The parameterization technique is based the following observation. In symbolic FSM traversal, the next state function δ can be, in general, complex. The next state functions of symbolic simulation at time step 0, $\mathbf{S_1}$, can be derived from δ as:

$$\mathbf{S_1}(i_{@0}) : I : \mathcal{B}^m \rightarrow S : \mathcal{B}^n = \{\delta(s,i) | s \in S_0, I \equiv IN_{@0}\}, \tag{3.1}$$

that is, by evaluating the state variables to the initial state values and substituting the combinational input variables with the input variables of time step 0. Often, because the state variables are evaluated to constant, the resulting components of $\mathbf{S_1}$ are very simple, such as constants, copies of an input, or complement of an input. Moreover, an input variable may be copied into several components of $\mathbf{S_1}$: there are then functional dependencies among the various state bits. We use these functional dependencies to obtain a simplified representation $\mathbf{PS_1}$ of $\mathbf{S_1}$. At each time step k, we produce a simple, parameterized representation of the next state functions to use for the next step $k+1$. By always presenting a simple set of functions at the present state signals of the network we are able to generate next state functions $\mathbf{S_{@k}}$ that are always simpler and more compact than the

```
CBSS(network_model) {
   assign(present_state_signals, reset_state_pattern);
   for (step = 0; step < MAX_SIMULATION_STEPS; step+1 ) {
      input_symbols = create_boolean_variables (m, step);
      assign(input_signals, input_symbols);
      foreach (gate) in (combinational_netlist) {
         compute_boolean_expression (gate);
      }
      output_symbols = read(output_signals);
      state_symbols  = read(next_state_signals);
      check_simulation_output(output_symbols);
      /* the next line also writes out the parametric equations */
      parametric_state_set = parameterize(state_symbols, step);
      assign (present_state_signals, par_state_set);
   }
}
```

Figure 3.4: The CBSS algorithm - pseudocode

ones involved in the pure symbolic simulation algorithm.

In practice, we never explicitly build the next function δ. Rather, at each clock tick k, we use the functional dependencies among the components of the next state function $\mathbf{S_{@k}}$ at time k to build a parameterized version, $\mathbf{PS_{@k}}$, for time $k+1$. If, in spite of our efforts, $\mathbf{PS_{@k}}$ becomes too complex to be represented with BDDs within our memory budget, a few symbolic variables are tied to constant values to simplify it.

Notice that the parametric representation allows us to avoid the computation and representation of the global next state functions of the circuit as in symbolic state traversal, thereby avoiding a lengthy simulation set-up time.

3.4.1 Using functional dependencies

We discover and exploit functional dependencies using a parametric representation of the next state set. Figure 3.3 illustrates the approach. We introduce some *intermediate* variables p_i. At a generic

clock tick k, we inspect the BDDs of $\mathbf{S_{@k}}$ and build a function $\mathbf{PS_{@k}}$ such that (see Definition 2.2):

$$\mathcal{R}(\mathbf{PS_{@k}}) = \mathcal{R}(\mathbf{S_{@k}}). \tag{3.2}$$

In practice, we will settle for a $\mathbf{PS_{@k}}$ such that 1) the number of parameter variables p is small, and 2) $\mathcal{R}(\mathbf{PS_{@k}})$ is a "large" and easily identifiable subset of $\mathcal{R}(\mathbf{S_{@k}})$:

$$\mathcal{R}(\mathbf{PS_{@k}}) \subseteq \mathcal{R}(\mathbf{S_{@k}}). \tag{3.3}$$

The set $\mathbf{PS_{@k}}$ that we generate has cardinality that is 2^p, where p is the number of parameters we introduce during the parameterization phase. The diagram in Figure 3.5 shows the relation between the whole state space of the system, $\mathbf{S_{@k}}$ and $\mathbf{PS_{@k}}$.

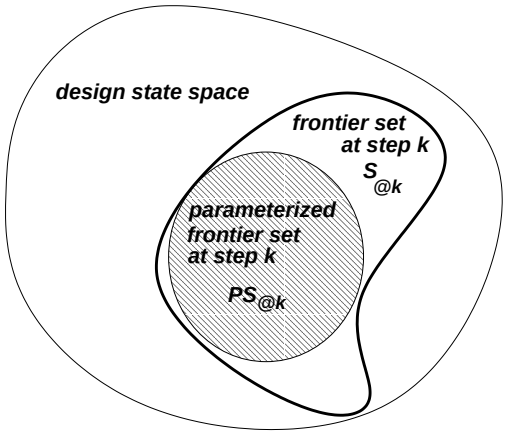

Figure 3.5: The parameterized frontier subset $\mathbf{PS_{@k}}$

Section 3.4.2 provides the details on $\mathbf{PS_{@k}}$ and its construction. The BDD of the next state functions for step $k+1$ is then built by simulation of the combinational portion of the circuit. In terms of the δ function, this corresponds to:

$$\mathbf{S_{k+1}}(i_{@k+1}) : I : \mathcal{B}^m \rightarrow S : \mathcal{B}^n = \{\delta(s,i) | s \in \mathbf{PS_{@k}}, I \equiv IN_{@k+1}\} \tag{3.4}$$

and a new $\mathbf{PS_{k+1}}$ constructed by parameterization. Notice that the state variables are effectively

replaced by the parametric variables p_i.

In addition, we build a second mapping $\mathbf{PE_{@k}}$. This second mapping expresses each p_i as a function of inputs and intermediates at the previous tick. $\mathbf{PE_{@k}}$ should also be "simple", for the following reason. Suppose an error is discovered at time k. There is then an assignment of primary inputs and intermediates at time k that exposes the bug. We need to be able to map the assignment of intermediates to an assignment of inputs and intermediates at time $k-1$, and then iteratively back to primary inputs at time $k-2, \cdots, 0$.

The parametric transformation develops in two phases: the first phase identifies *simple* variables, while the second phase parameterizes *unbound* functions. The pseudocode of the function is shown in Figure 3.6. It guarantees that $\mathcal{R}(\mathbf{S_k})$ can be parameterized in linear time. If this is not the case, it identifies variables for assignment, and cofactors $\mathbf{S_k}$ accordingly. The actual constant values used for the assignment could correspond to the values provided in a testbench for the design, if this is available. For instance, if at the third step of CBSS simulation we need to evaluate to a constant the variable corresponding to input x, we could extract the value assigned at input x in the testbench at the third step of logic simulation. By choosing values based on this criteria, we guarantee that our CBSS algorithm produces a design exploration that includes the search corresponding to logic simulation run on the same testbench. For instance, if there is a testbench that drives the design to a specific corner case to check it, CBSS can not only check that specific configuration of the system, but also cover a set of additional configurations that are "close" to the target one in the FSM model.

Whenever a testbench is not available, we can still automatically produce a random value for the variable assignment. This choice will drive the design through a random walk of the state space.

The pseudocode of the parameterization phase is shown in Figure 3.6. The details of functions `find_simple_complex_var`, `find_shared_eqclasses`, and `remap` are described in the following sections. Function `assign_&_cofactor` simply takes a vector of expressions and a set of variables, assigns a value to each of the variables in the set and partially evaluates each expression based on these values.

```
parameterize(state_equations, step) {
   <simple, complex> = find_simple_complex_var(state_equations);
   state_equations = assign_&_cofactor(state_equations, complex);
   state_equations = remap(state_equations, simple);
   append_param_equations(simple, step);
   <classes, shared> = find_shared_eqclasses(state_equations);
   state_equations = assign_&_cofactor(state_equations, shared);
   state_equations = remap(state_equations, classes);
   append_param_equations(classes, step);
   return state_equations;
}
```

Figure 3.6: `parameterize` function - pseudocode

3.4.2 How to classify the components of the state vector

We show how to quickly identify a function $\mathbf{PS}_{@\mathbf{k}}$ such that $\mathcal{R}(\mathbf{PS}_{@\mathbf{k}})$ is a "large" subset of $\mathcal{R}(\mathbf{S}_{@\mathbf{k}})$. The set of transformations presented in the next two sections can be applied to any Boolean vector function. For purposes of readability, in the following definitions we will refer to the generic function $\mathbf{V} : \mathcal{B}^n \rightarrow \mathcal{B}^m$. As explained above, the CBSS algorithm applies such transformation to the next state vector $\mathbf{S}_{@\mathbf{k}}$.

Definition 3.1. *A variable x is termed* **simple** *if there is a component $\mathbf{V}_i$ of $\mathbf{V}$ such that $\mathcal{S}(\mathbf{V}_i) = \{x\}$. Given a function $\mathbf{V}$, let Si denote the set of simple variables. A component $\mathbf{V}_i$ is termed* **simple** *if $\mathcal{S}(\mathbf{V}_i) \subseteq Si$.*

Definition 3.2. *Let again Si denote the set of simple variables. A non-constant component function $\mathbf{V}_i$ is termed* **complex** *if:*

1. $\mathcal{S}(\mathbf{V}_i) \cap Si \neq \emptyset$ *and*
2. $\mathcal{S}(\mathbf{V}_i) \cap \overline{Si} \neq \emptyset$.

For a complex function $\mathbf{V}_i$, a variable belonging to $\mathcal{S}(\mathbf{V}_i) \cap \overline{Si}$ is also termed **complex**.

Definition 3.3. *A function is* **unbound** *if it is neither simple nor complex. Two components $\mathbf{V}_i$ and $\mathbf{V}_j$ of $\mathbf{V}$ are termed* **equivalent** *if they are unbound and either $\mathbf{V}_i = \mathbf{V}_j$ or $\mathbf{V}_i = \overline{\mathbf{V}_j}$ holds.*

Definition 3.4. *Given an equivalence class* ε *of functions with reference to the previous definition, we indicate with* $\mathcal{S}(\varepsilon)$ *the set of variables belonging to the support of any function in* ε*. A variable* $x \in \mathcal{S}(\mathbf{V})$ *is said to be* **bound** *if it belongs only to the support of a single equivalence class of* $\mathbf{V}$*. It is termed* **shared** *if it belongs to more than one class.*

Example 3.2. *Consider the following function* $\mathbf{S}_{@\mathbf{k}}$*:*

$$\mathbf{S}_{@\mathbf{k}}(x,y) : \mathcal{B}^2 \rightarrow \mathcal{B}^7 = (x, \overline{x}, y, 0, f(x,y), g(x,y), \overline{y})$$

Its components are only: 1) constants, 2) functions of a single variable, or 3) functions of variables also appearing as single variables in other components (that is, simple functions).

In this situation, an exact parametric description is obtained by replacing x and y with two parameters:

$$\mathbf{PS}_{@\mathbf{k}} = (p_0, \overline{p_0}, p_1, 0, f(p_0,p_1), g(p_0,p_1), \overline{p_1})$$

Notice that $\mathbf{PE}_{@\mathbf{k}}$ *is just a data-transfer:* $p_0 = x$*,* $p_1 = y$*.*

Suppose now that $\mathbf{PS}_{@\mathbf{k}}$ consists only of simple and complex functions. By assigning a value to complex variables, other complex variables may become simple:

Example 3.3. *Consider*

$$\mathbf{S}_{@\mathbf{k}}(q,r,s,x,y) = (x, y, x+y+q+r, s+xq).$$

$\mathbf{S}_{@\mathbf{k},0}$ *and* $\mathbf{S}_{@\mathbf{k},1}$ *are simple.* $\mathbf{S}_{@\mathbf{k},2}$ *and* $\mathbf{S}_{@\mathbf{k},3}$ *are complex, as variables q, r and s are complex. If we assign q and r as* $q = 0$ *and* $r = 1$*, component* $\mathbf{S}_{@\mathbf{k},3}$ *become simple and* $\mathbf{S}_{@\mathbf{k}}$ *can have a simple parametric representation:*

$$\mathbf{PS}_{@\mathbf{k}}(p_0,p_1,p_2) = (p_0, p_1, 1, p_2).$$

Simple and complex variables (and functions) are identified in a two-pass scan of the BDDs of $\mathbf{S}_{@\mathbf{k}}$. Figure 3.7 shows the pseudocode for identifying them. We assume that initially, all component functions are labeled `UNBOUND`. The first `foreach` loop finds the support of each component of $\mathbf{S}_{@\mathbf{k}}$ and identifies simple variables. The second `foreach` loop identifies complex variables and places them in `Co`. It also classifies the functions whose support is all contained in `Si` as simple.

```
find_simple_complex_var(state_equations) {
   Si = Co = ∅;
   foreach (eq) in (state_equations) {
      if (support_size(eq) == 1) {
         Si = Si ∪ support(eq);
         assign_type(eq, SIMPLE);
      }
   }
   foreach (eq) in (state_equations) {
      if (support(eq) ∩ Si ≠ ∅) {
         csupp = support(eq) \ Si;
         if (csupp ≠ ∅) {
            Co = Co ∪ csupp;
            assign_type(eq, COMPLEX);
         } else {
            assign_type(eq, SIMPLE);
         }
      }
   }
   return <Si, Co>;
}
```

Figure 3.7: Classifying simple and complex variables - pseudocode

After complex variables are identified and removed, each component of $\mathbf{S}_{@\mathbf{k}}$ is labeled as either `SIMPLE` or `UNBOUND`. Unbound functions have no support variables in `Si`.

We then examine unbound functions. The simplest case occurs when one such function has support disjoint from all other components. For example, in Eq. 3.5 below:

$$\mathbf{S}_{@\mathbf{k}} = (f(p,q), x, y, g(x,y)). \tag{3.5}$$

the first component is unbound and has support disjoint from all others. The component can be replaced by an independent intermediate variable:

$$\mathbf{PS}_{@\mathbf{k}} = (p_0, p_1, p_2, g(p_1, p_2))$$

where

$$p_0 = f(p,q); \quad p_1 = x; \quad p_2 = y.$$

Consider now the more general situation:

$$\mathbf{S}_{@\mathbf{k}} = (f(p,q), \overline{f(p,q)}, x, y).$$

The first and second component of $\mathbf{S}_{@\mathbf{k}}$ can be replaced by p_0, $\overline{p_0}$ respectively.

Definition 3.4 partitions the set of unbound functions in $\mathbf{S}_{@\mathbf{k}}$ into equivalence classes. These classes can be discovered in a single scan of the array $\mathbf{S}_{@\mathbf{k}}$. If a value is assigned to all shared variables, then the support of each equivalence class will contain only bound variables, so that each class can be replaced by an independent parameter.

Example 3.4. *Consider*

$$\mathbf{S}_{@\mathbf{k}} = (x+y+z, \bar{x}\bar{y}\bar{z}, \bar{z}w, \bar{z}w).$$

By assigning the shared variable $z = 0$, the components of $\mathbf{S}_{@\mathbf{k}}$ become:

$$\mathbf{S}_{@\mathbf{k},z=0} = (x+y, \overline{x+y}, w, w)$$

A parametric representation of $\mathcal{R}(\mathbf{S}_{@\mathbf{k}})$ is then

$$\mathbf{PS}_{@\mathbf{k}} = (p_0, \overline{p_0}, p_1, p_1) \tag{3.6}$$

where $p_0 = x + y$ *and* $p_1 = w$.

Figure 3.8 shows the algorithm for finding shared variables. We first group the UNBOUND state expressions into equivalence classes. Then, we consider each variable in the support of these expressions, check if it belongs to one or more equivalence classes and tag it consequently.

```
find_shared_eqclasses(state_equations) {
   Sh = EC = ∅;
   foreach (eq) in (state_equations) {
      if (function_type(eq) == UNBOUND) {
         class = find_or_make_new_class(eq, EC);
         EC = EC ∪ class;
         foreach (x) in (support(eq)) {
            if (tag(x) == empty) tag(x) = class;
            else if (tag(x) ≠ class) tag(x) = shared;
         }
      }
   }
X   foreach (class) in (EC) {
      foreach (x) in (support(class)) {
         if (tag(x) == shared) Sh = Sh ∪ {x};
      }
   }
   return <Sh, EC>;
}
```

Figure 3.8: Classifying shared variables - pseudocode

3.4.3 The `remap` function

`remap` generates the new parameters for $\mathbf{PS}_{@\mathbf{k}}$ based on the results of the previous two routines. The first call remaps the variables in the *simple* set. Each of these variables is simply substituted by a new parameter variable in the state expressions with a single traversal of each of the BDDs. The second call remaps each equivalence class to a parameter. This operation is even simpler, since it just requires to represent each state equation with a single parameter based on the equivalence class it belongs to. The maximum numbers of parameters needed by the two calls is bounded by the number

of memory elements in the design to simulate. In fact, a new parameter is only assigned to Boolean expressions that occur at least once as a complete state equation. Thus, after parameterization, for each parameter, there is at least one equation whose expression is simply the parameter variable.

Example 3.5. *Suppose you are given a system to simulate with ten memory elements and eight inputs. After the first cycle of symbolic simulation, we obtain the following expressions for the state equations, where each combinational input was assigned a distinct Boolean variable literal a to h:*

$$\begin{array}{llll} s_0 = a & s_3 = ab & s_6 = d+e+f & s_8 = f+g \\ s_1 = a & s_4 = abc & s_7 = \overline{d}\overline{e}\overline{f} & s_9 = hg \\ s_2 = b & s_5 = b+c & & \end{array}$$

At first, all the equations are assigned the type `UNBOUND`*. With the first pass through the state equations, we detect the simple variables: a and b and we assign the type* `SIMPLE` *to* s_0*,* s_1 *and* s_2*. The second pass detects that* s_3 *is also simple, and classifies variable c and equations* s_4 *and* s_5 `COMPLEX`*. After evaluating variable c to* 0 *and remapping the simple variables, we obtain:*

$$\begin{array}{llll} s_0 = p_0 & s_3 = p_0 p_1 & s_6 = d+e+f & s_8 = f+g \\ s_1 = p_0 & s_4 = 0 & s_7 = \overline{d}\overline{e}\overline{f} & s_9 = hg \\ s_2 = p_1 & s_5 = p_1 & & \end{array}$$

At this point, we need to identify the equivalence classes for the remaining unbound functions. We find three equivalence classes: $\varepsilon_1 = \{s_6, s_7\}$*,* $\varepsilon_2 = \{s_8\}$*,* $\varepsilon_3 = \{s_9\}$*. Variables d and e are tagged with* ε_1*, h is tagged with* ε_2 *and f and g are* `shared`*. Consequently, we need to evaluate these last two variables to a constant value. We choose* 0 *for f and* 1 *for g. The set of equations at this point is:*

$$\begin{array}{llll} s_0 = p_0 & s_3 = p_0 p_1 & s_6 = d+e & s_8 = 1 \\ s_1 = p_0 & s_4 = 0 & s_7 = \overline{d}\overline{e} & s_9 = h \\ s_2 = p_1 & s_5 = p_1 & & \end{array}$$

and after remapping the unbound functions using one parameter for each equivalence class, we obtain:

$$s_0 = p_0 \qquad s_3 = p_0 p_1 \qquad s_6 = p_2 \qquad s_8 = 1$$
$$s_1 = p_0 \qquad s_4 = 0 \qquad s_7 = \overline{p_2} \qquad s_9 = p_3$$
$$s_2 = p_1 \qquad s_5 = p_1$$

Notice that the function in class ε_2 *was reduced to a constant, thus we did not need to use a parameter to remap it. This final set of equation is our new parameterized state vector. The* $\mathbf{PE_{@k}}$ *equations are:*

$$p_0 = a \qquad p_1 = b \qquad p_2 = d + e \qquad p_3 = h$$

Note that the number of parameters that are needed during each parameterization is always $\leq n$ where n is the number of state elements in the design. This is easy to derive based on the fact that for each parameter p_i there is at least one parameterized state equation $\mathbf{P}S_{@k,j}$ such that $\mathbf{P}S_{@k,j} = p_i$.

3.5 Implementation and complexity

In implementing the algorithm we made some observations that made possible to use the Boolean variables needed for the simulation efficiently. Since, in general, BDD packages can allow only a limited number of variables, this has also an impact on how many steps of simulation we can run. First, since we know that the number of parameters is bounded by the number of memory elements, we simply reserved an equivalent number of variables in the BDD manager for parameterization.

Second, we noticed that at the end of each parameterization step, the state equations do not depend on the combinational input variables any longer, but only on the parameters. Thus, we can reuse the same set of Boolean variables for the combinational inputs at every step of simulation. It follows that CBSS only needs a constant number of Boolean variables, equal to the number of inputs plus the number of states of the design to simulate. In contrast, a basic symbolic simulator

requires a new Boolean variable for each combinational input signal needs at each simulation step. Thus, a symbolic simulator needs a number of Boolean variables that depends on the length of the simulation and is equal to the number of combinational input signals times the number of simulation steps.

During simulation, the parametric equations **PE** at each step can be stored in BDD form. Since the variables used for these equations are the same involved in the simulation, sharing among the BDD nodes is possible and the additional memory required for these equations is not significant.

Moreover, while remapping the simple variables, we assign them in ascending variable order and we choose the parameters to reflect the same order, so that corresponding BDDs do not need to be recomputed, but can be simply duplicated and relabeled in a single pass. A more optimized approach would simply dynamically classify which variables are inputs and which are parameters, then, without modifying the BDDs at all, simple variables would just be reclassified as parameters at the next step of simulation and an equivalent number of parameters would become input variables to assign to the input signals.

The complexity of the algorithm can be computed considering each phase separately. We use here n for the number of states in the design, and **#BDD** for the size of the BDDs of the state equations:

- **simple variables** can be identified in a single pass of the state equations - $O(n)$.
- **complex variables** can be identified in another single pass of the state equations. We also need to cofactor each state equation w.r.t. to the complex variables, this can be done with a specialized cofactor routine that traverses each BDD once - $O(n \times \#\mathbf{BDD})$.
- **remapping simple variables** as we mentioned above can be done with a single pass of the state equations' BDDs - $O(\#\mathbf{BDD})$.
- **equivalence classes** can again be identified in a single pass of the state equations - $O(n)$.

- **shared variables** require similar treatment than complex variables, leading to the same worst case complexity - $O(n \times \#\mathbf{BDD})$.

- **remapping unbound functions** requires only assigning the proper parameter variable to each equivalence class - $O(n)$.

3.6 Experimental results

The CBSS algorithm was implemented in a C++ program and tested on the largest sequential circuits from the Logic Synthesis Benchmarks suite [68] and the ISCAS'89 Benchmark Circuits [16], including their 1993 additions. Table 3.1 reports results on all but the smallest testbenches of the two suites (we excluded from the table the circuits with less then 20 memory elements). The testbenches are grouped by benchmark suite. The experiments were run on a Linux PC equipped with a Pentium 4 processor running at 2.7Ghz and 2GB of memory and 512Kb of cache. As the underlying ROBDD package we used the CUDD package by Somenzi, [29], for which we set a reordering threshold of 200,000 nodes. We evaluated the simulator by running it for 5,000 symbolic simulation cycles on each testbench: at the end of each symbolic simulation step we would run our parameterization algorithm to simplify the state functions and then proceed to the next step. For the purpose of evaluating the performance of the approach, we chose a random Boolean value whenever we needed to evaluate complex and shared variables to constant. However, in a real-world context it is possible to choose the values based on the test stimulus, if one is available. For each circuit, the table reports first a few relevant metrics: the number of inputs In, outputs Out, memory elements FF, and internal network gates Gates.

The next three columns report the results of the parameterizations. The values are the average over the 5,000 steps of simulation. Our objective is to evaluate how many symbolic parameters we could find and the average number of states we could reach at each simulation step. To this end, the first of this group of columns, Param, reports the average number of symbolic parameters that we

Circuit	In	Out	FF	Gates	Parameterization			Time (s)		Efficiency	Memory (KB)	
					Params	Ass.d	Symbols	CBSS	Logic	ratio	CBSS	Logic
Logic Synthesis '91 - FSM tests												
ex1	9	19	20	622	0	0	9	0.69	0.04	29.68	4647	312
s1423	17	5	74	830	1.04	12.91	5.14	2.04	0.06	1.04	5818	320
s838	35	2	32	596	0.57	1.52	34.04	0.93	0.04	$7.62 \cdot 10^{8}$	4690	312
s953	16	23	29	658	1.15	6.1	11.05	1.36	0.04	62.19	5060	-
Logic Synthesis '91 - Addition '93												
bigkey	262	197	224	9211	0	228	34	163.49	0.55	$5.78 \cdot 10^{7}$	38255	516
clma	382	82	33	24482	1	0	383	75.77	1.5	$3.90 \cdot 10^{113}$	5078	836
dsip	228	197	224	3893	0	228	0	135.35	0.28	0	21289	404
mm9a	12	9	27	639	3.02	2	13.02	1.24	0.04	267.95	4658	-
mm9b	12	9	26	786	0	11.99	0.01	2.23	0.05	0.02	5339	-
mult16b	17	1	30	284	5.83	10.88	11.96	1.76	0.01	22.59	5563	308
mult32a	33	1	32	715	0.21	32.36	0.85	22.23	0.04	0	15980	-
s38417	28	106	1465	23771	47.5	19.66	55.83	190.55	1.67	$5.62 \cdot 10^{14}$	40613	956
s38584	38	304	1426	20281	7.48	25.71	19.77	488.99	1.35	2468.29	45244	864
s5378	35	49	163	3232	14.88	26.17	23.7	14.79	0.22	$2.03 \cdot 10^{5}$	13859	384
s838	34	1	32	618	0.5	1	33.5	0.85	0.04	$5.72 \cdot 10^{8}$	4690	-
s9234	36	39	135	3019	16.96	10.48	42.48	7.56	0.21	$1.70 \cdot 10^{11}$	5093	372
sbc	40	56	27	1143	2.92	22.22	20.7	3.76	0.07	$3.16 \cdot 10^{4}$	6066	324
ISCAS '89 - FSM tests												
s13207.1	62	152	638	9539	56.52	15.12	103.4	48.95	0.69	$1.89..10^{29}$	24684	568
s13207	31	121	669	9539	14.75	4.66	41.09	41.59	0.69	$3.88 \cdot 10^{10}$	9710	568
s1423	17	5	74	830	1.05	12.88	5.17	1.94	0.06	1.11	5834	-
s15850.1	77	150	534	11316	29.7	40.07	66.63	52.45	0.78	$1.69 \cdot 10^{18}$	35961	600
s15850	14	87	597	11316	4.39	2.78	15.61	35.69	0.74	$1.03 \cdot 10^{3}$	9386	604
s35932	35	320	1728	23085	1	35	1	194.66	1.67	0.02	38938	968
s38417	28	106	1636	27648	48.27	19.81	56.46	190.75	1.94	$1.01 \cdot 10^{15}$	39558	1068
s38584.1	38	304	1426	24619	7.45	25.75	19.71	475.62	1.68	3025.16	49685	972
s38584	12	278	1452	24619	6.26	6	12.27	271.83	1.65	29.88	45271	968
s5378	35	49	179	3973	14.86	26.13	23.73	14.95	0.06	$5.59 \cdot 10^{4}$	14226	-
s838	34	1	32	626	0.5	1	33.5	0.85	0.05	$7.15 \cdot 10^{8}$	4690	-
s9234.1	36	39	211	6585	18.06	19.51	34.55	16.61	0.43	$6.51 \cdot 10^{8}$	7965	464
s9234	19	22	228	6585	1.18	6.9	13.28	14.09	0.43	303.55	4964	464
s953	16	23	29	658	1.17	6.16	11.01	1.32	0.04	62.32	5029	312
ISCAS '89 - Addition '93												
prolog	36	73	136	1845	29.13	24	41.13	10.04	0.03	$7.20 \cdot 10^{9}$	9117	-
s1269	18	10	37	771	1.83	12.99	6.84	3.22	0.05	1.78	6019	312
s1512	29	21	57	990	9.85	5.93	32.92	2.29	0.06	$2.13 \cdot 10^{8}$	4960	324
s3271	26	14	116	2166	6.3	26	6.3	18.73	0.15	0.63	8295	352
s3330	40	73	132	2020	29.11	24.33	44.79	12.01	0.13	$3.28 \cdot 10^{11}$	9069	352
s3384	43	26	183	1734	53.32	17.99	78.33	10.6	0.14	$5.02 \cdot 10^{21}$	13529	352
s4863	49	16	104	2492	7.72	25.75	30.97	18	0.03	$3.50 \cdot 10^{6}$	8812	-
s6669	83	55	239	3272	77.86	68.73	92.13	275.47	0.04	$7.87 \cdot 10^{23}$	47362	388
s938	34	1	32	626	0.5	1	33.5	0.89	0.05	$6.82 \cdot 10^{8}$	4690	312
s967	16	23	29	677	1.25	6.13	11.12	1.41	0.05	78.98	5077	-

Table 3.1: Cycle Based Symbolic Simulation results

generate during a parameterization phase. For our second objective, we used the following reasoning: if we never evaluated a variable to constant, the number of symbols we had at each step would be given by the number of inputs symbols plus the number of parameters. However, since at every step some variables maybe be assigned to constant, we need to keep this into account by subtracting this amount from the number of live symbols that we carry across simulation steps. The average number of states that we reach at each step is then given by 2 to the power of this value, since, after parameterization each symbol doubles the number of states spanned by the parameterized state functions. The table shows the results we obtained with this evaluation: the second column of the group indicates the average number of symbolic variables that we assigned to a constant because they were classified as complex or shared variables, and the third column counts the number of live symbols as just described: Symbols = Param + IN - Ass.d. The actual size of the average state set visited at every step is 2^{Symbols}. This latter value also represents the average number of logic simulation equivalent traces that we carry on in parallel at every step.

The reminder of the table compares the results we obtained with CBSS to the performance of a compiled-level logic simulator. We built a logic simulator as described in Section 2.5 and we simulated again each of the testbenches for 5,000 cycles, providing a random stimuli to each circuit's inputs at each step. The two columns labeled Time compare the execution time for the CBSS simulation to the one for the logic simulator. We did not take into account the time spent compiling the circuit's netlist into assembly code for logic simulation. However, we measured this time and it was not transcurable: above 200s for the seven biggest benchmarks and above 1s for most of the testbenches. As the table indicates, once the compilation was completed, logic simulation could execute quite fast. As for the CBSS execution times, we point out that variable reordering was only triggered by the test *s6669* of the ISCAS suite, and thus it was not a factor for all the other benchmarks. Column Efficiency compares the performance of CBSS to logic simulation in terms of traces simulated per second of execution. Its value is computed as the ratio $2^{\text{Symbols}} \cdot$(Time-logic/Time-CBSS). It represents the number of traces visited by CBSS in the time of executing one logic simulation trace. A value of 1 in this column indicates that CBSS is providing the same

performance as a compiled-level logic simulator; when the value is less than 1, the logic simulator is more efficient; otherwise CBSS is providing "Efficiency" times better performance than a logic simulator. Note that most of the testbenches show an efficiency of 10-20 orders of magnitude over logic simulation, and this is particularly true for the most complex designs. Our intuition is that the more complex designs have more inputs and more memory elements that increase the possibility of discovering good parameterizations for the state vectors. For instance, the two variations of *s13207* in the ISCAS suite, provide very different efficiency results: the second one, having only half the inputs, can generate many fewer Symbols on average and thus it achieves lower efficiency. When the parameterization can only produce a small number of Symbols because of the high percentage of complex and shared variables, the extra time spent by CBSS in manipulating Boolean expressions makes this approach less attractive compared to logic simulation. This is the case mostly for the smaller designs, because of their limited potential for parameterizations.

Finally, the last two columns compare the memory profile of the two approaches. Even the smallest designs require a minumum of 4-5 KB to start the CUDD package in CBSS. However, the memory profiles are only moderately sensitive to the size of the design. As for the logic simulation memory column, we were able to collect the memory profile of the simulator only for the medium to large designs of the suites, and we report a '-' for the testbenches for which we could not gather this data. This last column can be used to gain an insight on the impact of design size over the memory profile of logic simulation, which can then be compared to the corresponding one for CBSS.

Overall we see that a high average number Symbols is key to a high efficiency over logic simulation. In general, testbenches that contain *highly sequential* components (such as counters) have a lower potential for good parameterizations: if the state bits of a counter take constant value at some point in time, that is, they are represented by constants, then they will be represented by constants also at the next clock tick. On the other hand, other circuits are more data-path intensive, they contain several large data-transfer or arithmetic operations, and in this cases it is easier to assign state bits independently, hence the larger number of parameter variables.

3.7 Conclusion

CBSS was published in [10]. This algorithm has shown to improve the scalability of symbolic simulation by providing a quick and memory friendly parameterization technique for the state equations. It can find quickly a large subset of the frontier set which can be represented very efficiently. The experimental results shows that in most cases we can achieve 10-20 orders of magnitude or more better efficiency over a compiled logic simulator.

However, in a few cases we noticed that many variables in the support of the state vector are complex or shared and need to be evaluated to constant. In those cases the performance is no longer competitive with logic simulation and the breadth of the state exploration is limited. In order to improve on the quality of the parameterization, we need to explore better techniques to represent the state vector through parameters. To this end, the next chapter introduces the theory of disjoint support decomposition of Boolean functions. This theory will be exploited in Chapter 6 to present an algorithm that can perform an exact parameterization of the state vector, thus guaranteeing to achieve the same maximal search breadth of symbolic simulation.

Chapter 4

Disjoint Support Decompositions

We introduce now a new property of logic functions which will be useful to further improve the quality of parameterizations in symbolic simulation. In informal terms, a function has a Disjoint Support Decomposition (DSD) when it can be expressed by composing two other functions such that there is no sharing between the variables in the support of each function. Moreover, in general a function may have multiple distinct DSD.

This chapter defines and characterizes the Disjoint Support Decomposition (DSD) of logic functions. We provide two novel contributions to the theory of DSD. First, we define a canonical form to represent simulataneously all the DSD of a logic function and we show that any Boolean function has a unique representation within this canonical form. As we discuss in the next section dedicated to the previous work, Ashenhurst had shown that, given a decomposition for a function, there is a unique way of assigning its variables to the support of the component functions in the decomposition (except when associative operators are involved). Our second result proves that the component functions are also uniquely determined, once we apply the rules of our canonical form.

The next chapter will make this theory and its properties applicable to any Boolean function that can be represented through a ROBDD by providing a novel algorithm that automatically exposes all the decompositions of a function by generating our canonical form . This algorithm takes as input a ROBDD representation of a function and returns a tree graph that represents its decompositions and

has worst-case complexity that is quadratic in the size of the ROBDD of the function.

4.1 Introduction

Disjoint support decomposability is an intrinsic property of Boolean functions. Given a Boolean function $F(x_1,\ldots,x_n)$, it is often possible to represent F by means of simpler component functions. When F can be represented by means of two other functions, say K and J, such that the inputs of J and K do not intersect, $F = K(x_1,\ldots,x_{j-1},J(x_j,\ldots,x_n))$, then we say that F has a *simple disjoint support decomposition*.

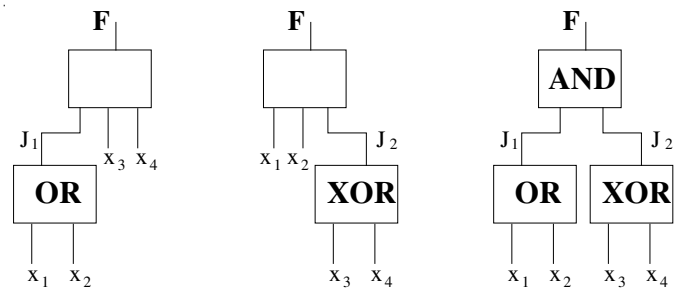

Figure 4.1: Decompositions for Example 4.1

Example 4.1. *The function $F = (x_1 + x_2)(x_3 \oplus x_4)$ has a simple disjoint support decomposition where $K = J_1(x_3 \oplus x_4)$ and $J_1 = x_1 + x_2$ as in Figure 4.1.a. The decomposition $K = (x_1 + x_2)J_2$ and $J_2 = x_3 \oplus x_4$ is also a simple disjoint support one (Figure 4.1.b). Note that it is possible to combine these two decompositions and represent the function as $F = K(J_1, J_2)$ where $K = j_1 j_2$ (Figure 4.1.c).*

A disjoint support decomposition can also be seen as a way of partitioning the inputs of a function, each element of the partition being the set of inputs to one of the component functions. For instance, with reference to the previous example, the decomposition in Figure 4.1.c corresponds to the partition $\{\{x_1,x_2\},\{x_3,x_4\}\}$. If we consider each of the inputs x_i of F, they can belong to the support of at most one of the component functions, otherwise we would violate the hypothesis of non-intersection. We can also guarantee that they belong to no less than one function; if that was

not the case, the resulting function would not have x_i in its support and thus it could not be equal to F.

When a function can be decomposed in more than one way, there is always a decomposition of maximal granularity, that is, a decomposition that imposes a finer partition on the support of F and such that the elements of this partition can be composed to generate all the other decompositions. The last decomposition of Example 4.1 is a maximal decomposition.

We discuss now some previous work on the subject and introduce some formal definitions related to disjoint support decompositions, before we present our contributions.

4.2 Related work on Disjoint Support Decompositions

Algorithms for extracting disjunctive decompositions are a classic research subject of switching theory. Ashenhurst and Singer [3, 63] developed the first theoretical framework in the '50s. In particular, Ashenhurst presented in [3] a classification of the various types of disjoint decompositions. They also introduced an algorithm to detect all the simple decompositions of a function based on *decomposition charts*. The method consists in partitioning the support variables of a function in two sets A and B and detecting if there exist a decomposition such that $F(A,B) = L(P(A),B)$. The method is efficient for functions of up to six variables and it is exponential in the number of variables in the support since it needs to try all the possible partitions of the variable support set. Ashenhurst showed in [3] how simple decompositions can be combined to obtain complex ones and proved that the partition of the support variables induced by decomposition is unique, with the exception of functions representing associative operations. Curtis and Karp explore applications for the theory in the area of synthesis of digital circuits in [30, 44].

In the early '70s, Shen *et al.* , [62], presented an algorithm based on the Jacobian that quickly rules out some partitions as candidates for a disjunctive decomposition. This method achieves good performance when used on undecomposable functions. However, it requires even more computation time for functions which have a decomposition. This algorithm has been implemented recently in

[60].

Alternative simplified techniques, such as algebraic factorization [14], have been extremely successful in transforming large two-level covers in multiple-level representations, and have been extended in various ways to include other forms of decomposition. Algebraic factoring [14] is a form of disjunctive decomposition. In algebraic factorization, one attempts to decompose a 2-level cover of F into a product $G * H$, where G and H have no variables in common. Factoring is a powerful step in passing from a Boolean cover to a multiple-level representation in multiple-level logic synthesis [15]. Logic synthesis have been also attempted starting directly from BDD representations: In [28] it was shown, for instance, that all implicants of a function could be implicitly represented in a BDD. A two-level synthesis algorithm, finding an optimal cover of a function from its BDD, was also developed in [53].

Links between BDDs and multiple-level logic have been explored in [46, 45, 47]. In [47], in particular, it is shown that particular BDD topologies may lead to the identification of particular decompositions. For instance, the presence of a *two-cut* (a partition of the BDD with only two boundary nodes) leads to the identification of disjunctive, MUX-based decompositions. On the other hand, the presence and aspect of two-cuts depends on the variable order of the BDD. Therefore, topological approaches must rely on tailored ordering algorithms.

Decomposition has also been considered in the context of technology mapping [55] and function representation [8, 7]. Bertacco and Damiani proposed in [7] a new function representation that merges BDD and a restricted type of decomposition. In that paper, it was shown that a special decomposition, using only NOR functions, is indeed canonical. If a function F can be decomposed into the NOR of disjoint-support components:

$$F = (f_1 + \cdots + f_n)' \tag{4.1}$$

then, provided that no component function f_i is itself the OR of other disjoint-support functions, the functions f_i are uniquely determined, up to a permutation.

This result was used to develop a hybrid normal form (MLDDs) for logic functions based on Shannon and disjoint-support NOR decompositions. Algorithms for translating BDDs into MLDDs and for the direct manipulation of MLDDs were also presented. This algorithm is capable of identifying a NOR-tree decomposition regardless of the variable ordering selected. The ability of discovering decomposition and the efficiency of the representation, however, are impaired by the restriction to NOR gates. Finally, a preliminary version of the material presented here and in the next chapter was developed by Bertacco and Damiani in [9].

4.3 Terminology

This section covers first a few background definitions that are required for the reminder of the presentation and then provide a formal definition of Disjoint Support Decompositions. For background definitions on Boolean functions the reader is referred to Section 2.3.

We introduce here a special class of functions which will be helpful for our purposes. It consists of those functions whose components are the permutations and/or complementations of the input variables $x_1, \cdots, x_n$ [22, 51].

Definition 4.1. *A function* $\mathbf{F}(x_1, \cdots, x_n)$*:* $\mathcal{B}^n \rightarrow \mathcal{B}^n$ *is termed a NP-function if for each of its components* F_i *either* $F_i = x_j$ *or* $F_i = \overline{x_j}$ *for some* j *and* $\mathcal{S}(F_i) \cap \mathcal{S}(F_k) = \emptyset, i \neq k$.

Definition 4.2. *Two functions* $F(x_1, \cdots, x_n)$ *and* $G(x_1, \cdots, x_n)$ *are said to be* **NP-equivalent** *if there is a NP-function* $\mathbf{NP}(x_1, \cdots, x_n)$ *such that*

$$F(x_1, \cdots, x_n) = G(\mathbf{NP}(x_1, \cdots, x_n)) \tag{4.2}$$

that is, F *is obtained by composing* G *with* **NP**.

We can now define the operation of decomposition of a function F as finding other, simpler

functions $L : \mathcal{B}^k \rightarrow \mathcal{B}$ and $A_1, \cdots, A_k$ such that

$$F(x_1, x_2, \cdots, x_n) = L(A_1(x_1, \cdots, x_n), A_2(x_1, \cdots, x_n), \cdots, A_k(x_1, \cdots, x_n)) \tag{4.3}$$

The following definition classifies the decomposing function L as a *divisor* of F and introduces the concept of *prime* function. In informal terms, a prime function is any function for which no disjoint support decomposition exists: for instance $F = a + b$ is prime, since it cannot be decomposed by any simpler function. Another example of prime function is the majority function: $F = ab + bc + ca$ has no decomposition through disjoint support components.

Definition 4.3. *A function $L(y_1, \cdots, y_k) : \mathcal{B}^k \rightarrow \mathcal{B}$ is said to* **divide** *a function $F(x_1, \cdots, x_n)$, $n \geq k \geq 2$ if there are k non-constant functions $A_1, \cdots, A_k : \mathcal{B}^n \rightarrow \mathcal{B}$ such that*

$$\begin{aligned} F(x_1, \cdots, x_n) &= L(A_1(x_1, \cdots, x_n), A_2(x_1, \cdots, x_n), \cdots) \\ \mathcal{S}(A_i) \cap \mathcal{S}(A_j) &= \emptyset; \qquad i \neq j \end{aligned} \tag{4.4}$$

If $n > k$, we say that L **divides** *F* **properly**.

F is said to be **prime** *if it cannot be divided properly by any L.*

Note that, based on the above definition, any function F can always be divided by itself, although improperly. We indicate by F/L any ordered list of functions $(A_1, A_2, \cdots)$ satisfying Eq. 4.4. The list of variables $y_1, \cdots, y_k$ and F/L will be termed **formals list** and **actuals list** of L, respectively.

Definition 4.4. *We call a* **disjunctive decomposition** *of F any pair $(L, F/L)$ that satisfies Eq. 4.4. We distinguish two situations to define* **maximal** *decompositions:*

- *If $L = y_1 \otimes y_2 \otimes \ldots y_n$, and $\otimes$ is one of the associative operators: OR, AND, XOR, the decomposition is said to be maximal iff none of the F/L can be further divided by the same operator, that is the cardinality of the inputs of L is maximal;*

- *Otherwise the decomposition is maximal iff L is a prime function.*

If a function L divides F, then any other function L' that is NP-equivalent to L will also divide F: The actuals list F/L' will be a permutation of the original ones, possibly with some functions $A_i \in F/L$ complemented.

Example 4.2. *Consider the function* $F = \overline{x_1}x_2 + \overline{x_1}x_3 + x_1x_4x_5$. *It can be divided by* $L(y_1,y_2,y_3) = y_1y_2 + \overline{y_1}y_3$. *The formals list is* (y_1,y_2,y_3), *while the actuals list is* (x_1,x_4x_5,x_2+x_3). *It can also be divided by* $L'(y_1,y_2,y_3) = \overline{y_2}\ \overline{y_3} + y_2y_1$. *In this second case the formals list is the same as before , while the actuals list is* $(x_4x_5,x_1,\overline{x_2+x_3})$. *Notice that L and* L' *are NP-equivalent.*

4.3.1 Decomposition trees.

As each function in the actuals list F/L may be itself decomposable, the lists associated with the decomposition of F and of its actuals, form a tree, hereafter called a **decomposition tree** for F.

Leaves of a decomposition tree of a function F are labeled by variables x_i or their complements $\overline{x_i}$. Nodes of the decomposition tree are labeled by a function L that divides the subfunction rooted at that node of the tree.

Example 4.3. *Consider the function* $F = \overline{h}((a \oplus b) \cdot MAJ(c,d,e+f) \cdot g) + h(k + j \cdot m)$. *Figure 4.2. represents its decomposition tree. The node labeled MUX corresponds to the function* $L(y_1,y_2,y_3) = y_3y_1 + \overline{y_3}y_2$ *with indexes of the formals list are increasing left to right for the edges in the picture. The node labeled MAJ corresponds to the function majority. The other nodes corresponds to AND, OR and XOR functions.*

4.4 The unique maximal Disjoint Support Decomposition

We can now introduce a characterization of decompositions and decomposition trees. In particular, we show in this Section our first novel result, that, under simple restrictions, every logic function has a unique decomposition tree, and that, much like its BDD, this decomposition tree is also unique. In general, it may be expected that any nontrivial function F can be divided by many functions.

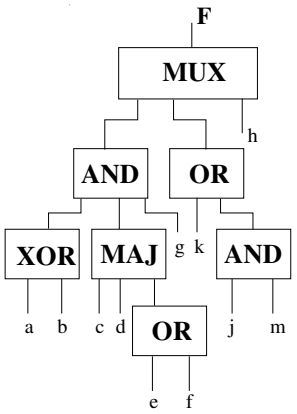

Figure 4.2: A decomposition tree for Example 4.3.

Moreover, for a given divisor L, one may expect that many different functions could contribute to F/L. This section provides a characterization of the divisors of F: we prove that there is actually a unique (modulo NP-equivalence - see Section 4.3) prime function L maximally dividing F, possibly coinciding with F.

We then show an important property of such prime function, namely, that it divides any other function M that divides F. This result leads to a partial ordering of Boolean functions based on the maximal divisor of any function F and is key to the definition of decomposition tree that is presented in Section 4.4.3.

In order to show the uniqueness of the maximal DSD, we now need to introduce the concept of *kernel* function. Notice first that all the Boolean functions with only two inputs can only be one of the associative operators: *AND*, *OR*, *XOR* or their complement or one of their NP-equivalent variants. These functions are also always prime, since they cannot be properly divided by any other function. If a function F can be divided by a prime function L that is a 2 inputs associative operator: AND_2, OR_2, XOR_2, we call *kernel* that function K_F that: 1) divides F, 2) is the same associative operator as L, but 3) has the maximum number of input operands. For instance if $F = a + be + cf$, it can be divided by $L = x_1 + x_2$, but its kernel function is $K_F = x_1 + x_2 + x_3$. In the case where the prime function L has more than 2 inputs, $|\mathcal{S}(L)| > 2$, the kernel function is L itself: $K_F = L$. We show in this section that for a given F, there is a unique K_F, Section 4.4.2 proves that F/K_F

is unique. The reason why we refer to K_F in our presentation is to disambiguate among the many similar functions that can divide a function F that is an associative operator (similar since they differ only in the number of input operands): we choose to use the one with maximum granularity because that is the only one that can impose a unique partitioning on the actuals list of F, F/K_F.

In order to prove the results, we need to introduce some auxiliary terminology.

Definition 4.5. *Given a set of variables $\mathcal{S}$, a* **partition** *$\mathcal{P}$ of $\mathcal{S}$ is a collection of disjoint subsets of $\mathcal{S}$:*

$$\begin{aligned} &\mathcal{P} = \{\mathcal{S}_1, \mathcal{S}_2, \cdots, \mathcal{S}_k\} \\ &\mathcal{S}_i \neq \emptyset; \qquad i = 1, \cdots k \\ &\mathcal{S}_i \cap \mathcal{S}_j = \emptyset, \qquad i, j = 1, \cdots k; \qquad i \neq j \\ &\bigcup \mathcal{S}_i = \mathcal{S} \end{aligned} \tag{4.5}$$

Given a partition $\mathcal{P}$ of $\mathcal{S}$ into k subsets $\mathcal{S}_1, \cdots \mathcal{S}_k$, we call a **selection** *of $\mathcal{S}$ a subset $\mathcal{S}^{\mathcal{P}}$ of $\mathcal{S}$ containing exactly k variables $x_1, \cdots, x_k$, where $x_i \in \mathcal{S}_i$.*

In other words, $\mathcal{S}^{\mathcal{P}}$ contains one representative variable for each subset $\mathcal{S}_i$ in the partition $\mathcal{P}$.

4.4.1 Decomposition by prime functions.

We first show that any function is decomposed by a unique prime function L. Among all the functions that divide a function F, the prime function L is the one with the smallest number of inputs. We prove here that there is a unique such function L for any F and that any other divisor M with a larger number of inputs, can be further divided. The following Lemma demonstrates this last statement and it is used to show the uniqueness of the prime function L in Theorem 4.2.

Lemma 4.1. *Consider an arbitrary function $F(x_1, \cdots, x_n)$, $n \geq 2$, and let L denote a function dividing F. Then, for any other function M dividing F, if $|\mathcal{S}(M)| > |\mathcal{S}(L)|$, M is decomposable.*

Proof. Let $A_1, A_2, \cdots, A_{|\mathcal{S}(L)|}$ denote the functions in F/L. Recall that such functions are all non-constant and share no support variables. These properties must also hold for the functions in F/M, hereafter listed as $P_1, P_2, \cdots, P_{|\mathcal{S}(M)|}$.

The starting point of the proof is the presumed equality

$$F = L(A_1, A_2, \cdots) = M(P_1, P_2, \cdots). \tag{4.6}$$

The sets $\mathcal{S}(P_1), \cdots, \mathcal{S}(P_{|\mathcal{S}(M)|})$ form a partition of $\mathcal{S}(F)$. Consider building a selection from this partition. We indicate with $x_{P_1}, x_{P_2}, \cdots$ the selected variables, and with $\mathcal{X}_M$ the selection $\{x_{P_1}, x_{P_2}, \cdots\}$ just constructed. Notice in particular that $|\mathcal{S}(M)| = |\mathcal{X}_M|$.

The selection must satisfy one additional property: $\mathcal{X}_M$ must be such that for at least two functions in F/L, say, A_1 and A_2,

$$\mathcal{X}_M \cap \mathcal{S}(A_1) \neq \emptyset \qquad \text{and} \qquad \mathcal{X}_M \cap \mathcal{S}(A_2) \neq \emptyset. \tag{4.7}$$

It is always possible to construct $\mathcal{X}_M$ so that Eq. 4.7 holds, as follows. If Eq. 4.7 is not satisfied by a current selection $\mathcal{X}_M$, then it must be (say) $\mathcal{X}_M \cap \mathcal{S}(A_2) = \emptyset$. Select a variable x_{A_2} from $\mathcal{S}(A_2)$. Notice that x_{A_2} also belongs to the support of some function in F/M, say, to $\mathcal{S}(P_1)$. By replacing x_{P_1} with x_{A_2} in $\mathcal{X}_M$ the new set is still a valid selection, and it satisfies Eq. 4.7.

Consider assigning constant values to the variables not belonging to $\mathcal{X}_M$, in the following way. Since $x_{P_i} \in \mathcal{S}(P_i)$, it is always possible to assign values to the remaining variables of $\mathcal{S}(P_i)$ in such a way that, under this assignment, $P_i = x_{P_i}$ or $P_i = \overline{x_{P_i}}$. For our purpose, complementation is irrelevant. Hence, we will assume for simplicity that this assignment results in $P_i = x_{P_i}$.

We indicate with f^* a function resulting from another function f after this partial assignment.

By applying the same partial assignment to the left-hand side of Eq. 4.6, we obtain

$$L(A_1^*, A_2^*, \cdots) = M(x_{P_1}, x_{P_2}, \cdots). \tag{4.8}$$

Notice that the support of the right-hand side of Eq. 4.8 is precisely $\mathcal{X}_M$. Because $|\mathcal{X}_M| = |\mathcal{S}(M)| > |\mathcal{S}(L)|$, the support of at least one of the functions $A_1^*, A_2^*, \cdots$ must contain 2 or more variables. Because of the partial assignment, some of the functions A_i^* may actually be constants, and the function L may simplify to a different function L^*. From Eq. 4.7, however, at least two functions A_i^* are not constant, hence $|\mathcal{S}(L^*)| \geq 2$ and L^* is a function of at least two inputs. Eq. 4.8 then indicates that L^* divides M and M/L^* is the set of non-constant A_i^*. □

We can now prove that the the prime function L dividing F is unique:

Theorem 4.2. *Let L denote a prime function dividing F. Then, L divides any other function M that divides F.*

Proof. Consider a selection $\mathcal{X}_M$ of $|\mathcal{S}(M)|$ variables and a sensitizing assignment as in the Proof of Lemma 4.1. Eq. 4.6 then reduces to:

$$L(A_1^*, A_2^*, \cdots) = M(x_{P_1}, x_{P_2}, \cdots). \tag{4.9}$$

(remember that f^* is the function resulting from a function f after a partial assignment).

By the way of choice of the variables x_{P_j}, we know that at least two of the functions A_i^* are not constant. We now show that, because of the primality of L, actually none of them can be constant. If, by contradiction, any of the A_i^* were a constant, then L could be replaced in Eq. 4.9 by a function L^* such that $|\mathcal{S}(L^*)| < |\mathcal{S}(L)|$. L^* would also have at least two inputs because at least two A_i^* are not constant. Hence, Eq. 4.9 would indicate that L^* divides M, and therefore it would divide F. We would then have two functions, namely, L and L^*, with $|\mathcal{S}(L)| > |\mathcal{S}(L^*)|$, that divide F. From Lemma 4.1, L would then be decomposable, against the assumption. None of the A_i^* of Eq. 4.9 can then be constant.

Suppose first $|\mathcal{S}(M)| > |\mathcal{S}(L)|$. At least one of the functions A_i^* must have support size larger than one. Hence, Eq. 4.9 indicates that L divides M, and $M/L = \{A_i^*\}$.

If $|\mathcal{S}(M)| = |\mathcal{S}(L)|$, each of the A_i^* must be either a variable x_j from the selection or its complement. In other words, $(A_1, \cdots, A_{|\mathcal{S}(L)|}) = \mathbf{NP}(x_1, \cdots, x_{|\mathcal{S}(M)|})$. Therefore M is NP-equivalent to L. □

Example 4.4. *Consider the function $F(x_1,x_2,x_3,x_4) = x_1x_2x_3 + x_1x_2x_4 + x_3x_4$. It can be decomposed as $F = MAJORITY(x_1x_2, x_3, x_4)$. It is easy to verify that MAJORITY is prime. From Theorem 4.2, F cannot be decomposed with any prime function L other than MAJORITY, while maintaining arguments with disjoint support. It follows that MAJORITY is the kernel of F.*

The following result is a direct application of the previous Theorem. It has relevant applications, however, in the rest of the present work.

Corollary 4.3. *If a function F can be decomposed into the 2-input OR (AND, XOR) of two disjoint-support functions, then it cannot be decomposed using any of the other two operators.*

4.4.2 A characterization of F/K_F.

We complete the uniqueness results by providing a characterization of the functions in F/K_F. Theorem 4.4 below, in particular, considers the case where the prime L is not an associative operator. It follows that $|\mathcal{S}(L)| > 2$. As we discussed in the previous Section, in this case $K_F = L$. We show here that, in this situation, the functions in F/K_F are unique. The case where $|\mathcal{S}(L)| = 2$ and $K_F \neq L$ is then considered in Theorem 4.5.

Theorem 4.4. *Let L denote a prime function, with $|\mathcal{S}(L)| > 2$, and let $\mathcal{S}_A = \{A_1, A_2, \cdots, A_{|\mathcal{S}(L)|}\}$, $\mathcal{S}_B = \{B_1, B_2, \cdots, B_{|\mathcal{S}(L)|}\}$ denote two sets of disjoint-support functions such that*

$$L(A_1, A_2, \cdots) = L(B_1, B_2, \cdots) \tag{4.10}$$

Then, there exists a NP-function **NP** *such that*

$$(A_1, A_2, \cdots) = \mathbf{NP}(B_1, B_2, \cdots). \tag{4.11}$$

In other words, $A_1, A_2, \cdots$ *coincides with* $B_1, B_2, \cdots$ *or their complements.*

Proof. We prove the result by contradiction: We assume that two sets of functions exist that satisfy Eq. 4.10 but violate Eq. 4.11, and draw the conclusion that L is not prime.

The proof mechanism is again based on building a selection from the supports of $B_1, B_2, \cdots$.

We need consider two cases. In the first case, the support partition induced by $A_1, \cdots$ coincides with that of $B_1, \cdots$. In the second case, it does not.

First case.

It is not restrictive to assume that $\mathcal{S}(A_1) = \mathcal{S}(B_1); \mathcal{S}(A_2) = \mathcal{S}(B_2), \cdots$. Consider any assignment that is complete for the variables in $\mathcal{S}(A_2), \mathcal{S}(A_3), \cdots$, but that does not assign any values to those variables in $\mathcal{S}(A_1)$. In this case, all functions except A_1 and B_1 reduce to constants. Let $a_2, a_3, \cdots$ denote the constant values $A_2^*, A_3^*, \cdots$. Eq. 4.10 reduces to

$$L(A_1, a_2, a_3, \cdots) = L(B_1, b_2, \cdots)\ . \tag{4.12}$$

Notice that we can always choose the values $a_2, a_3, \cdots$ in such a way that

$$L(A_1, a_2, \cdots) = A_1 \qquad \text{or} \qquad L(A_1, a_2, \cdots) = \overline{A_1}. \tag{4.13}$$

In this case, Eq. 4.12 becomes

$$A_1 = L(B_1, b_2, \cdots) \qquad \text{or} \qquad A_1 = \overline{L}(B_1, b_2, \cdots). \tag{4.14}$$

Since A_1 is, by construction, not a constant, Eq. 4.14 indicates that $L(B_1, b_2, \cdots)$ is also non-constant. On the other hand, L has only one non-constant argument, namely, B_1, and therefore

$L(B_1, b_2, \cdots)$ coincides with either B_1 or with $\overline{B}_1$. Hence, Eq. 4.14 ultimately implies that

$$A_1 = B_1 \qquad \text{or} \qquad A_1 = \overline{B}_1. \tag{4.15}$$

By repeating the same reasoning for all functions in S_A, eventually $(A_1, A_2, \cdots) = \mathbf{NP}(B_1, B_2, \cdots)$ for some NP function.

Second case.

Suppose the support of one function of $\mathcal{S}_A$ (say, A_2) overlaps with the support of (at least) two functions B_j (say, B_1 and B_2). Consider constructing a selection $\mathcal{X}_B$ from $\mathcal{S}_B$ containing at least two variables from $\mathcal{S}(A_2)$. We impose one more requirement on $\mathcal{X}_B$, namely, that for at least another function $A_i, i \neq 2$ $\mathcal{X}_B \cap \mathcal{S}(A_i) \neq \emptyset$.

It is always possible to construct such a selection. If, for a current selection $\mathcal{X}_B$, $\mathcal{X}_B \cap \mathcal{S}(A_i) = \emptyset; i \neq 2$, it would imply $\mathcal{X}_B \subseteq \mathcal{S}(A_2)$. Choose then a variable from, say, $\mathcal{S}(A_3)$. This variable must belong to the support of some B_j. Replace then x_{B_j} with this new variable. For the new selection, $\mathcal{X}_B \cap \mathcal{S}(A_3) \neq \emptyset$, and, since $|\mathcal{X}_B| \geq 3$, at least two variables still belong to $\mathcal{S}(A_2)$.

Notice that, since at least two variables belong to $\mathcal{S}(A_2)$, for at least another function of $A_1, A_2, \cdots$ (say, A_1) $\mathcal{X}_B \cap \mathcal{S}(A_1) = \emptyset$.

Consider applying a partial assignment, such that all functions $B_1, B_2, \cdots$ reduce to variables in $\mathcal{X}_B$ or their complements: $B_i^* = x_{B_i}$. Since no variables of A_1 are included in $\mathcal{X}_B$, A_1 reduces to a constant a_1, and Eq. 4.10 becomes

$$L(a_1, A_2^*(x_{B_1}, x_{B_2}), A_3^*, \cdots) = L(x_{B_1}, x_{B_2}, \cdots) \tag{4.16}$$

Notice that, since the left-hand side of Eq. 4.16 must have support $\mathcal{X}_B$:

1. A_2^* is not a constant;

2. A_3^* is not a constant;

Because a_1 is a constant, however, we can replace L by a simpler function L^* :

$$L^*(A_2^*(x_{B_1}, x_{B_2}), A_3^*, \cdots) = L(x_{B_1}, x_{B_2}, \cdots) \tag{4.17}$$

Eq. 4.17 then indicates that we have been able to decompose L using L^*, and $L/L^* = \{A_2^*, A_3^*, \cdots\}$. This contradicts the assumption that L be a prime function. Hence, this second case is impossible, and F/L is unique. □

Notice that the constraint $|\mathcal{S}(L)| > 2$ is essential to the proof, for if $|\mathcal{S}(L)| = 2$, Eq. 4.16 reduces to

$$L(a_1, A_2^*(x_{B_1}, x_{B_2})) = L(x_{B_1}, x_{B_2}) \tag{4.18}$$

indicating only that A_2^* coincides with L or its complement.

Example 4.5. *Consider again the function $F(x_1, x_2, x_3, x_4) = x_1x_2x_3 + x_1x_2x_4 + x_3x_4$. Its kernel function is MAJORITY. From Theorem 4.4, the only possible elements of the actuals list are $A_1 = x_1x_2$, $A_2 = x_3$ and $A_3 = x_4$ or any other NP-quivalent set.*

Example 4.6. *Consider the function $F = x_1 + x_2 + x_3$. It can be decomposed using the function $OR_2(a, b)$ at the root. From Corollary 4.3, no other prime function can be used. The functions F/OR_2, however, are not uniquely identified, as $A_1 = x_1 + x_2$, $A_2 = x_3$ and $A_1 = x_1$, $A_2 = x_2 + x_3$ are both legitimate choices.*

We now address the case where the prime function L dividing F is a 2-input function. It is convenient to restrict our attention to the associative operators OR, AND, XOR: All other 2-input functions are in fact NP-equivalent to one of these operators. Example 4.6 already showed that the inputs of L are not identified uniquely. Because the operators are associative, however, instead of decomposing F using only two arguments A_1, A_2, we allow the number of inputs to the divisor function to be as large as possible and in this case we call K_F such maximum-inputs divisor function.

To this end, we report here a result derived from [7] :

Theorem 4.5. *Suppose a function F is decomposable using one binary associative operator $\otimes$ (where $\otimes = AND, OR, XOR$) as*

$$F = A_1 \otimes A_2 \otimes \cdots \otimes A_n \tag{4.19}$$

and suppose further that none of the component functions A_i is further decomposable using $\otimes$; then the set of functions $\{A_1, \cdots, A_n\}$ is :

- *unique in the case of AND, OR decompositions.*
- *unique modulo complementation for XOR decompositions.*

Proof. The proof follows by contradiction. Assume that there exist two distinct sets of component functions that decompose F, namely, $\{A_1, \cdots, A_n\}$ and $\{B_1, \cdots, B_q\}$; we show that this leads necessarily to the violation of some properties of the functions A_i or B_i.

Consider first the case where $\otimes = OR$. Since the two sets are distinct, at least one of the functions B_j (say, B_1) must differ from any of the functions A_i. Since $\{A_i\}, \{B_j\}$ are both actuals lists for the decomposition of F, it must be :

$$A_1 + \cdots + A_n = B_1 + \cdots + B_q \ . \tag{4.20}$$

Since all functions B_i have disjoint support, it is possible to find a partial assignment of the variables such that $B_j = 0, j = 2, \cdots, q$. Notice that the variables in $\mathcal{S}(B_1)$ have not been assigned any value. Corresponding to this partial assignment, Eq. 4.20 becomes:

$$A_1^* + \cdots + A_n^* = B_1 \tag{4.21}$$

In Eq. 4.21, A_i^* denotes the residue function obtained from A_i with the aforementioned partial assignment.

We need now to distinguish several cases, depending on the assumptions on the structure of the

left-hand side of Eq. 4.21.

1. The left-hand side reduces to a constant. Hence, B_1 is a constant, against the assumptions.

2. The left-hand side contains two or more terms. Since these terms must have disjoint support,B_1 is further decomposable by *OR*, against the assumptions.

3. The left-hand side reduces to a single term. It is not restrictive to assume this term to be A_1^*. If $A_1 = A_1^*$, then we have $B_1 = A_1$, against the assumption that B_1 differs from any A_i. Hence, it must be $A_1^* \neq A_1$, and

$$\mathcal{S}(B_1) = \mathcal{S}(A_1^*) \subset \mathcal{S}(A_1) \qquad \text{strictly.} \tag{4.22}$$

We now show that also this case leads to a contradiction.

Consider a second assignment, zeroing all functions $A_i, i \neq 1$. Eq. 4.20 now reduces to

$$A_1 = B_1^* + \cdots + B_q^* \tag{4.23}$$

By the same reasonings carried out so far, the r.h.s. of Eq. 4.23 can contain only one term. We now show that this term must be B_1.

In fact, if $A_1 = B_j^*, j \neq 1$, then by Eq. 4.22 one would have

$$\mathcal{S}(A_1) = \mathcal{S}(B_j^*) \supset \mathcal{S}(B_1) \tag{4.24}$$

against the assumption of B_1, B_j being disjoint-support. Hence, it must be $A_1 = B_1^*$. In this case, by reasonings similar to those leading to Eq. 4.22, we get

$$\mathcal{S}(A_1) = \mathcal{S}(B_1^*) \subset \mathcal{S}(B_1) \qquad \text{strictly} \tag{4.25}$$

which contradicts Eq. 4.22. Hence, B_1 cannot differ from any A_i.

The case where $\otimes = AND$ is derived similarly, with the only difference that we choose partial assignments such that the component functions evaluate to 1.

Finally, for the case $\otimes = XOR$, we choose partial assignments such that the component functions evaluate to 0. Case 1) and 2) are still analogous to the previous derivation above. For case 3), we may reduce to either $B_1 = A_1^*$ or $B_1 = \overline{A_1^*}$. However, in both cases the relation between the supports still holds, in particular Eq. 4.22 and 4.24 are still valid. From the two equations we can then derive the contradiction. □

The Corollary below indicates how the decomposition of Theorem 4.5 is the common denominator of all the other decompositions through an associative operator:

Corollary 4.6. *Suppose F is divided by an associative operator $\otimes$, and let $B_1, \cdots, B_q$ denote a collection of disjoint-support functions such that*

$$F = B_1 \otimes B_2 \otimes \cdots \otimes B_q. \tag{4.26}$$

Then each function B_j can be expressed using terms from the actuals list F/K_F:

$$B_j = A_{k_j+1} \otimes \cdots A_{k_{j+1}} \qquad with \qquad k_1 = 0, k_q = k. \tag{4.27}$$

In other words, F/K_F forms a base *for expressing all possible ways of decomposing F using $\otimes$.*

Proof. We prove the results only for $q = 2$, $\otimes = OR$, the generalizations being straightforward.

Consider the equality

$$F = B_1 + B_2 = A_1 + \cdots + A_k. \tag{4.28}$$

Consider two distinct partial assignments leading to $B_1 = 0$ and to $B_2 = 0$, respectively. Eq. 4.28

becomes

$$B_2 = A_1^* + \cdots + A_k^*; \tag{4.29}$$

and

$$B_1 = A_1^{**} + \cdots + A_k^{**}. \tag{4.30}$$

By computing the *OR* of the two components,

$$F = B_1 + B_2 = A_1^* + A_1^{**} + A_2^* + A_2^{**} + \cdots + A_k^* + A_k^{**} \tag{4.31}$$

From Theorem 4.5, there can be at most k terms in the right-hand side of Eq. 4.31. For each function A_i, at least one of A_i^*, A_i^{**} must be nonzero (or otherwise $\mathcal{S}(A_i) \cap \mathcal{S}(F) = \emptyset$). Hence, for each A_i, either $A_i^* = 0$ and $A_i^{**} = A_i$, or $A_i^* = A_i$ and $A_i^{**} = 0$. Each term in the right-hand sides of Eq. 4.29 is then either 0 or coincides with some A_i. It is not restrictive to assume that the first k_1 terms are nonzero. Hence, Eqs. 4.29 and 4.30 reduce to Eq. 4.27. □

In summary, a function can be decomposed in exactly one of the following ways :

1. By the binary associative operators *AND* or *OR*. In this case, hereafter K_F denotes the *AND* or *OR* function with the largest support size and K_F is unique in the sense of Theorem 4.5.

2. By an *XOR* operator. Also in his case F/K_F will be taken to denote the finest-grain decomposition. F/K_F is unique modulo complementation of an even number of its elements.

3. By a PRIME function of three or more inputs. F/K_F is unique modulo complementation of some of its elements.

Note that the complement of a function has a decomposition that can be derived immediately from the decomposition of the function: If a function is OR-decomposable, its complement has an

AND-decomposition where the inputs are complemented and conversely. The complement of functions with XOR-decompositions are also XOR-decomposition with one of the inputs complemented. Finally, PRIME-decompositions have complements which are PRIME-decompositions with the kernel function K_F complemented.

4.4.3 The normal Decomposition Tree

In Sections 4.4.1 and 4.4.2, we showed that for a given function F, the kernel K_F and the actuals F/K_F are unique, up to complementations and permutations. We now establish some conventions so as to choose a unique representative of all the decomposition trees corresponding to the same function F. These conventions will lead to the definition of the normal Decomposition Tree.

In representing decomposition trees, we reference the tree by a pointer to the root node. Moreover, we use *signed* edges, much like common ROBDD representations ([13]). If a decomposition tree represents the function F, the tree obtained by complementing the edge to the root node represents the function $\overline{F}$.

Definition 4.6. *Given a function, its* **normal Decomposition Tree** *is a tree graph and it is denoted* $\mathbf{DT}(F)$.

It is defined recursively as follows.

The root node represents F, and it is labeled by the type of decomposition. The root node has $|F/K_F|$ outgoing edges, each edge pointing to the root of $DT(A_i)$. In order to resolve permutation ambiguities, the elements of F/K_F are ordered according to the order of their top variable in their ROBDD representation.

In order to resolve complementation ambiguities, the following rules are adopted:

- *If F has PRIME or XOR decomposition, the set F/K_F contains functions with positive ROBDD polarity. In the case of XOR decomposition, the root node will be referred to through a complement edge if necessary.*

- *If F has an AND decomposition, DeMorgan rule is applied: the root node is labeled OR and*

will be referred to through a signed edge. The fanout edges of the root node point to the complements of F/AND_k.

From Theorems 4.2 and 4.4, it follows trivially that with this set of conventions and with the full labeling of *PRIME* nodes, there is a one-to-one correspondence between normal decomposition trees and logic functions.

Example 4.7. *Consider the function* $F = MAJORITY(a \oplus b, cd + e, ITE(fg, h, i))$. *F has the following disjoint-support representation:*

$$\begin{aligned} F &= MAJORITY(G, H, I); \\ G &= a \oplus b; \\ H &= L + e; \\ I &= ITE(M, h, i); \\ L &= cd; \\ M &= fg; \end{aligned}$$

The data structure of its normal decomposition tree is reported in Figure 4.3. Notice that the representation is normalized by representing each AND decomposition with it's dual OR and using complement edges. Moreover, since the function I is decomposed through a PRIME, all of its actuals list element must have positive polarity. Thus, the first element A_1 *for the decomposition of I is the function* $\overline{fg}$ *instead of* fg *and the kernel we use for I is* $K_I = \overline{x_0}x_1 + x_0x_2$ *which takes into account the polarity change.*

The decomposition tree represents concisely all possible disjunctive decompositions of F. This property will be useful to the decomposition algorithm presented in the next chapter. In order to extract a decomposition from a normal Decomposition Tree, we need to define the concept of a *cut* of a DT. Given a generic tree graph, a cut is any set of nodes that separates each leaf of the tree from the root and such that in any path from the root to the leaves there is only one node that

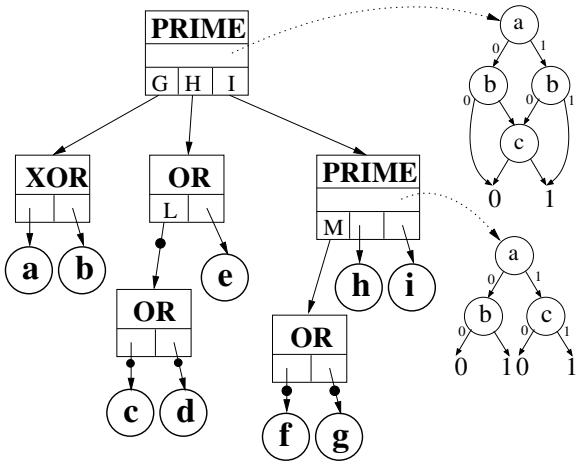

Figure 4.3: Decomposition representation of the function of Example 4.7

belongs to the cut. Since for our Decomposition Trees each node corresponds to a function, we define cuts as a collection of functions. The last lemma of this chapter shows that there is a one to one correspondence between divisors of a function F and cuts through its normal Decomposition Tree.

To formalize this aspect of decomposition trees, we need to introduce a few definitions, which will be used again when we describe our decomposition algorithm.

Definition 4.7. *We say that a function* $G(x_1, \cdots, x_n)$ **appears explicitly** *in* $DT(F)$ *if one of the following holds:*

1. $G = F$, *or*

2. G *appears explicitly in* $DT(A_i)$ *for some* A_i *in* F/K_F.

In other words, functions appearing explicitly in $DT(F)$ *correspond to tree nodes.*

We say G **appears implicitly** *in* $DT(F)$ *if one of the following holds :*

1. $G = \overline{F}$

2. $F = \otimes(A_1, \cdots, A_n)$ *and* $G = \otimes(B_1, \cdots, B_m)$, *where* $\otimes$ *is OR, XOR, and where each* $B_i \in \{A_1, \cdots, A_n\}$

3. *F and B_i are as above and $G = \overline{\otimes(B_1, \cdots, B_m)}$*

4. *G appears implicitly in one of the subtrees $DT(A_i)$.*

Finally, we say G **appears** *in $DT(F)$ if it appears explicitly or implicitly.*

Example 4.8. *Consider the function $F = x_1x_2x_3 + x_4x_5$. A decomposition tree is reported in Figure 4.4.a. Figure 4.4.b depicts the equivalent normal Decomposition Tree (the Figure uses the* signed edges *convention to represent the complementation of a node in the tree). The functions $G_1 = F$, $G_2 = x_1x_2x_3$, $G_3 = x_4x_5$, and all the functions corresponding to a simple input variable, appear explicitly in $DT(F)$ and are indicated in the Figure. Functions $G_4 = \overline{x_1} + \overline{x_2}$, $G_5 = \overline{x_1} + \overline{x_3}$ and $G_6 = \overline{x_2} + \overline{x_3}$ appear implicitly in the decomposition tree by rule (2) on implicit appearance of Definition 4.7. Moreover, functions $G_7 = x_1x_2$, $G_8 = x_1x_3$ and $G_9 = x_2x_3$ also appear implicitly by rule (3) of the Definition.*

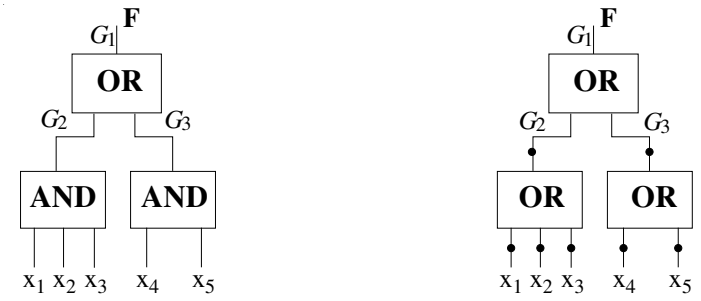

Figure 4.4: Decomposition tree for Example 4.8.

Notice that if a function G appears in $DT(F)$, then every function in $DT(G)$ will also appear in $DT(F)$.

Definition 4.8. *A set of functions $C = \{A_i\}$ is called a* **cut** *of $DT(F)$ if the following hold:*

1. *each function A_i appears in $DT(F)$.*

2. $\mathcal{S}(A_i) \cap \mathcal{S}(A_j) = \emptyset; \qquad i \neq j$

3. $\bigcup_{A_i \in C} \mathcal{S}(A_i) = \mathcal{S}(F)$.

Example 4.9. *A possible cut for the function of Example 4.8 is given by the set* $\{G_3, G_4, x_3\}$*, that is,* $\{x_4x_5, x_1x_2, x_3\}$*. Notice that* G_4 *appears only implicitly.*

Lemma 4.7. *For every function M dividing F,* F/M *is a cut of* $DT(F)$*. Conversely, for any cut* $\mathcal{C}$ *of* $DT(F)$*, there is a function M such that* $F/M = \mathcal{C}$*.*

Proof. The first part of the theorem is proved by induction on the number of variables in $\mathcal{S}(F)$.

The base of the induction (when $|\mathcal{S}(F)| = 2$) is trivial. For the generic induction step, let $m = |\mathcal{S}(M)|$ and let $y_1, \cdots, y_m$ denote formal inputs to M.

Since M divides F, there exist m functions $P_1(x_1, \cdots), P_2, \cdots$ (the actuals list of F/M) such that:

$$F = M(P_1, P_2, \cdots, P_m). \tag{4.32}$$

By assumption, $P_1, \cdots, P_m$ are disjoint support and their support must coincide with $S(F)$. Thus, we need to show only that $P_1, \cdots, P_m$ appear in $DT(F)$.

From Theorem 4.2, the prime function L that divides F, divides also M. Therefore, there exist $l = |S(L)|$ disjoint-support functions $B_1, B_2, \cdots, B_l$ of $y_1, \cdots, y_m$ such that

$$M = L(B_1, \cdots, B_l). \tag{4.33}$$

It is not restrictive to assume that the support variables y_i are numbered so that

$$\mathcal{S}(B_i) = \{y_{b_{i-1}+1}, \cdots, y_{b_i}\} \qquad i = 1, \cdots, l \tag{4.34}$$

for suitable integers b_i, with $b_0 = 0$ and $b_l = m$.

We now need to distinguish whether L is a 2-input function (i.e. an associative operator), or a prime function with three or more inputs. The latter case is simpler and we carry it out first.

By composing Eqs. 4.32 and 4.33 one obtains

$$F = M(P_1, \cdots, P_m) = L(B_1(P_1, \cdots, P_{b_1}), \cdots, B_l(P_{b_{l-1}+1}, \cdots, P_m)) \tag{4.35}$$

Since L divides F and it is a prime function, from Theorem 4.2 it must be

$$A_i = B_i(P_{b_{i-1}+1}, \cdots, P_{b_i}) \qquad i = 1, \cdots, l \tag{4.36}$$

where $\{A_1, \cdots, A_l\} = F/L$.

Notice that by definition of decomposition tree, all A_i appear in $DT(F)$.

For each function A_i, B_i is either a single-input function, or a multiple-input function. In the first case, $A_i = P_{b_i}$ (modulo complementation), and therefore P_{b_i} appears in $DT(F)$. In the second case, Eq. 4.36 indicates that A_i is decomposed by B_i. Since $|\mathcal{S}(A_i)| < |\mathcal{S}(F)|$, by induction each of $P_{b_{i-1}+1}, \cdots, P_{b_i}$ must appear in $DT(A_i)$, hence in $DT(F)$.

Consider now the case where L is an associative operator $\otimes$. In this case, one can write

$$M = G_1(y_1, \cdots, y_{m_1}) \otimes G_2(y_{m_1+1}, \cdots, y_m) \tag{4.37}$$

for suitable functions G_1, G_2. By substituting the formals y_i with the actuals P_i in Eq. 4.37 , and taking into account Theorem 4.5,

$$F = G_1(P_1, \cdots, P_{m_1}) \otimes G_2(P_{m_1+1}, \cdots, P_m) = A_1 \otimes A_2 \otimes \cdots \otimes A_k \tag{4.38}$$

where $\{A_1, \cdots, A_k\} = F/K_F$. We now focus on G_1, the same reasoning being then applicable to G_2. Equation 4.38 is the subject of Corollary 4.6: Either G_1 coincides with some A_i (in which case it appears explicitly in $DT(F)$), or it is expressible as the sum of some of the A_i. In this second case, we have

$$G_1(P_1, \cdots, P_{m_1}) = A_1 \otimes \cdots \otimes A_{k_1} \quad 2 \leq k_1 < k. \tag{4.39}$$

Let F_2 denote the function $A_1 \otimes \cdots \otimes A_{k_1}$. Notice that F_2 appears implicitly in $DT(F)$. Hence, every function appearing in $DT(F_2)$ will appear in $DT(F)$. By the inductive assumption, for every function M dividing F_2, F_2/M appears in $DT(F_2)$, hence in $DT(F)$. Eq. 4.39 states precisely that G_1 divides F_2, thus $P_1, \cdots, P_m$ appear in $DT(F_2)$ and consequently in $DT(F)$.

The second statement of the theorem can be trivially proved by building the function M corresponding to the decomposition tree obtained from $DT(F)$ by substituting a distinct variable y_i for each node A_i of the cut $\mathcal{C}$. □

Example 4.10. *Consider the function F of Example 4.7. The function* $MAJORITY(x_1 \oplus x_2, x_3, x_4) = (x_3 + x_4)(x_1 \oplus x_2) + x_3x_4$ *is a divisor of F and the cut* $\mathcal{C}$ *of* $DT(F)$ *with reference to the Example is* $\mathcal{C} = \{a, b, H, I\}$.

4.5 On the decomposability of Boolean functions

Shannon proved in [61] that for a sufficiently large support size, $|\mathcal{S}()|$, almost all Boolean functions require an exponential number of elements for their representation. He also showed in the same paper that the fraction of all functions of a given size support, $|\mathcal{S}()|$, that are decomposable, approaches 0 as $|\mathcal{S}()|$ approaches infinity. Sasao provided some quantitative results on how fast this limit is approached in [59]. He reports there that at $|\mathcal{S}()| = 5$, the percentage of functions that are undecomposable is already 99.9%. However, common experience indicates that most functions representing the functionality of digital systems can be represented by logic networks with much less than an exponential number of elements. The reason lies in the fact that most functions used in digital designs are not randomly picked at all, but instead are usually the results of the designers' natural choice of building complex systems by "adding" together simpler components.

The next chapter presents a new algorithm to expose the disjoint support decomposition components of a Boolean function. Because of its scalability, the algorithm has made possible to compute the disjoint decomposition of many complex functions, showing that most functions used in

industrial testbenches are in fact decomposable. It is natural to think that functions that can be represented by a sub-exponential network are more prone to be decomposable because inputs are less intertwined together in their path to produce the output value.

Chapter 5

A novel algorithm for Disjoint Support Decompositions

This chapter introduces a new algorithm to expose the maximal disjoint support decomposition of a Boolean function. The most relevant aspect of this algorithm is that its complexity is only worst-case quadratic in the size of the BDD representation of a function. Previously known algorithms had complexity exponential in the size of the support of the function to be decomposed. It is a well known fact that there exists functions for which the BDD representation is exponential in the size of their support – for instance, the functions representing the outputs of an integer multiplier [18]: in such situations our algorithm does not present any mayor complexity benefit. However, most complex functions that arise in the design and verification of digital circuits have BDD representations that are sub-exponential, hence the widespread use BDDs. For all these complex functions the algorithm introduced here below is the first that can find a disjoint support decomposition in sub-exponential time. Moreover, our algorithm finds the maximal disjoint support decomposition, and consequently all the other decompositions, since they can be all derived from it, as we showed in the previous chapter; on the other hand, previous algorithms could only find one or a few decompositions for a function, not necessarily the maximal DSD.

The algorithm traverses the BDD representation of a function in a bottom up fashion. At each

node it constructs the decomposition of the function rooted at the node based on the type of decompositions of each of the two cofactors of the node. The algorithm works by identifying all the possible situations that may occur at a BDD node and constructing the proper decomposition. The central part of this chapter analyzes the cases that may arise and shows what the resulting decomposition should be for each of them. We present implementation details at the end of the chapter and results we found by decomposing Boolean functions derived from the functionality of industrial digital circuit testbenches.

5.1 Building the decomposition bottom-up

The algorithm to expose the maximal Disjoint Support Decomposition starts from a BDD representation of the function F – see Section 2.3.1 – and finds all its disjoint support components by traversing the BDD tree recursively in a bottom up fashion.

Since we are presenting a recursive approach, we assume to know the disjoint support decomposition of the two cofactors F_0, F_1 with respect to a variable z. This chapter describes how to build the decomposition tree $DT(F)$ from the decomposition of the cofactors, $DT(F_0)$ and $DT(F_1)$.

In principle, one could build $DT(F)$ by running a case analysis based on the decomposition type of F_0, F_1. Example 5.1 below, however, indicates that this information alone may be not enough, and additional comparisons need be carried out on $DT(F_0), DT(F_1)$:

Example 5.1. *Let G, H, J denote three functions, with pairwise disjoint supports. Suppose they all have a PRIME kernel. Suppose also that the two cofactors of F w.r.t. z are as follows:*

$$F_0 = G$$

$$F_1 = G + H$$

That is, F_0 has a PRIME decomposition, while F_1 has an OR decomposition. The decomposition for

F can be found as follows:

$$F = \overline{x}G + xG + xH = G + xH = OR(xH, G)$$

and K_F is an OR function.

Consider now the case where F_1 is as above, while $F_0 = J$. Again we have a situation where F_0 and F_1 decompose through a PRIME and an OR function, respectively. However the decomposition of F results as:

$$F = \overline{x}J + xG + xH = MUX(x, G + H, J)$$

and K_F is a PRIME function.

Thus, functions with different decomposition types can have cofactors whose decomposition types are identical.

In practice, in order to build the decomposition, it is necessary to take the specific actual lists of F_0 and F_1 into consideration. The resulting analysis involves additional comparisons on the actual lists that are often numerous and complex. Therefore, we present here a different solution, based on the following observation:

Example 5.2. *Suppose that F has a decomposition with K_F a PRIME function:*

$$F = K_F(A_1(z), A_2, \cdots, A_l). \tag{5.1}$$

The two cofactors will then have decomposition

$$\begin{aligned} F_0 &= K_F(A_1(z=0), A_2, \cdots, A_l) \\ F_1 &= K_F(A_1(z=1), A_2, \cdots, A_l). \end{aligned} \tag{5.2}$$

If neither $A_1(z=0)$ nor $A_1(z=1)$ is a constant, the kernels of F_0 and F_1 coincide, and the two actuals lists differ in exactly one element. It will be shown below in Section 5.2 that also the inverse is true: if F_0 and F_1 have the same prime kernel and very similar *actuals lists, then F will have the same kernel as F_0 and the actuals list can be readily constructed. A similar observation holds also if F has OR, AND, or XOR decomposition.*

Example 5.2 suggests that we may subdivide the problem by distinguishing the case where both $A_1(z=0)$ and $A_1(z=1)$ are constants, from the case when at most one of them is constant. In fact, the former will requires a simpler analysis to identify the decomposition of the function F. For both the two cofactors of A_1 to be constant, A_1 must have a single variable in its support and it must be $A_1 = z$ or $A_1 = \overline{z}$. We refer to this situation as a *new decomposition*, since in this case we are starting a new decomposition block that contains the single variable at the top of our bottom-up decomposition construction. We refer to the other situation as an *inherited decomposition*, since in this case we are, generally speaking, expanding a block that exists already in the decomposition of the cofactors of F by "adding" the variable z to it.

Definition 5.1. *We say that the decomposition of F: $\langle K_F,\ F/K_F\rangle$ is* **inherited** *if $|\mathcal{S}(A_1)| \geq 2$. It is termed* **new** *otherwise.*

It will be shown that in an inherited decomposition, F shares the kernel (and some actuals) with at least one of its cofactors. In a new decomposition, this is not guaranteed to happen.

Let $A_{10} = A_1(z=0)$ and $A_{11} = A_1(z=1)$, respectively. We further classify *inherited decompositions* as follows:

1. Neither A_{10} nor A_{11} is constant, $A_{10} \neq \overline{A_{11}}$, and
   (a) F has *PRIME* decomposition;
   (b) F has *AND*, *OR* , or *XOR* decomposition;
2. Exactly one of A_{10}, A_{11} is constant (*i.e.* A_1 is the *OR* or *AND* of z – or $\overline{z}$ – with a suitable function); and

(a) F has *PRIME* decomposition;

(b) F has *AND*, *OR*, *XOR* decomposition.

3. $A_{10} = \overline{A_{11}}$ and A_{10} is not a constant (*i.e.* A_1 is the *XOR* of z with a suitable function); and

(a) F has a *PRIME* decomposition;

(b) F has *AND* or *OR* decomposition.

Notice that since A_1 has a *XOR* decomposition, F cannot have a *XOR* decomposition.

Notice that, in the first type of inherited decompositions, A_1 is essentially an arbitrary function of three or more variables. A_1 may of course have a *XOR*, *OR*, or *AND* decomposition, we just exclude the situation where z (or $\overline{z}$) appears as an element of its actuals list. The three scenarios are mutually exclusive, and together they cover all the possibilities for inherited decompositions.

Given this classification of decomposition types, we proceed now as follows: Sections 5.2 to 5.4 cover all the three subtypes of inherited decompositions, Section 5.5 analyzes new decompositions. Each Section shows how to determine which scenario a Shannon decomposition belongs to, and how to construct $DT(F)$ from $DT(F_0)$ and $DT(F_1)$.

5.2 Case 1. Neither A_{10} nor A_{11} is constant and $A_{10} \neq \overline{A_{11}}$

This case was implicitly described in Example 5.2. We need to distinguish the two subcases where F is prime and where F is decomposed by an associative operator. The two subcases are addressed separately by the two Lemmas below:

Case 1.a - *PRIME* decomposition

Lemma 5.1. *A function F has a PRIME decomposition with arbitrary function $A_1(z, \cdots)$ in its actuals list if and only if :*

1. *F_0 and F_1 both have PRIME decompositions;*

2. *the actuals lists F_0/K_{F_0} and F_1/K_{F_1} have the same size, and they differ in exactly one element, called G and H, respectively;*

3. *either*

$$F_0(G=0) = F_1(H=0) \quad \textit{and} \quad F_0(G=1) = F_1(H=1) \tag{5.3}$$

or

$$F_0(G=0) = F_1(H=1) \quad \textit{and} \quad F_0(G=1) = F_1(H=0) \tag{5.4}$$

must hold.

Moreover, if Eq. 5.3 holds, then F/K_F is obtained from F_0/K_{F_0} by replacing G with $A_1 = \overline{z}G + zH$, else by replacing G with $A_1 = \overline{z}G + z\overline{H}$.

Notice that Lemma 5.1 does not require any explicit comparison between K_{F_0} and K_{F_1}. These comparisons are replaced by the comparison of generalized cofactors. We can thus avoid building explicit representations of K_{F_0}, K_{F_1}.

Proof. To prove the *only if* part, notice that Eq. 5.2 indicates that K_F divides F_0 and F_1. Since we assumed K_F to be *PRIME*, K_F will be NP-equivalent to K_{F_0}, K_{F_1}. All elements of F/K_F have positive BDD polarity, hence, $A_2, \cdots, A_l$ will appear with the same polarity in F_0/K_{F_0} and F_1/K_{F_1}. One or both of A_{10}, A_{11}, however, may have negative BDD polarity. Therefore, F_0/K_{F_0} will actually contain either A_{10} or $\overline{A_{10}}$. The same reasoning obviously applies to F_1/K_{F_1}. We indicate with G, H the functions actually appearing in F_0/K_{F_0} and F_1/K_{F_1}, respectively. To verify the third point, consider taking the generalized cofactors of F_0 and F_1 with respect to G and H. If A_{10} and A_{11} have the same polarity (say, positive), then $K_{F_0} = K_{F_1}$ and we have:

$$F_0(A_{10}=0) = K_{F_0}(G=0, A_2, \cdots, A_l) = K_{F_1}(H=0, A_2, \cdots, A_l) = F_1(A_{11}=0) \tag{5.5}$$

$$F_0(A_{10}=1) = K_{F_0}(G=1, A_2, \cdots, A_l) = F_1(A_{11}=1). \tag{5.6}$$

If A_{10} and A_{11} have opposite polarity (say, A_{10} has negative polarity), then

$$F_0(A_{10}=0) = K_{F_0}(G=0,A_2,\cdots,A_l) = K_{F_1}(H=1,A_2,\cdots,A_l) = F_1(A_{11}=1) \quad (5.7)$$

$$F_0(A_{10}=1) = K_{F_0}(G=1,A_2,\cdots,A_l) = F_1(A_{11}=0). \quad (5.8)$$

Hence, Eqs. 5.5 and 5.7 reduce to Eq. 5.3 and 5.4.

To prove the *if* part, recall that F_0 and F_1 both have PRIME decompositions and that their actuals list differ in exactly one element (G vs. H). The cofactors of F_0 and F_1 with respect to G and H are then well defined. Suppose first Eq. 5.3 holds:

$$F_1 = \overline{H}F_1(H=0) + HF_1(H=1) = \overline{H}F_0(G=0) + HF_0(G=1). \quad (5.9)$$

Using the decomposition of F_0 in Eq. 5.9 :

$$F_1 = \overline{H}K_{F_0}(G=0,A_2,\cdots,A_l) + HK_{F_0}(G=1,A_2,\cdots,A_l) = K_{F_0}(H,A_2,\cdots,A_l). \quad (5.10)$$

Eq. 5.10 indicates that K_{F_0} divides F_1 as well, hence F_0 and F_1 have the same decomposition type. From Eq. 5.3 it also follows that

$$F = \overline{z}F_0 + zF_1 = \overline{z}K_{F_0}(G,A_2,\cdots,A_l) + zK_{F_0}(H,A_2,\cdots,A_l) = K_{F_0}(\overline{z}G + zH,A_2,\cdots,A_l) \quad (5.11)$$

Eq. 5.11 indicates precisely that F has *PRIME* decomposition (its kernel being K_{F_0}), and that $F/K_F = \{\overline{z}G + zH, A_2, \cdots, A_l\}$, which is what we needed to prove.

The case where Eq. 5.4 holds can be handled in the same way, just by replacing H with $\overline{H}$. □

Example 5.3. *The function $F = azb + e\overline{z}b + cb \oplus d$ has kernel $K_F(x_1,x_2,x_3,x_4) = x_1x_2 + x_3x_2 \oplus x_4$ and actuals list $(az + e\overline{z}, b, c, d)$. By computing the cofactors w.r.t. z, we obtain F_0 and F_1 with kernel identical to K_F and actuals lists: $F_0/K_{F_0} = (e,b,c,d)$ and $F_1/K_{F_1} = (a,b,c,d)$, respectively. Since the two cofactors satisfy all the three conditions of Lemma 5.1, we can find the decomposition of F*

from their kernels and actuals lists.

Case 1.b - Associative decomposition

The case where F is decomposed by an associative operator is slightly more complex. Therefore, we first provide the intuition, and then prove formally a criterion for identifying such a case. Suppose F has a (say) *OR* decomposition:

$$F = OR_k(A_1(z), A_2, \cdots, A_k)$$

The two cofactors will also have *OR* decomposition:

$$F_0 = OR_k(A_{10}, A_2, \cdots, A_k) \quad \text{and} \quad F_1 = OR_k(A_{11}, A_2, \cdots, A_k)$$

Notice, however, that one or both of A_{10}, A_{11} may have a *OR* decomposition as well. Let

$$A_{10} = OR_l(B_1, B_2, \cdots, B_l) \quad \text{and} \quad A_{11} = OR_m(C_1, \cdots, C_m) \quad \text{where} \quad l, m \geq 1.$$

Therefore, $K_{F_0} = OR_{k-1+l}$, $F_0/K_{F_0} = \{B_1, \cdots, B_l, A_2, \cdots, A_k\}$ and $K_{F_1} = OR_{k-1+m}$, $F_1/K_{F_1} = \{C_1, \cdots, C_m, A_2, \cdots, A_k\}$. Notice that all the functions B_i must differ from all of the C_j, and that the two actuals lists still have at least one element in common $(A_2, \cdots, A_k)$. These observation are formalized in Lemma 5.2 below:

Lemma 5.2. *A function F has an OR decomposition with arbitrary function $A_1(z, \cdots)$ in its actuals list if and only if:*

1. *both F_0 and F_1 have OR decompositions;*
2. *the set of common actuals $\mathcal{A}_c = \{A_2, \cdots, A_k\}$ is not empty;*
3. *$F_0/K_{F_0} - \mathcal{A}_c \neq \emptyset$ and $F_1/K_{F_1} - \mathcal{A}_c \neq \emptyset$.*

Proof. The *only if* part of the proof follows immediately from the previous observations. For the *if* part, let $B_1, \cdots, B_l$ denote the functions in $F_0/K_{F_0} - \mathcal{A}_c$, and $C_1, \cdots, C_m$ those in $F_1/K_{F_1} - \mathcal{A}_c$. Then ,

$$F_0 = OR_{k-1+l}(B_1, \cdots, B_l, A_2, \cdots, A_k) \quad \text{and} \quad F_1 = OR_{k-1+m}(C_1, \cdots, C_m, A_2, \cdots, A_k)$$

Hence,

$$F = z'F_0 + zF_1 = OR_k(\overline{z}OR_l(B_1, \cdots, B_l) + zOR_m(C_1, \cdots, C_m), A_2, \cdots, A_k) \tag{5.12}$$

We need to show now that $\overline{z}OR_l(B_1, \cdots, B_l) + zOR_m(C_1, \cdots, C_m)$ does not have an *OR* decomposition. Suppose, by contradiction, that it has an *OR* decomposition. Then, some of the terms $(B_1, \cdots, B_l)$ would coincide with some of the $(C_1, \cdots, C_m)$, against our assumptions. Hence, Eq. 5.12 indicates that F has a OR_k decomposition. □

Identical results can be shown for the *AND* and *XOR* cases.

5.3 Case 2. Exactly one of A_{10}, A_{11} is constant

We now assume that exactly one of A_{10}, A_{11} is a constant. We consider only the case $A_{10} = 0$, so that effectively $A_1 = zA_{11}$. The other cases can be handled similarly. In this scenario we need to consider separately the case where F will have a *PRIME* decomposition, and the case where F will be decomposed by an associative operator.

Case 2.a - *PRIME* decomposition

In this case :

$$F = K_F(A_1, A_2, \cdots, A_l)$$

where K_F is a *PRIME* function. Recalling that $A_1 = zA_{11}$, the two cofactors are:

$$F_0 = K_F(0, A_2, \cdots, A_l) \quad \text{and} \quad F_1 = K_F(A_{11}, A_2, \cdots, A_l) \tag{5.13}$$

Eq. 5.13 indicates that:

1. K_F is also the kernel of F_1;

2. F_1/K_F differs from F/K_F in exactly one element (A_{11} va. A_1).

3. K_F **is not** the kernel of F_0.

Again, the following Lemma helps us avoid comparing kernels explicitly:

Lemma 5.3. *A function F has a PRIME decomposition with $A_1 = zG$ in its actuals list, for a suitable non-constant function G if and only if :*

1. *F_1 has a PRIME decomposition;*

2. *there exists a function $G \in F_1/K_{F_1}$ such that $F_1(G = 0) = F_0$.*

Proof. For the *only if* part, recall that Eq. 5.13 indicates that F_1 has the same kernel as F. The second point also follows immediately from Eq. 5.13, using $G = A_{11}$.

For the *if* part, notice that, since $F_0 = F_1(G = 0)$,

$$F = z'F_0 + zF_1 = z'K_{F_1}(0, A_2, \cdots, A_l) + zK_{F_1}(G, A_2, \cdots, A_l) = K_{F_1}(zG, A_2, \cdots, A_l)$$

indicating precisely that K_{F_1} is also the kernel of F , and that $A_1 = zG$. □

It is worth noticing that Lemma 5.3 does not indicate which function in F_1/K_{F_1} needs be chosen for the cofactoring. Indeed, all functions $A_i \in F_1/K_{F_1}$ such that $\mathcal{S}(A_i) \cap \mathcal{S}(F_0) = \emptyset$ are candidates.

Case 2.b - Associative decomposition

Since we assumed at the beginning of Section 5.3 that A_1 has an *AND* decomposition, zA_{11}, F can have only *OR* or *XOR* decomposition. We focus here on *OR* decompositions, the *XOR* case being conceptually identical.

Again, we need to consider the case where A_{11} itself may have an *OR* decomposition.

Let

$$A_{11} = OR_l(B_1, \cdots, B_l) \quad l \geq 1.$$

The case where A_{11} does not have a *OR* decomposition is implicitly addressed by $l = 1$. The decomposition of F can then be written as :

$$\begin{aligned} F &= OR_k(zA_{11}, A_2, \cdots, A_k) \quad k \geq 2 \\ F_0 &= OR_{k-1}(A_2, \cdots, A_k) \qquad (5.14) \\ F_1 &= OR_k(A_{11}, A_2, \cdots, A_k) = OR_{k+l-1}(B_1, \cdots, B_l, A_2, \cdots, A_k) \qquad (5.15) \end{aligned}$$

Equation 5.15 indicates that F_1 will also have an *OR* decomposition. F_0, however, may have a different decomposition: in fact, in the special case $k = 2$, Eq. 5.14 simplifies to $F_0 = A_2$ and A_2 does not have an *OR* decomposition by hypothesis. In the general case, all the actuals of F_0/OR_{k-1} will belong to F_1/OR_{k+l-1}. In the special case $k = 2$, F_0 itself will be an element of F_1/OR_{k+l-1}. These observations are formalized below:

Lemma 5.4. *A function F has an OR_k decomposition with $A_1 = zG$ in its actuals list, for a suitable non-constant function G if and only if:*

1. *F_1 has an OR_{k+l-1} decomposition with $k \geq 2$ and $l \geq 1$;*

2. *either $k > 2$ and F_0 has an OR_{k-1} decomposition and $F_0/K_{F_0} \subset F_1/K_{F_1}$; or $k = 2$ and $F_0 \in$*

F_1/K_{F_1}.

Proof. The *only if* part follows directly from the above observations. For the *if* part, suppose:

$$F_1 = OR_{k-1+l}(B_1, \cdots, B_l, A_2, \cdots, A_k)$$

and

$$F_0 = OR_{k-1}(A_2, \cdots, A_k).$$

Consequently, we have:

$$\begin{aligned} F &= \overline{z}F_0 + zF_1 = OR_2(OR_{k-1}(A_2, \cdots, A_k), zOR_l(G_1, \cdots, G_l)) \\ &= OR_k(zOR_l(G_1, \cdots, G_l), A_2, \cdots, A_k) \end{aligned}$$

which is what we needed to show. Notice that the algebra holds also for the corner case $k = 2$. □

5.4 Case 3. $A_{10} = \overline{A_{11}}$ and A_{10} is not a constant

In this scenario A_1 has *XOR* decomposition : $A_1 = z \oplus A_{10}$. It is not restrictive to assume that A_{10} has positive BDD polarity. Again, we need to address the case where F has a *PRIME* decomposition separately from the other cases.

Case 3.a - *PRIME* decomposition

If F has *PRIME* decomposition, then

$$F_0 = K_F(A_{10}, A_2, \cdots, A_l) \quad \text{and} \quad F_1 = K_F(\overline{A_{10}}, A_2, \cdots, A_l) \tag{5.16}$$

Again, K_{F_0} and K_{F_1} are NP-equivalent to K_F, hence, F_0 and F_1 have *PRIME* decompositions. Moreover, F_0/K_{F_0} and F_1/K_{F_1} are identical (because of the definition of normal Decomposition Tree - see

also Section 4.4.3). Another consequence of Eq. 5.16 is that:

$$F_0(A_{10}=0) = F_1(A_{10}=1) \qquad F_0(A_{10}=1) = F_1(A_{10}=0)$$

The following Lemma provides necessary and sufficient conditions for identifying this case:

Lemma 5.5. *A function F has a PRIME decomposition with $A_1 = z \oplus G$ in its actuals list, for a suitable non-constant function G if and only if:*

1. *F_0 and F_1 have PRIME decompositions;*
2. *$F_0/K_{F_0} = F_1/K_{F_1}$;*
3. *there exists a function H in F_0/K_{F_0} such that:*

$$F_0(H=0) = F_1(H=1) \quad \textit{and} \quad F_0(H=1) = F_1(H=0) \tag{5.17}$$

In this case, either $G = H$ or $G = \overline{H}$.

Proof. The *only if* part follows directly from the introduction to this case. For the *if* part, observe that if Eq. 5.17 holds, then:

$$\begin{aligned} F_1 &= \overline{H}F_0(H=1) + HF_0(H=0) \\ F &= \overline{z}F_0 + zF_1 = \overline{z}\overline{H}F_0(H=0) + \overline{z}HF_0(H=1) + z\overline{H}F_0(H=1) + zHF_0(H=0) \\ &= (\overline{z}\overline{H} + zH)F_0(H=0) + (z\overline{H} + \overline{z}H)F_0(H=1) = F_0(H = z \oplus H). \end{aligned}$$

Hence, F has the same kernel as F_0, and its actuals list coincides with that of F_0, except for one element, namely, H, which is being replaced by either $z \oplus H$ or by $\overline{(z \oplus H)}$, depending on the polarity of the BDD representation. □

Notice that Lemma 5.5 does not indicate which function of F_0/K_{F_0} needs to be *XOR*-ed with z. Unfortunately, there is no way of knowing other than checking each function until Eq. 5.17 is verified.

Case 3.b - Associative decomposition

The difference from Case 3.a lies again in the fact that the candidate H may have the same decomposition type (*AND*, *OR*) as F. The way to handle this difference has been described already in Sections 5.2 and 5.3 for the other cases. Therefore, we omit it from the present analysis.

5.5 New decompositions

We now consider the case where $A_1 = \overline{z}$ or $A_1 = z$. We need to distinguish three subcases, namely,

(a) F has an *AND* or *OR* decomposition;

(b) F has an *XOR* decomposition;

(c) F has a *PRIME* decomposition.

These cases will be handled separately in the three paragraphs below.

Case a - *AND* or *OR* decomposition

$F = OR(z,G)$, then the two cofactors are $F_{\overline{z}} = G$ and $F_z = 1$. Conversely, if $F_z = 1$, then $F = F_{\overline{z}}\overline{z} + 1z = OR(z, F_{\overline{z}})$. Hence the decomposition is inferred by verifying that one of F_z is the constant 1. Since $z \notin S(F_G)$, F has a *OR* decomposition with $z \in F/OR$. The second case can be treated similarly showing that F has an *OR* decomposition with $\overline{z} \in F/OR$ if and only if the cofactor $F_{\overline{z}}$ is the constant 1. The case of *AND* decomposition is symmetrical, with the constant 0 replacing the constant 1. In summary, a new *AND* or *OR* decomposition is discovered if one of the two cofactors F_0 or F_1 is a constant:

- $F_1 = 1 \rightarrow F = z + F_0$.
- $F_0 = 1 \rightarrow F = \overline{z} + F_1$.
- $F_0 = 0 \rightarrow F = z \cdot F_1 = \overline{\overline{z} + \overline{F_1}}$.
- $F_1 = 0 \rightarrow F = \overline{z} \cdot F_0 = \overline{z + \overline{F_0}}$.

Case b - *XOR* decomposition

If $F = XOR(z,G)$, then $F_{\overline{z}} = G, F_z = \overline{G}$, and conversely, if $F_z = \overline{F_{\overline{z}}}$, then F has XOR decomposition with $z \in F/XOR$. For this case, the decomposition is inferred by checking that $F_z = \overline{F_{\overline{z}}}$.

Case c - *PRIME* decomposition

This case is by far the most complex of all. There are no necessary and sufficient conditions for identifying this case : It is determined by failing to construct any other type of decomposition. As mentioned, we do not need to keep track of the particular *PRIME* function used in the decomposition. Therefore, the task at hand is just to identify the actuals list F/K_F. Unlike the previous cases, in order to build this list, we will need to compare not just the actuals lists F_0/K_{F_0}, F_1/K_{F_1}, but the entire trees. Fortunately, this comparison can still be carried out efficiently. The rest of this section contains the details of this construction and the theoretical justification.

Consider once again the Shannon decomposition of a function F with disjunctive decomposition $F = K_F(z,A_2,A_3,\cdots,A_l)$:

$$F_{\overline{z}} = K_F(0,A_2,A_3,\cdots) \qquad F_z = K_F(1,A_2,A_3,\cdots) \tag{5.18}$$

Let $L_{\overline{y_1}}(y_2,\ldots,y_m)$ and $L_{y_1}(y_2,\ldots,y_m)$ denote the functions $K_F(0,y_2,..,y_m)$ and $K_F(1,y_2,..,y_m)$, respectively. Eq. 5.18 can then be written as

$$F_{\overline{z}} = L_{\overline{y_1}}(A_2,A_3,\cdots) \qquad F_z = L_{y_1}(A_2,A_3,\cdots) \tag{5.19}$$

In general, $L_{\overline{y_1}}$ and L_{y_1} may be further decomposable. Moreover, they may depend on only a subset of $y_2,\cdots,y_m$. For this reason, in order to determine the decomposition of F, it is not sufficient to compare the actuals list of $F_{\overline{z}}, F_z$. However, from Eq. 5.19, $L_{\overline{y_1}}$ divides $F_{\overline{z}}$. From Lemma 4.7, the set of functions $\{A_2,A_3,\cdots\}$ forms a cut of $DT(F_{\overline{z}})$ and thus F/K_F also contains a cut of the same decomposition tree. Similar reasoning applies to F_z.

The definition of uniform-support is needed to identify which functions from the two decomposition trees of the cofactor we need to select as components of $DT(F)$:

Definition 5.2. *Given a function F and a variable $z \in \mathcal{S}(F)$, a function A appearing in $DT(F_{\overline{z}})$ or in $DT(F_z)$ is said to have* **uniform-support** *if it has positive polarity and exactly one of the following is true:*

1. $\mathcal{S}(A) \subseteq \mathcal{S}(F_{\overline{z}}) \cap \overline{\mathcal{S}(F_z)}$ *and A appears in $DT(F_{\overline{z}})$ only;*
2. $\mathcal{S}(A) \subseteq \mathcal{S}(F_{\overline{z}}) \cap \mathcal{S}(F_z)$ *and A appears in both $DT(F_{\overline{z}})$ and $DT(F_z)$;*
3. $\mathcal{S}(A) \subseteq \overline{\mathcal{S}(F_{\overline{z}})} \cap \mathcal{S}(F_z)$ *and A appears in $DT(F_z)$ only.*

A is also termed **maximal** *if for no other uniform-support function B appearing in $DT(F_{\overline{z}})$ or $DT(F_z)$, we have $\mathcal{S}(A) \subset \mathcal{S}(B)$.*

For a given pair of decomposition trees $DT(F_z)$, $DT(F_{\overline{z}})$, we denote by $Max(F_{\overline{z}}, F_z)$ the set of maximal uniform support functions. It is this set of functions, togheter with the top variable z, that we will use as the actuals list for the decomposition of F. Theorem 5.6 shows that this is the correct set of functions for F/K_F.

Example 5.4. *Consider the function F of Figure 5.1. The decomposition of the two cofactors F_0 and F_1 is shown by its normal Decomposition Tree (which includes signed edges to indicate complementation of the function rooted at the signed node). The set $Max(F_{\overline{z}}, F_z)$ for this function is $\{x_1 + x_2, x_3, x_4x_5, x_6\}$. Notice that $x_1 + x_2$ appears implicitly in $DT(F_0)$ by rule (2) of Definition 4.7, while it appears implicitly in $DT(F_1)$ by rule (3) of the same Definition since the first element of F_1/K_{F_1} is $A_1 = \overline{x_1 + x_2 + x_6}$.*

The first three elements of the maximal set satisfy condition 2 of the definition of uniform support, while the last element satisfies condition 1.

As we mentioned, the set $Max(F_{\overline{z}}, F_z)$ effectively represents the actuals list of F. This is stated by the following Theorem:

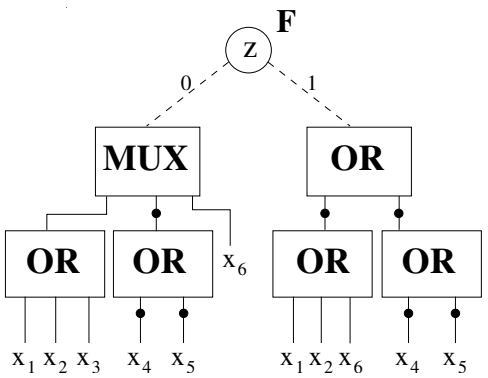

Figure 5.1: PRIME decomposition.

Theorem 5.6. *For a function F with decomposition $F/K_F = \{z, A_1, \cdots, A_l\}$, the actuals list is given by $\{z\} \cup Max(F_{\overline{z}}, F_z)$.*

We first illustrate the result with an example and then prove the Theorem.

Example 5.5. *Based on Theorem 5.6, the actuals list for the decomposition of the function in Figure 5.1 is given by $\{z, x_1 + x_2, x_3, x_4x_5, x_6\}$. The kernel function can then be easily derived by substituting the corresponding element of the formals list for each element of the actuals list. The formals list is $\{y_1, y_2, y_3, y_4, y_5\}$ and the kernel is $K_F = \overline{y_1}MUX(y_2 + y_3, y_4, y_5) + y_1((\overline{y_2 + y_5}) + y_4)$.*

The proof of Theorem 5.6 requires the proof of some properties of uniform-support functions.

Lemma 5.7. *Any two maximal uniform-support functions of $DT(F_{\overline{z}})$ or $DT(F_z)$ have disjoint support.*

Proof. We prove the Lemma by contradiction by showing that if two uniform-support functions A_1,A_2 share support variables, then at least one of them is not maximal. Notice, first of all, that there must be at least one decomposition tree where both functions appear. In fact, if one function only appeared in $DT(F_{\overline{z}})$ and the other only appeared in $DT(F_z)$, then, by definition of uniform-support, they would also be disjoint support. For sake of simplicity, we assume that both functions appear in $DT(F_z)$.

We need now to distinguish a few cases.

- Both A_1 and A_2 appear in $DT(F_z)$ *explicitly*. It is easy to see that, in order for the supports to overlap, either A_1 appears as a node in the subtree $DT(A_2)$, or A_2 appears as a node in the subtree $DT(A_1)$. In the first case, $\mathcal{S}(A_1) \subset \mathcal{S}(A_2)$, while in the second case $\mathcal{S}(A_2) \subset \mathcal{S}(A_1)$. In either case, one of the two functions is not maximal, as we intended to show.

- One of the two functions (say, A_1) appears only *implicitly*, while A_2 appears *explicitly*. Then it must be:

$$A_1 = \otimes_{i=1}^{k} B_i \qquad k \geq 2 \tag{5.20}$$

where $\otimes$ is one of *AND*, *OR*, *XOR* , and B_i are disjoint-support functions. Moreover, there is a function Q_1 appearing explicitly in $DT(F_z)$ such that

$$Q_1 = \otimes_{i=1}^{m} B_i \quad 2 \leq k < m \tag{5.21}$$

Notice that Q_1 does not need to be uniform-support. A_2 shares support variables with A_1, thus, either A_2 appears explicitly in $DT(Q_1)$ or Q_1 appears explicitly in $DT(A_2)$. If Q_1 appears explicitly in $DT(A_2)$ or if $A_2 = Q_1$, however, $\mathcal{S}(A_2) \supset \mathcal{S}(A_1)$, and A_1 is not maximal.

A_2 must then appear explicitly in $DT(Q_1)$, *i.e.*in exactly one of $DT(B_i)$, $i = 1 \ldots m$. But if A_2 appears explicitly in any $DT(B_i), i = 1, \cdots, k$, then $\mathcal{S}(A_2) \subset \mathcal{S}(A_1)$ and A_2 is still not maximal. Finally, if A_2 appears explicitly in any $DT(B_i), i = k+1, \cdots, m$, then $\mathcal{S}(A_2) \cap \mathcal{S}(A_1) = \emptyset$, against the hypothesis.

- Finally, suppose that both A_1 and A_2 appear *implicitly*. Then there must be an associative operator $\oslash =$ *AND*, *OR* or *XOR* such that

$$A_2 = \oslash_{i=1}^{l} C_i. \tag{5.22}$$

Moreover, there must be a function Q_2 appearing explicitly in $DT(F_z)$ such that

$$Q_2 = \oslash_{i=1}^{n} C_i \quad 2 \leq l < n. \tag{5.23}$$

As both Q_1 and Q_2 appear explicitly in $DT(F_z)$, exactly one of the following must hold :

1. $\mathcal{S}(Q_1) \cap \mathcal{S}(Q_2) = \emptyset$. But then $\mathcal{S}(A_1) \subseteq \mathcal{S}(Q_1) \cap \mathcal{S}(Q_2) \supseteq \mathcal{S}(A_2) = \emptyset$, against the hypothesis.

2. Q_1 appears in $DT(C_i)$ for one of the functions $C_i, i \leq l$. But, from Eq. 5.22, $\mathcal{S}(A_1) \subseteq \mathcal{S}(C_i) \subseteq \mathcal{S}(A_2)$, and again one of the functions (A_1) is not maximal.

3. Q_1 appears in $DT(C_i)$ for some $C_i, l < i \leq n$. This case is also impossible since it would be $\mathcal{S}(A_1) \subseteq \mathcal{S}(C_i) \cap \mathcal{S}(A_2) = \emptyset$.

4. $Q_1 = Q_2$. Then, the operator $\otimes$ of Eq. 5.20 must coincide with $\oslash$, and the functions C_i in Eq. 5.23 must coincide with the functions B_i in Eq. 5.21. Hence, A_2 can be written as

$$A_2 = \otimes_{i=j}^{l} B_i \quad \text{for} 1 \leq j \leq k < l \leq m. \tag{5.24}$$

Consider then the function

$$U = \otimes_{i=1}^{l} B_i. \tag{5.25}$$

U contains all the functions in the decomposition of $A_1/\otimes$ and of $A_2/\otimes$. Hence, U has uniform support, and $\mathcal{S}(U) \supset \mathcal{S}(A_1), \mathcal{S}(U) \supset \mathcal{S}(A_2)$, showing again that at least one of A_1, A_2 is not maximal.

In summary, in all cases, the assumption that A_1, A_2 share variables leads to the conclusion that at least one of them is not maximal, as we intended to prove. □

Lemma 5.8. *The set $Max(F_{\overline{z}}, F_z)$ contains a cut of $DT(F_{\overline{z}})$ and of $DT(F_z)$.*

Proof. We only prove that $Max(DT(F_{\overline{z}}), DT(F_z))$ contains a cut of $DT(F_z)$, the second part being entirely symmetrical.

Consider the collection C of functions $A_i \in Max(DT(F_{\overline{z}}), DT(F_z))$ such that $S(A_i) \cap S(F_z) \neq \emptyset$. From the definition of uniform support, for each such function, $S(A_i) \subseteq S(F_z)$ and they appear in $DT(F_z)$. From Lemma 5.7, they are disjoint-support. Therefore,

$$\bigcup_{A_i \in C} S(A_i) \subseteq S(F_z). \tag{5.26}$$

It remains to be shown that the containment relation 5.26 is actually an equality. To this regard, notice that for each variable $x_i \in S(F_z)$, the function x_i is trivially uniform-support. Either it is maximal, or there exist a maximal uniform-support function X_i appearing in $DT(F_z)$ whose support contains x_i. This function must then belong to $Max(DT(F_{\overline{z}}), DT(F_z))$ and therefore x_i must belong to the left-hand side of Eq. 5.26. This completes the proof. □

We define now a *bi-cut* as a set of uniform-support functions that provides a cut for the cofactors' decomposition trees:

Definition 5.3. *Given a function F and a variable $z \in \mathcal{S}(F)$, a collection of uniform support functions (not necessarily maximal) $\mathcal{C}_2 = \{A_i\}$ is termed a* **bi-cut** *if the following holds:*

1. $\mathcal{S}(A_i) \cap \mathcal{S}(A_j) = \emptyset \quad$ *for* $i \neq j$;
2. $\mathcal{C}_2$ *contains a cut of* $DT(F_{\overline{z}})$ *and of* $DT(F_z)$.

Example 5.6. *Consider a function F such that $F_{\overline{z}} = (x_1 + x_2)x_4$ and $F_z = (x_1 + x_2 + x_3)x_5$ as in Figure 5.2 (we present a non-normal decomposition tree for improved readability). A possible bi-cut for such function is $\mathcal{C}_2 = \{x_1 + x_2, x_3, x_4, x_5\}$. Note that the set $\mathcal{C} = \{x_1 + x_2 + x_3, x_4, x_5\}$ is not a bi-cut since it does not contain a cut of $DT(F_{\overline{z}})$.*

From Lemma 5.8, $Max(DT(F_{\overline{z}}), DT(F_z))$ is a bi-cut. It is also straightforward to verify that $Max(DT(F_{\overline{z}}), DT(F_z))$ has minimum size among bi-cuts. We now show that bi-cuts have a one-to-one correspondence to decompositions. These facts will be enough to prove Theorem 5.6.

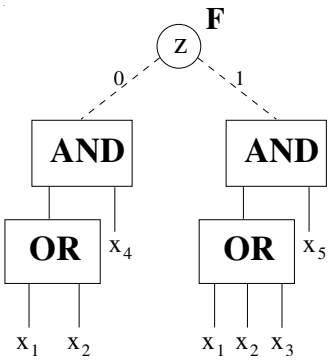

Figure 5.2: Function for Example 5.6.

Lemma 5.9. *Let M denote any function dividing F, such that $F/M = \{z, A_2, \cdots, A_m\}$. Then, the subset $\mathcal{C}_2 = \{A_2, \cdots, A_m\}$ is a bi-cut of F w.r.t. z. Conversely, for each bi-cut $\mathcal{C}_2$ there exists a function M such that $F/M = \{z\} \cup \mathcal{C}_2$.*

Proof. Eq. 5.19 shows that $\mathcal{C}_2$ contains a cut of $DT(F_{\overline{z}})$ and of $DT(F_z)$. The functions A_i are all disjoint-support, and each of them appears in at least one of $DT(F_{\overline{z}}), DT(F_z)$ (or else F would be independent from the variables in $\mathcal{S}(A_i)$). We also need to show, however, that each A_i has uniform support. To this end, suppose, for the sake of contradiction, that the support of one of the functions (say, $\mathcal{S}(A_2)$) is not uniform. It is not restrictive to assume that A_2 appears in $DT(F_{\overline{z}})$. Then $\mathcal{S}(A_2) \subseteq \mathcal{S}(F_{\overline{z}})$. Since we take A_2 to be not uniform, it must be

$$\mathcal{S}(A_2) \cap \mathcal{S}(F_z) \neq \emptyset \tag{5.27}$$

otherwise A_2 would be uniform by condition 1 of the definition of uniform-support; and

$$\mathcal{S}(A_2) \cap \overline{\mathcal{S}(F_z)} \neq \emptyset \tag{5.28}$$

otherwise A_2 would be uniform by condition 2. Let $\mathcal{C}$ indicate a subset of $\mathcal{C}_2$ forming a cut of

$DT(F_z)$; it follows that

$$\mathcal{S}(\mathcal{C}) = \bigcup_{A_i \in \mathcal{C}} \mathcal{S}(A_i) = \mathcal{S}(F_z); \tag{5.29}$$

The last equality being valid by definition of cut.

From Eq. 5.28, if $A_2 \in \mathcal{C}$, then $\mathcal{S}(\mathcal{C}) \cap \overline{\mathcal{S}(F_z)} \neq \emptyset$, contradicting Eq. 5.29. Hence, A_2 cannot belong to $\mathcal{C}$. We now show that A_2 cannot be left out of $\mathcal{C}$ either: From Eq. 5.27, there is a variable x in $\mathcal{S}(A_2) \cap \mathcal{S}(F_z)$. Since all functions of $\mathcal{C}_2$ are disjoint-support, x cannot be in the support of any other function of the bi-cut $\mathcal{C}_2$. Hence, if A_2 is left out of $\mathcal{C}$, $x \notin \mathcal{S}(\mathcal{C})$ and $\mathcal{C}$ is not a cut of $DT(F_z)$. In summary, A_2 could not be in a cut of $DT(F_z)$, but it could not be left out, a contradiction. Hence, A_2 must have uniform support and $\mathcal{C}_2$ is a bi-cut of F w.r.t. z.

We now show that for any given bi-cut $\mathcal{C}_2$ we can construct a decomposition of F. Consider the subset $\mathcal{C}_0 = \{A_2, \cdots, A_{c_0}\}$ of $\mathcal{C}_2$ forming a cut of $F_{\overline{z}}$. From Lemma 4.7, there exists a function $L_0(y_2, \cdots, y_{c_0})$ such that $F_{\overline{z}}/L_0 = \mathcal{C}_0$. Let also $\mathcal{C}_1 = \{A_{c_1}, \cdots, A_m\}$ denote the subset of $\mathcal{C}_2$ forming a cut of $DT(F_z)$. There exists then a function $L_1(y_{c_1}, \cdots, y_m)$ such that $F_z/L_1 = \mathcal{C}_1$. It is then easy to verify that the function $L(y_1, \cdots, y_m) = \overline{y_1} L_0(y_2, \cdots, y_{c_0}) + y_1 L_1(y_{c_1}, \cdots, y_m)$ satisfies

$$L(z, A_2, \cdots, A_m) = F, \tag{5.30}$$

that is, we have constructed a decomposition of F from $\mathcal{C}_2$. □

Finally, the proof of Theorem 5.6 follows:

Proof. - Theorem (5.6) - From Lemma 5.9, a function L can be found such that $F/L = \{z\} \cup Max(F_{\overline{z}}, F_z)$. Then, from Theorem 4.2, F/K_F cannot contain more elements than $\{z\} \cup Max(F_{\overline{z}}, F_z)$. Since $Max(F_{\overline{z}}, F_z)$ is a bi-cut of minimum size, F/K_F cannot contain fewer elements either, and consequently F/K_F and F/L must have the same size. In this case, however, from Theorems 4.2 and 4.4, K_F must be NP-equivalent to L and F/K_F must coincide with $\{z\} \cup Max(F_{\overline{z}}, F_z)$, modulo NP-equivalence. □

5.6 Putting it all together: The DSD procedure

We detail now the decomposition procedure. This description sets the stage for the complexity analysis presented in Section 5.7.

The algorithm traverses the nodes of the BDD of F in a bottom-up fashion. During the sweep, each node is inspected, and the decomposition tree of the function rooted at this node is determined from the decomposition of its cofactors and the top variable using the results presented earlier in this chapter.

The BDD node is then labeled with a *signed* – see Section 4.4.3 – pointer (`DEC *`) to the root of its decomposition tree.

```
void decompose_node(BDD* node) {
   node = NodeRegular(node);
   if (node->dec != NULL) return;
   var z = node->topVar;
   BDD *cof0 = node->cofactor0;
   BDD *cof1 = node->cofactor1;
   decompose_node(cof0);
   decompose_node(cof1);
   DEC *dec0 = GetDecomposition(cof0);
   DEC *dec1 = GetDecomposition(cof1);
   DEC *res = decompose(z,dec0, dec1);
   node->dec = res;
   return
}
```

The function `GetDecomposition` simply extracts the `DEC` pointer from a BDD node, and complements it if the BDD node was complemented. The call to `decompose` is the decomposition procedure proper:

```
DEC *decompose(var z, DEC* dec0, DEC* dec1) {
   res = decompose_INHERITED(z, dec0, dec1); // cases 1 2 3
   if (res) return(res);
   res = decompose_NEW(z, dec0, dec1);
   return(res);
}
```

We attempt the decomposition as an inherited or new decomposition. Each subroutine then considers all the corresponding cases from the previous section.

A `DEC` node contains a `.type` field and an `.actuals` list. The type field has four possible values: `VAR` (for simple variables), `OR`, `XOR` and `PRIME`; and it represents the decomposition type of the function rooted at that node. The actuals list is a list of signed pointers to BDD nodes. Each pointer represents a function in F/K_F.

It is worth noting that `decompose_INHERITED`, `decompose_NEW` are just switches, activating other procedures. In addition, since we must succeed with at at least one type of decomposition, the return value of `decompose` is guaranteed to be non-null. Finally, when two or more cases require a similar analysis, we group them in the same procedure so that portion of the computation can be shared; this is especially exploited in building inherited decompositions:

```
DEC* decompose_INHERITED(var z, DEC* dec0, DEC* dec1) {
   // case 1.b 2.b 3.b for AND/OR dec.
   res = decompose_INHERITED_OR_123.b(z, dec0, dec1);
   if (res) return(res);
   // case 1.b 2.b for XOR dec.
   res = decompose_INHERITED_XOR_12.b(z, dec0, dec1);
   if (res) return(res);
   //case 1.a 2.a 3.a
   res = decompose_INHERITED_PRIME_1.a(z, dec0, dec1);
```

```
    if (res) return(res);
    res = decompose_INHERITED_PRIME_2.a(z, dec0, dec1);
    if (res) return(res);
    res = decompose_INHERITED_PRIME_3.a(z, dec0, dec1);
    return(res);
}

DEC* decompose_NEW(var z, DEC* dec0, DEC* dec1) {
    res = decompose_NEW_OR(z, dec0, dec1); //case 4.a
    if (res) return(res);
    res = decompose_NEW_XOR(z, dec0, dec1); //case 4.b
    if (res) return(res);
    res = decompose_NEW_PRIME(z, dec0, dec1); //case 4.c
    return(res);
}
```

Since the maximal decomposition is unique, the calling order of the various subprocedures is irrelevant; with the following exception: since we only detect a new *PRIME* decomposition by failing all other cases, the procedure that builds a new *PRIME* decomposition, `decompose_NEW_PRIME`, must be kept last. In practice, we exploit this level of freedom by ordering the procedures based on the amount of analysis that they require, the fastest ones first; and disregarding even the grouping of new decompositions and inherited ones. For instance, Cases 4.a and 4.b are the fastest, and our implementation of `decompose_node` executes first of all `decompose_NEW_OR` and `decompose_NEW_XOR`.

In the reminder of this section, we will not discuss complement edges for BDD and DEC nodes any further. In particular, the segments of pseudo-code consider only nodes with positive polarity for simplicity, the extensions to include also nodes with negative polarity being straightforward.

We now analyze the subprocedures of `decompose` in detail.

5.6.1 Inherited decompositions

OR decompositions

`decompose_INHERITED_OR_123.b` groups the constructions described in Sections 5.2, 5.3 and 5.4 for identifying OR decompositions. For all of the three cases, we need to consider the actuals lists of the two cofactors and identify the common elements, which will be part of the resulting actuals list. To this list, we need to add a new element obtained by calling the second prototype of `decompose_node` with the node's top variable and the reminder *OR* decompositions as cofactors. Notice that this new element must be the first element of the resulting actuals list, based on the definition of normal Decomposition Tree from Section 4.4.3.

This procedure is successful as long as at least one of the two cofactor has an *OR* decomposition and there is at least one element in common between the actuals lists of F_0 and F_1. If, the actuals list of one cofactor is a proper subset of the other, then we have a Case 2.b decomposition. Otherwise we have a Case 1.b or 3.b decomposition.

Moreover, if one of the cofactors does not have a *OR* decomposition, for the purpose of this analysis, we consider its actuals list to have only one element, the cofactor function itself: Lemma 5.4 shows how to treat this situation in its special case of $k = 2$.

```
DEC* decompose_INHERITED_OR_123.b(var z, DEC* dec0, DEC* dec1) {
   DEC* res, dec0_residue, dec1_residue;
   list  common = list_intersect(dec0->actuals, dec1->actuals);
   if ( list_size(common) > 0 &&
        dec0->type == dec1->type == OR ) {
        dec0_residue = buildDecNode( OR,  dec0->actuals - common);
        if ( list_size(dec0_residue->actuals) == 0)
             dec0_residue = CONST_0;
        if ( list_size(dec0_residue->actuals) == 1)
             dec0_residue = getFirst(dec0_residue->actuals);
```

```
        // equivalently for right_residue
        G = decompose(z, dec0_residue, dec1_residue);
        res = buildDecNode(OR, { G, common}); //constructs node
        return res;
    }
    else if (list_intersect(dec0->actuals, dec1) ||
             list_intersect(dec1->actuals, dec0) )
        // build resulting decomposition
        // similar to above case
    }
    else return 0;
}
```

XOR decompositions

Inherited *XOR* decompositions can arise only from Cases 1.b and 2.b of Section 5.2.

Similarly to what has been discussed in the previous section, we need once again to check that at least one of the two cofactors is an *XOR* decomposition and that there is at least one element in common between the two actuals lists. The rest of the construction corresponds to the one for inherited *OR* decompositions.

PRIME decompositions

The first type of inherited *PRIME* decomposition is Case 1.a. The conditions for that case require that the two cofactors be both *PRIME* decompositions, the actuals lists differ in exactly one element and the cofactors w.r.t. those two elements match.

Example 5.7. *Consider again the function of Example 4.7 and assume that the top variable in its*

BDD representation was g. We consider available the decompositions of the cofactors w.r.t. g:

$$\begin{aligned} F_0 &= MAJORITY(G,H,i); \\ & G = a \oplus b; \\ & H = L + e; \\ & L = cd; \\ F_1 &= MAJORITY(G,H,N); \\ & N = ITE(f,h,i); \end{aligned}$$

Both F_0 and F_1 are decomposed by the same PRIME function MAJORITY. Their actuals lists are (G,H,h) and (G,H,N), respectively. They differ in exactly one element, namely, N instead of h.

We then check if Eq. 5.3 or 5.4 holds. This check can be carried out by computing $F_0(i=0)$, $F_0(i=1)$, $F_1(N=0)$, $F_1(N=1)$, and verifying that $F_0(i=0) = F_1(N=0)$, $F_0(i=1) = F_1(N=1)$. We then form a representation of the function $I = g'i + gMUX(f,h,i)$ and construct the decomposition of F as $MAJORITY(G,H,I)$. Note that, unless the decomposition of I is already known, we need to build that, too using the second prototype of `decompose_node`.

The following pseudocode checks if Eq. 5.3 or 5.4 hold. It returns the decomposition of F if the tests are successful:

```
DEC* decompose_INHERITED_PRIME_1.a (var z, DEC* dec0, DEC* dec1) {
   DEC* res;
   BDD* left_el, right_el, l0, r0;
   if (dec0->type != dec1->type != PRIME) return 0;
   if (list_size(dec0->actuals) != list_size(dec1->actuals))
    return 0;
   common = list_intersect(dec0->actuals, dec1->actuals);
   if (list_size(common) != size(dec0->actuals) -1) return 0;
```

```
    // the two functions differ in exactly one argument
    left_el = dec0->actuals - common;
    right_el = dec1->actuals - common;
    l0 = cofactor(dec0, left_el, 0);
    r0 = cofactor(dec1, right_el, 0);
    // compute also l1 and r1

    if ( ((l0 == r0) && (l1 == r1)) ||
         ((l1 == r0) && (l0 == r1))  ) {
       G = decompose (z, left_el->dec, right_el->dec);
       res = buildDecNode( PRIME, { G, common } );
       return res;
     }
     else return 0;
}
```

Case 2.a has a more complex set of comparisons. As the reader may recall from Section 5.3, Lemma 5.3 does not indicate precisely which is the function G to use to cofactor F_1. Instead we have a pool of candidates which are all the functions $A_i \in F_1/K_{F_1}$ such that $\mathcal{S}(A_i) \cap \mathcal{S}(F_0) = \emptyset$.

Thus we can detect such decomposition by considering the generalized cofactors (see Definition 2.3) of F_1 with respect to a subset of its actuals list elements and compare the result with F_0 to check if there is an element that satisfies the condition 2 of the Lemma.

It is important to note that each of these cofactor operations have complexity that it is only linear in the size of the BDD of F_1 (instead of quadratic). The reason for this simplified operation lies in the fact that the functions that we use in the cofactor operation are one in the decomposition of the other. To see this, consider a function $F = L(G,\ldots)$ and suppose we want to compute the cofactor w.r.t. $G = 1$. Then, $F_{G=1} = K_F(1,\ldots)$. To compute the last expression, we just need to consider any

combination of inputs of G such that $G = 1$, for instance a cube that satisfies G. We can then take the cofactor of F w.r.t. this cube to obtain our result, which is a linear time operation.

In general, we need to identify all the candidate $A_i \in F_1/K_{F_1}$ functions, and for each of those compute two generalized cofactors: $F_1(A_i = 0)$ and $F_1(A_i = 1)$ until we find a match. In the worst case, this entails the computation of $2 \times n$ cofactors, where n is the number of candidate elements.

Example 5.8. *Consider the functions $F_{\overline{z}} = ITE(A, CD, B + C), F_z = CD$. The actuals list of $F_{\overline{z}}$ contains A, B, C, D, of which only A and B are disjoint support from F_z.*

We observe that by assigning $B = 1$, however, $F_{\overline{z}} = A + CD \neq F_z$, and that assigning $B = 0$ results in $F_{\overline{z}} = C(A + D) \neq F_z$. The function B is then discarded. Assigning $A = 1$ instead results in $F_{\overline{z}} = ITE(1, CD, B + C) = CD = F_z$. A new function $Z = A + z$ is constructed, and F is decomposed as $ITE(Z, CD, B + C)$.

The following pseudocode reflects the observations above:

```
DEC* decompose_INHERITED_PRIME_2.a (var z, DEC* dec0, DEC* dec1) {
   DEC* res;
   BDD* l0,l1;
   tree_tag(dec1->actuals);
   // find the untagged elements in the left actuals list
   tryset = list_untagged(dec0->actuals);
   foreach(BDD* argument in tryset) {
      l1 = cofactor(dec0, argument, 1);
      l0 = cofactor(dec0, argument, 0);
      if ( l1 == dec1 ) {
         G = decompose (z, argument->dec, CONST_1);
         list actuals = dec0->actuals - argument + G;
         res = buildDecNode( PRIME, actuals );
         return res;
```

```
        } else if ( l0 == dec1 ) {
            // similar to above.
        }
    }
    // if unsuccessful, repeat by labeling the left tree
}
```

Case 3.a can be carried out analogously to case 1.a, with the difference that now instead of checking that the lists differ in exactly one element, we expect them to be identical. Once again the candidate function H with reference to Lemma 5.5 can be any of the actuals list elements.

5.6.2 New decompositions

OR and *XOR* decompositions

`decompose_NEW_OR` and `decompose_NEW_XOR` implement the checks of Sections 5.5 and 5.5. In the general case we create a new decomposition tree node of type `OR` or `XOR` and with an actuals list of length 2. However, note that it is possible that the non-constant cofactor has already a decomposition of the same type. If we detect this situation, the decomposition node will have an actuals list that is the same of its cofactor with the new element z prepended.

PRIME decompositions

In order to implement the construction of a new *PRIME* decomposition, we need to construct the set $Max(F_0, F_1)$ as shown in Theorem 5.6.

Construction of $Max(G,H)$

This operation allows us to find the set of maximal uniform support functions of two functions G, H whose decomposition is known. We show now how to construct a decomposition tree whose

root node has as its actuals list precisely the set of functions $Max(G,H)$. We call this tree also $Max(G,H)$.

Given two normal Decomposition Trees DT_G and DT_H, representing the decomposition of two functions G and H, respectively, the tree $Max(G,H)$ is the tree obtained as follows:

1. $Max(G,H)$ contains each node appearing in both DT_G and DT_H;

2. $Max(G,H)$ contains each arc appearing in both DT_G and DT_H;

3. if a node N of DT_G represents a function F_N, such that $\mathcal{S}(F_N) \cap \mathcal{S}(H) = \emptyset$, then the tree rooted at N belongs to $Max(G,H)$. Similarly for nodes of DT_H.

4. there is a node N labeled OR (XOR) for each pair of nodes $N_G \in DT_G, N_H \in DT_H$ labeled OR (XOR) and such that $\mathcal{S}(F_{N_1}) \cap \mathcal{S}(F_{N_2}) \neq \emptyset$. The actuals of N are the actuals common to N_G and N_H. The node N is suppressed if it has fewer than two actuals.

5. a root node is added. There is an arc from the root node to each node with no ancestors.

The construction above takes trivially time linear in the size of the two trees.

Example 5.9. *Figure 5.3 illustrates two decomposition trees DT_G and DT_H and the construction of $Max(G,H)$. In the graph we represent AND nodes as* `AND` *instead of complemented* `OR` *only for readability.*

The node `OR` *and node l belong to the intersection by rule 3. The tree rooted at* `PRIME` *by rules 1 and 2. The two nodes* `AND` *follow rule 4 producing the* `AND` *in the $Max(G,H)$ tree.*

To build a new *PRIME* decomposition, we simply need to build the $Max(F_0,F_1)$ tree and label the root node with type `PRIME`.

Example 5.10. *Consider the case $F_0 = ITE(abc, d+e+f, g \oplus h)$, $F_1 = ITE(ab, e+f+g, h \oplus c)$.*

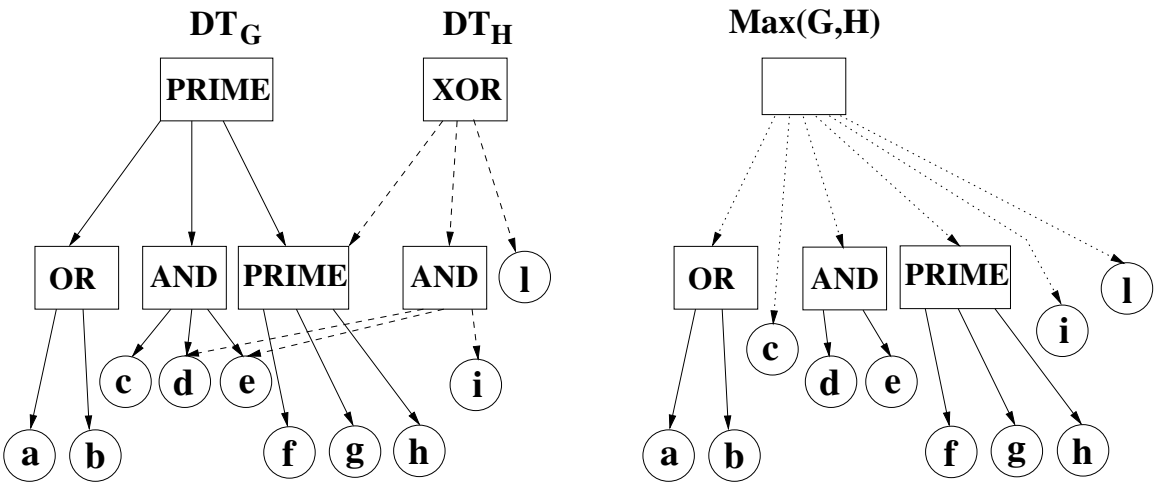

Figure 5.3: Two functions and the construction of their $Max(G,H)$ tree.

The set $Max(F_0,F_1)$ *is given by:*

$$\begin{aligned} A &= ab; \\ E &= e+f; \\ Max(F_0,F_1) &= \{A,E,c,d,g,h\}. \end{aligned}$$

Thus, the decomposition of $F = \overline{z}F_0 + zF_1$ *is given by* $F = K_F(z,A,E,c,d,g,h)$.

5.7 Complexity analysis and considerations

This section analyzes the complexity of the algorithm, given a function F whose BDD representation has **#BDD** nodes and whose support $|\mathcal{S}_F|$ has **#VAR** variables.

Notice, first of all, that the length of any actuals list in DT_F is bound by the number of variables in the support of the function, $|F/K_F| \leq |\mathcal{S}_F|$. We now analyze the complexity of each procedure in `decompose`.

The new decomposition procedures `decompose_NEW_OR` and `decompose_NEW_XOR` require only constant time operations: $O(k)$. `decompose_NEW_PRIME` requires only building the set

$Max(F_0, F_1)$. As pointed out previously, the complexity of this operation is linear in the size of the decomposition trees involved. The number of nodes in a decomposition tree is bound by **#VAR**; thus the complexity for this procedure is $O(\#\mathbf{VAR})$.

Inherited decomposition procedures involve recursive calls to `decompose`. The inherited procedures for *OR* and *XOR* decompositions require intersecting two actuals lists, operation linear in their length, and performing a recursive decomposition call. Note that, at each recursive call, the support of the function to decompose has at least one fewer variable, since the `common` portion of the final actuals list must have at least a support of size one. In conclusion, for these two procedures, we can write a recursive equation of their complexity: $O(|\mathcal{S}_F|) = O(\#\mathbf{VAR}) + O(|\mathcal{S}_F| - 1)$.

`decompose_INHERITED_PRIME_1.a` has a similar treatment, with two differences: 1) In addition of intersection the actuals lists, we need to compute also 4 generalized cofactors. As we showed in Section 5.6.1, these are special cofactors operations whose complexity is linear with the size of the BDDs involved. 2) At each recursive step, the support of the function to decompose now has at least two fewer variables, since we are dealing with `PRIME` nodes which have at least three inputs. The recursive operation for this procedure is thus: $O(|\mathcal{S}_F|) = O(\#\mathbf{VAR}) + 4 \cdot O(\#\mathbf{BDD}) + O(|\mathcal{S}_F| - 2)$.

`decompose_INHERITED_PRIME_2.a` and `decompose_INHERITED_PRIME_3.a` require a list intersection, a number of cofactors operations, up to twice the length of the actuals lists and a recursive call to `decompose`. However, in this case the call is guaranteed to be terminated by a new *OR* decomposition whose complexity, as we saw, is constant: $O(\#\mathbf{VAR}) + 2 \cdot O(\#\mathbf{BDD} \cdot \#\mathbf{VAR}) + O(k)$.

By solving the recursive equation of `decompose_INHERITED_PRIME_1.a`, we obtain a complexity of $O(\#\mathbf{BDD} \cdot \#\mathbf{VAR})$, which cannot be made worse even by terminating any of the recursive steps. with a `decompose_INHERITED_PRIME_3.a` call. Thus this is also the worst complexity of `decompose`.

Since we need to call this procedure for each BDD node in the representation of F, the overall complexity of our algorithm is: $O(\#\mathbf{BDD}^2 \cdot \#\mathbf{VAR})$.

Previously known algorithms – see Section 4.2 – had exponential complexity in the size of $\mathcal{S}_F$ and would compute only one of the many decompositions of a function. The complexity of our algorithm is dominated by the size of the BDD that represents the function F, not by the number of variables in its support. Moreover, it has the advantage of computing the finest granularity decomposition, from which all others can be derived.

For those functions whose BDD representation has size exponential in the number of the input variables, our algorithm has no better complexity than previously known ones. However, it is known that most functions representing digital circuit have corresponding BDDs whose size is much more compact and thus it is possible to build such BDDs even for some very large functions. Using our algorithm it is practically always possible to find the maximal disjunctive decomposition of a function, once a BDD has been built.

5.8 Experiments on the decomposability of industrial testbenches

The algorithm described in this chapter was implemented in a C++ program and tested on the circuits from the Logic Synthesis Benchmarks suite [68] and the ISCAS'89 Benchmark Circuits [16], including their 1993 additions. We report results on all the testbenches of the two suites. The testbenches are grouped by benchmark suite and by group within the suite: the Logic Synthesis suite includes two-level combinational circuits, multi-level combinational networks, sequential circuits and the tests added in '93. The ISCAS '89 suite includes a set of core sequential testbenches and additional circuits from '93. For all the sequential circuits, we considered only the combinational portion of the tests, we created an additional primary output for each latch input net and an additional primary input for each latch output. For each testbench, we first built the ROBDDs representing each output node as a function of the primary inputs, and then we attempted the decomposition of this functions.

The decomposition results are reported in 5.1. Next to the testbench's name we indicate how many of the output functions we could decompose: Output corresponds to the number of outputs of

the circuit, DEC reports how many of this output functions have a disjoint support decomposition. Output functions that are constant or a copy of a single input signal are considered decomposable. When not all of the outputs could be decomposed, we also looked at the non-decomposable outputs and checked if any of the two cofactors w.r.t. the top variable were decomposable. Column Dec Cof reports in how many cases at least one of the two cofactors resulted decomposable. We report a "-" in this column when all the outputs of the circuits were decomposable and thus the decomposability of the cofactors is not meaningful. By just glancing at the table, it's easy to notice that the column Dec Cof has a - for most of the circuits, meaning all of the outputs for that testbench are found to be decomposable.

Often, if a function is not decomposable, its cofactors are, and thus it is still possible to obtain a representation that has almost all the advantages and properties of disjoint support decompositions, except for a non-disjoint multiplexer corresponding to the node with the top variable of the specific BDD. Notice that even fairly big functions have a disjoint decomposition in most cases. For visual reference to the more complex testbenches, the table reports in boldface those circuits whose BDD construction, before starting the decompostion, required building more than 10,000 nodes.

The following two columns report the number of inputs of the circuit (Inputs) and the maximum number of inputs to any block in the decomposition tree of the output functions for that testbench (FanIn). It is worth noticing that in many cases, even the most complex, decomposition can reduce considerably the largest fanin to any block in the network's representation, while keeping each block disjoint support from the others. Then we indicated the total number of blocks in the normal decomposition trees. These latter two values are helpful in giving an indication of the amount of partitioning possible in the routing of the benchmark circuit.

The last four columns provide performance information. The first time/memory pair reports the time in seconds and the amount of memory in kilobytes required to produce the ROBDDs of all the output functions of a testbench. The second pair indicates the *additional* time and memory required to construct the normal decomposition trees from the ROBDDs. All the experiments were run on a Linux PC equipped with a Pentium 4 processor running at 2.7Ghz and 2GB of memory

and 512Kb of cache. In running the tests, we used a proprietary ROBDD package. In particular, our ROBDD package records the support of the functions associated to each ROBDD node. While this feature is convenient because of the number of support operations and tests we need to perform, its efficiency could be optimized. Moreover, our decomposition package has also room for implementation improvements.

Circuit	Outputs	DEC	Dec Cof	Inputs	FanIn	Blocks	BDD performance Time (s)	BDD performance Mem (KB)	DEC performance Time (s)	DEC performance Mem (KB)
Logic Synthesis '91 - Two level tests										
5xp1	10	9	0	7	7	20	0.00	13	0.00	1
9sym	1	0	0	9	9	1	0.00	59	0.00	1
alu4	8	1	0	14	14	10	0.04	417	0.01	22
apex1	45	43	0	45	30	224	0.02	373	0.02	37
apex2	3	3	-	39	29	16	0.17	1384	0.02	22
apex3	50	39	2	54	42	200	0.01	399	0.02	23
apex4	19	5	0	9	9	22	0.01	384	0.01	23
apex5	88	88	-	117	14	463	0.03	377	0.01	59
bw	28	15	4	5	5	57	0.00	11	0.00	3
clip	5	0	2	9	9	5	0.00	62	0.00	3
con1	2	0	2	7	6	2	0.00	2	0.00	0
duke2	29	24	3	22	17	91	0.00	108	0.00	13
e64	65	65	-	65	2	2080	0.01	83	0.00	5
misex1	7	1	0	8	7	8	0.00	4	0.00	1
misex2	18	17	1	25	7	105	0.00	10	0.00	3
misex3c	14	2	5	14	14	20	0.01	186	0.00	11
misex3	14	2	1	14	14	16	0.07	410	0.00	15
o64	1	1	-	130	2	129	0.00	85	0.00	8
rd53	3	1	1	5	5	6	0.00	7	0.00	0
rd73	3	1	0	7	7	8	0.00	41	0.00	1
rd84	4	2	0	8	8	16	0.01	84	0.00	1
sao2	4	4	-	10	8	12	0.00	41	0.00	2
seq	35	35	-	41	33	198	0.08	385	0.02	41
vg2	8	8	-	25	24	23	0.00	95	0.00	4
xor5	1	1	-	5	2	4	0.00	5	0.00	0
Logic Synthesis '91 - FSM tests										
daio	6	5	1	6	3	4	0.00	1	0.00	0
ex1	39	39	-	30	23	677	0.00	67	0.01	54
ex2	21	21	-	22	18	340	0.00	37	0.00	8
ex3	12	12	-	13	10	98	0.00	10	0.00	9
ex4	23	23	-	21	8	283	0.00	14	0.00	5
ex5	11	11	-	12	10	78	0.00	10	0.00	4
ex6	16	12	4	14	13	58	0.00	15	0.00	8
ex7	12	12	-	13	11	97	0.00	13	0.00	3
s1196	32	24	6	33	21	68	0.01	59	0.00	34
s1238	32	24	6	33	21	68	0.01	64	0.00	35
s1423	79	77	2	92	32	330	0.01	376	0.06	348

Table 5.1: Disjoint Support Decomposition results - *continued on next page*

Circuit	Outputs	DEC	Dec Cof	Inputs	FanIn	Blocks	BDD performance Time (s)	BDD performance Mem (KB)	DEC performance Time (s)	DEC performance Mem (KB)
s1488	25	23	2	15	12	59	0.01	77	0.00	11
s1494	25	23	2	15	12	59	0.01	77	0.00	11
s208	10	10	-	20	3	53	0.00	10	0.00	3
s27	4	4	-	8	2	11	0.00	1	0.00	0
s298	20	17	3	18	8	32	0.00	9	0.00	3
s344	26	23	0	25	7	37	0.00	11	0.00	3
s349	26	23	0	25	7	37	0.00	10	0.00	3
s382	27	27	-	25	7	70	0.00	11	0.00	5
s386	13	13	-	14	9	67	0.00	12	0.00	3
s400	27	27	-	25	7	70	0.00	11	0.00	4
s420	18	18	-	36	3	113	0.00	25	0.00	10
s444	27	27	-	25	7	70	0.00	25	0.00	6
s510	13	5	3	26	19	24	0.00	25	0.00	9
s526n	27	24	2	25	8	66	0.00	16	0.00	4
s526	27	24	3	25	8	66	0.00	15	0.00	4
s641	42	42	-	55	18	150	0.00	37	0.01	25
s713	42	42	-	55	18	150	0.00	42	0.01	28
s820	24	22	1	24	17	109	0.00	29	0.00	7
s832	24	22	1	24	17	109	0.00	30	0.00	7
s838	34	34	-	68	3	233	0.00	46	0.01	67
s953	52	47	2	46	17	97	0.01	54	0.00	13
Logic Synthesis '91 - Multi level tests										
9symml	1	0	0	9	9	1	0.00	37	0.00	1
alu2	6	4	0	10	10	8	0.00	78	0.00	5
alu4	8	4	0	14	14	14	0.01	199	0.00	11
apex6	99	99	-	135	14	369	0.00	77	0.00	24
apex7	37	37	-	49	9	155	0.00	44	0.00	13
b1	4	3	1	3	3	2	0.00	1	0.00	0
b9	21	21	-	41	8	54	0.00	15	0.00	4
C1355	32	0	0	41	41	32	0.23	1545	73.57	41689
C17	2	1	1	5	4	4	0.00	0	0.00	0
C1908	25	7	0	33	32	93	0.05	754	2.40	5787
C2670	140	119	1	233	78	187	0.05	666	1.36	8017
C3540	22	14	0	50	50	49	0.53	2301	2.05	16348
C432	7	1	1	36	36	23	0.01	329	0.07	489
C499	32	0	0	41	41	32	0.16	1406	100.61	40187
C5315	123	80	10	178	66	186	0.04	371	0.12	1032
C7552	108	107	1	207	118	295	0.18	1148	0.25	1899
C880	26	26	-	60	41	96	0.02	373	0.41	3374
c8	18	10	8	28	3	69	0.00	19	0.00	2
cc	20	20	-	21	4	32	0.00	7	0.00	2
cht	36	36	-	47	3	74	0.00	12	0.00	4
cm138a	8	8	-	6	2	40	0.00	1	0.00	1
cm150a	1	1	-	21	20	2	0.00	8	0.00	2
cm151a	2	2	-	12	11	2	0.00	3	0.00	1
cm152a	1	0	0	11	11	1	0.00	2	0.00	1
cm162a	5	5	-	14	4	19	0.00	5	0.00	1
cm163a	5	5	-	16	3	26	0.00	3	0.00	1

Table 5.1: Disjoint Support Decomposition results - *continued on next page*

Circuit	Outputs	DEC	Dec Cof	Inputs	FanIn	Blocks	BDD performance Time (s)	BDD performance Mem (KB)	DEC performance Time (s)	DEC performance Mem (KB)
cm42a	10	10	-	4	2	30	0.00	1	0.00	1
cm82a	3	3	-	5	3	6	0.00	2	0.00	1
cm85a	3	3	-	11	3	20	0.00	6	0.00	1
cmb	4	4	-	16	2	33	0.00	10	0.00	1
comp	3	3	-	32	3	63	0.00	24	0.00	19
count	16	16	-	35	3	168	0.00	7	0.00	2
cu	11	11	-	14	6	43	0.00	4	0.00	1
decod	16	16	-	5	2	64	0.00	2	0.00	1
des	245	245	-	256	14	560	0.07	373	0.02	77
example2	66	49	17	85	11	281	0.00	25	0.00	12
f51m	8	8	-	8	7	13	0.00	17	0.00	1
frg1	3	3	-	28	19	12	0.00	77	0.00	4
frg2	139	139	-	143	17	519	0.01	336	0.01	60
k2	45	43	2	45	30	224	0.01	353	0.02	38
lal	19	19	-	26	2	89	0.00	11	0.00	3
ldd	19	18	1	9	5	60	0.00	8	0.00	2
majority	1	1	-	5	4	2	0.00	1	0.00	0
mux	1	1	-	21	20	2	0.00	13	0.00	2
my_adder	17	17	-	33	3	48	0.00	67	0.00	9
pair	137	137	-	173	28	724	0.03	374	0.06	275
parity	1	1	-	16	2	15	0.00	5	0.00	1
pcler8	17	17	-	27	3	99	0.00	12	0.00	4
pcle	9	9	-	19	3	62	0.00	5	0.00	2
pm1	13	13	-	16	3	46	0.00	5	0.00	1
rot	107	104	3	135	42	351	0.04	621	0.25	2114
sct	15	14	1	19	3	63	0.00	11	0.00	2
tcon	16	8	8	17	3	8	0.00	2	0.00	1
term1	10	10	-	34	10	66	0.00	69	0.00	5
too_large	3	3	-	38	29	16	0.06	587	0.01	16
ttt2	21	18	2	24	8	66	0.00	31	0.00	3
unreg	16	16	-	36	3	48	0.00	7	0.00	3
vda	39	29	10	17	17	81	0.00	121	0.01	15
x1	35	35	-	51	17	181	0.01	192	0.00	16
x2	7	7	-	10	6	24	0.00	4	0.00	1
x3	99	99	-	135	14	369	0.01	121	0.00	22
x4	71	71	-	94	8	207	0.00	48	0.00	13
z4ml	4	4	-	7	3	9	0.00	14	0.00	1
Logic Synthesis '91 - Addition '93										
b12	9	8	0	15	8	31	0.01	58	0.00	2
bigkey	421	194	3	487	10	232	0.19	371	0.02	83
clma	115	115	-	416	36	532	24.25	373	0.01	38
cordic	2	2	-	23	8	18	0.09	349	0.00	3
cps	109	109	-	24	15	1147	0.03	210	0.01	66
dalu	16	15	1	75	31	227	0.12	373	0.02	33
dsip	421	194	3	453	12	232	0.18	372	0.03	85
ex4p	28	28	-	128	15	46	0.06	363	0.01	15
ex5p	63	54	2	8	8	271	0.04	195	0.00	8
i10	224	224	-	257	74	1098	0.64	3294	14.21	102333

Table 5.1: Disjoint Support Decomposition results - *continued on next page*

Circuit	Outputs	DEC	Dec Cof	Inputs	FanIn	Blocks	BDD performance Time (s)	BDD performance Mem (KB)	DEC performance Time (s)	DEC performance Mem (KB)
i1	13	13	-	25	3	43	0.00	4	0.00	2
i2	1	1	-	201	6	187	0.00	154	0.00	17
i3	6	6	-	132	2	126	0.00	28	0.00	8
i4	6	6	-	192	2	186	0.00	53	0.00	37
i5	66	66	-	133	2	132	0.01	19	0.00	7
i6	67	1	29	138	5	69	0.00	45	0.01	8
i7	67	3	64	199	6	72	0.01	64	0.00	12
i8	81	18	63	133	17	93	0.05	372	0.04	35
i9	63	0	0	88	13	63	0.01	93	0.01	58
mm4a	16	16	-	20	13	52	0.00	60	0.00	8
mm9a	36	36	-	40	28	117	0.03	386	0.26	2110
mm9b	35	35	-	39	29	293	0.03	384	0.35	1892
mult16a	17	17	-	34	3	64	0.00	61	0.00	7
mult16b	31	31	-	48	3	61	0.01	13	0.00	4
mult32a	33	33	-	66	3	128	0.04	381	0.01	105
s208	9	9	-	19	3	46	0.00	5	0.00	4
s38584	1730	1611	113	1465	36	5146	9.92	946	0.17	887
s5378	212	211	0	199	52	784	0.11	242	0.02	154
s838	33	33	-	67	3	562	0.01	33	0.00	65
s9234	174	169	5	172	40	509	0.10	280	0.01	59
sbc	83	83	-	68	21	404	0.02	110	0.01	44
sqrt8ml	4	4	-	8	5	11	0.00	11	0.00	1
sqrt8	4	4	-	8	7	7	0.00	11	0.00	1
squar5	8	4	2	5	5	14	0.00	8	0.00	1
t481	1	1	-	16	2	15	0.02	190	0.00	1
table3	14	0	2	14	14	14	0.01	176	0.02	152
table5	15	3	2	17	17	25	0.01	169	0.02	130
ISCAS '89 - FSM tests										
s1196	32	24	6	32	21	68	0.00	59	0.00	34
s1238	32	24	7	32	21	68	0.00	69	0.01	56
s13207.1	790	783	7	700	42	1805	1.01	371	0.03	97
s13207	790	783	7	700	42	1805	1.04	371	0.03	96
s1423	79	77	2	91	32	330	0.01	191	0.01	148
s1488	25	23	2	14	12	59	0.01	77	0.00	11
s1494	25	23	2	14	12	59	0.01	77	0.00	11
s15850.1	684	651	33	611	148	2074	1.9	1059	0.62	2606
s15850	684	651	33	611	148	2074	2.13	813	0.58	2349
s208	9	9	-	18	3	46	0.00	5	0.00	7
s27	4	4	-	7	2	11	0.00	1	0.00	0
s298	20	17	2	17	8	32	0.00	7	0.00	2
s344	26	23	0	24	7	37	0.00	10	0.00	3
s349	26	23	0	24	7	37	0.00	10	0.00	3
s35932	2048	2048	-	1763	6	3371	8.00	371	0.01	135
s382	27	27	-	24	7	70	0.00	10	0.00	3
s38584.1	1730	1611	113	1464	36	5146	12.53	801	0.16	959
s38584	1730	1611	113	1464	36	5146	11.32	824	0.14	912
s386	13	13	-	13	9	67	0.00	14	0.00	4
s400	27	27	-	24	7	70	0.00	11	0.00	5

Table 5.1: Disjoint Support Decomposition results - *continued on next page*

Circuit	Outputs	DEC	Dec Cof	Inputs	FanIn	Blocks	BDD performance Time (s)	BDD performance Mem (KB)	DEC performance Time (s)	DEC performance Mem (KB)
s420	17	17	-	34	3	154	0.00	16	0.00	11
s444	27	27	-	24	7	70	0.01	23	0.00	6
s510	13	5	5	25	19	24	0.00	23	0.00	4
s526n	27	24	2	24	8	66	0.01	15	0.00	4
s526	27	24	3	24	8	66	0.01	15	0.00	4
s5378	213	212	0	214	51	795	0.10	288	0.03	211
s641	42	42	-	54	18	150	0.00	54	0.01	55
s713	42	42	-	54	18	150	0.01	46	0.00	41
s820	24	22	1	23	17	109	0.00	30	0.00	7
s832	24	22	1	23	17	109	0.00	30	0.00	7
s838	33	33	-	66	3	562	0.00	37	0.00	100
s9234	250	245	4	247	48	707	0.42	372	0.05	250
s953	52	47	2	45	17	97	0.01	58	0.00	12
ISCAS '89 - Addition '93										
prolog	158	152	4	172	67	424	0.03	175	0.00	101
s1196	32	24	6	32	21	68	0.01	59	0.00	34
s1269	47	30	9	55	35	97	0.01	231	0.03	115
s1512	78	78	-	86	18	285	0.01	136	0.00	33
s3271	130	102	28	142	15	353	0.03	198	0.01	38
s3330	205	199	5	172	67	424	0.03	199	0.02	159
s3384	209	172	37	226	39	373	0.03	151	0.01	202
s344	26	23	0	24	7	37	0.00	10	0.00	3
s4863	88	66	2	153	22	190	1.96	4231	9.03	43400
s499	44	44	-	23	5	423	0.00	41	0.00	9
s635	33	33	-	34	2	591	0.00	16	0.00	4
s6669	269	194	44	322	16	380	0.92	2237	1.66	7848
s938	33	33	-	66	3	562	0.01	37	0.00	100
s967	52	47	2	45	17	97	0.01	59	0.00	11
s991	36	36	-	84	54	53	0.01	102	0.00	21

Table 5.1: Disjoint Support Decomposition results

In most cases the additional time to decompose a function is small compared to the time required to build the initial ROBDDs. However, there are a few cases where this is not the case: specifically *C1355* and *C499* of the Logic Synthesis suite cannot find a decomposition for any of the primary outputs, yet the algorithm is very time consuming. These circuits are very similar, they have the same number of inputs and outputs and they are both error correcting circuits as reported in [68]. By inspecting the two circuits we found that the intermediate nodes of these circuits up to about half way in the bottom-up construction were often decomposable; then the repetitive application of the algorithm `decompose_NEW_PRIME`, Section 5.6.2, made so that the top half of the construction produces almost invariably a PRIME decomposition with a kernel identical to the function itself.

Circuit *i10* from Logic Synthesis '91 - Addition '93 instead, requires times and memory resources above average because of the long actuals lists that are produced during the computation. Table 5.1 reports decomposition results for all the circuits in the test suites mentioned above with two exceptions: we could not apply the decomposition algorithm to circuit C6288 (a 16-bit multiplier) since we run out of memory building the initial ROBDDs for it; circuit s38417 runs out of memory during the decomposition because of its large support size and long intermediate actuals lists involved. We hope to be able to tackle this latter testbench with a more clever implementation of the decomposition algorithm. We summarized the results and found that we could decompose 16,472 functions out of a total of 18,584. The total time spent constructing the ROBDDs was 79.63s, while the time spent after that to attempt the functions' decompositions was 209.11s.

5.9 Conclusion

We presented in this chapter a novel algorithm that can generate the maximal disjoint support decomposition of a Boolean function represented by its BDD. The worst case complexity of this algorithm is only quadratic in the size of the BDD representation, while previously proposed algorithms has exponential complexity on the number of variables in the support of the function. We found it very fast in practice as we were able to obtain the decomposition of most testbenches in time comparable to the construction of their BDD. Experimental results indicate that the majority of functions representing the behavior of digital systems are indeed decomposable and the maximal disjoint decomposition has a fine granularity, as indicated by the support size of the biggest component block.

The next chapter will exploit these encouraging results in devising a new type of parameterization for symbolic simulation. This time the parameterization is exact, meaning that we generate a set of parameterized functions whose range matches exactly the frontier set of represented by the state vector.

Chapter 6

Exact Parameterizations for Symbolic Simulation

The theory of disjoint support decompositions provides important insights on the structure of a Boolean function and on the role and influence of each of its support variables. Moreover, the algorithm presented in the previous chapter allows us to take advantage of such insights very efficiently. We saw at the end of Chapter 3 that the parameterization technique of Cycle-Based Symbolic Simulation is computationally very efficient, but not exact. Often we may need to compromise by exploring only a subset of the possible set of states of the design under verification in order to maintain simulation efficiency. In some cases, this trade off produces simulation performance that is comparable to plain logic simulation in terms of vectors simulated per second.

Here we present a new parameterization technique that, by exploiting the disjoint decomposition properties of the functions in the state vector, can produce an exact parameterization, that is a new set of functions spanning the exact same state set as the original state vector. These new functions have smaller support than the original ones, and thus a simpler BDD representation.

In other words, if the parameterization of CBSS was building a function $\mathbf{PS}_{@\mathbf{k}}$ such that

$$\mathcal{R}(\mathbf{PS}_{@\mathbf{k}}) \subseteq \mathcal{R}(\mathbf{S}_{@\mathbf{k}}),$$

the algorithm presented in this chapter builds a parameterized vector function such that

$$\mathcal{R}(\mathbf{PS}_{@\mathbf{k}}) = \mathcal{R}(\mathbf{S}_{@\mathbf{k}}).$$

We call the new algorithm *DSD-based Symbolic Simulation*, (DSD-SS).

6.1 Re-encoding the state function

This new technique for improved scalability and robustness in symbolic simulation is similar to CBSS in that it inserts a parameterization phase in the feedback look of symbolic simulation as indicated in Figure 3.1. However, now we take a completely different approach to how we perform the parameterization.

In order to generate the parametric state vector for DSD-SS, at each step of simulation we start by generating the disjoint support decomposition representation for each of the component functions of the state vector. While each element of the vector has a tree decomposition with no reconvergence, as described in Chapter 4, it is now possible that two or more elements intersect at some intermediate node of their trees.

Figure 6.1 shows an example of a decomposed state vector for a small design with only four memory elements. The dashed line delimits the decomposition of component s_1 to show that each single component function is represented by a tree. We call this structure a **decomposition graph**.

The decomposed representation is generated dynamically during the simulation. We then use this representation to generate a parameterization of the state vector. The parameterization we propose is based on the observation that at each symbolic simulation step k, it is possible to substitute the state function $\mathbf{S}_{@\mathbf{k}} : \mathcal{B}^{mk} \rightarrow \mathcal{B}^{n}$ with a new function $\mathbf{PS}_{@\mathbf{k}}$ such that $\mathcal{R}(\mathbf{S}_{@\mathbf{k}}) = \mathcal{R}(\mathbf{PS}_{@\mathbf{k}})$ without affecting the results of the simulations; namely: 1) The set of outputs that can be generated by the circuit and 2) the set of states the circuit can reach at each cycle. If we can find a suitable function $\mathbf{PS}_{@\mathbf{k}}$ that also has a smaller BDD representation (*i.e.*, fewer BDD nodes), then we can control the size of the Boolean expression and improve the performance of symbolic simulation.

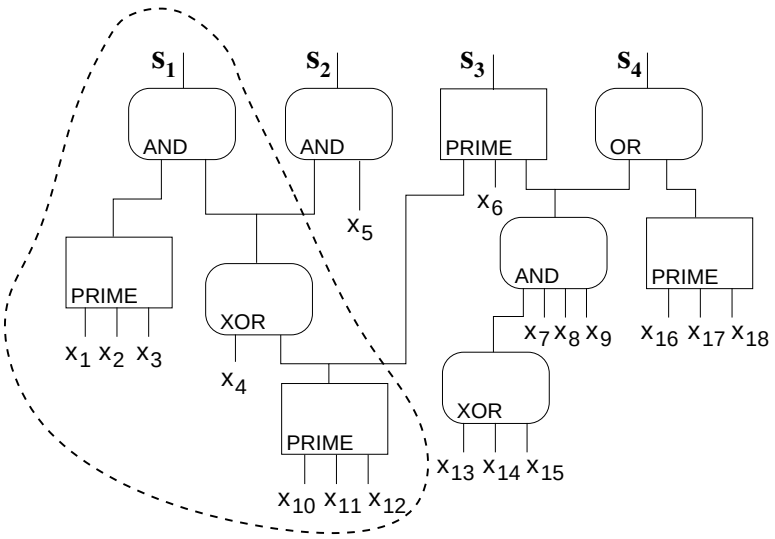

Figure 6.1: A decomposed state vector for a small design

The relationship between the set of states spanned by the new $\mathbf{PS_{@k}}$ vector function versus the original state vector and the entire search space is reported in Figure 6.2. It is worth comparing it with the corresponding Figure 3.5 of the CBSS parameterization of Chapter 3.

In the following sections we present various transformations that we apply to the decomposition graph to accomplish the objective of producing an exact parameterization with a more compact representation than the original state vector. For each of these transformations, we show that the function vectors before and after the transformation span the same identical range. The first technique, called *reduction at free points*, is independent of the type of decomposition node it applies to. *Prime function elimination* is specific to *PRIME* nodes, while *non-dominant variable removal* refers to variable inputs that fan out to associative operators nodes.

In presenting the techniques, we will refer to the generic vector function $\mathbf{F}$ instead of $\mathbf{S_{@k}}$ since such transformations can be applied to any Boolean vector function. Moreover, we will use the terms *decomposition graph* $\mathbf{F}$ and *function* $\mathbf{F}$ interchangeably to refer to the multiple output function $\mathbf{F}$.

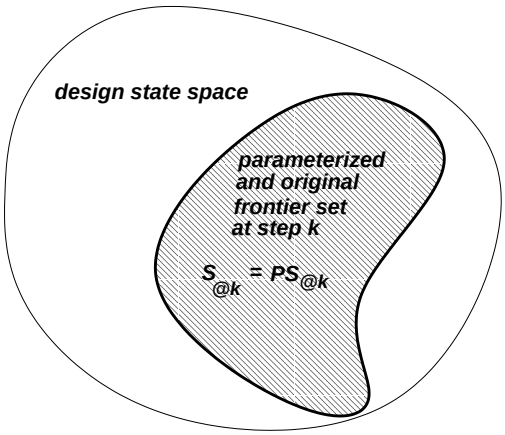

Figure 6.2: The parameterized frontier set $\mathbf{PS_{@k}}$

6.2 Reduction at Free Points

The first transformation, called *reduction at free points*, aims to simplify the decomposition graph by finding nodes which constitute a single cut-point. In other words, the output of such nodes is only affected by a set of variables which don't influence any other portion of the graph.

We first provide the definition of a *free point* and we show an example transformation. Then we provide proof that the transformation does not affect the range of the vector function.

The following definition is also illustrated in Figure 6.3:

Definition 6.1. *A* **free point** *p in a decomposition graph of* $\mathbf{F}$ *is a function corresponding to an output of a block in the graph. It has the property that, if we substitute the sub-graph rooted at the point with a new input variable w, the new function* $\mathbf{G}$ *has disjoint support with the function rooted at p:*

$$\mathbf{F}(x_1, \cdots, x_m) = \mathbf{G}(w, x_{p+1}, \cdots, x_m) \circ p(x_1, \cdots, x_p) \tag{6.1}$$

and $\mathcal{S}(\mathbf{G}) \cap \mathcal{S}(p) = \emptyset$.

Figure 6.3 shows three free points with darkened circles. Note that the output of p is a free point since none of the variables in the support of p appears in the support of other parts of the graph. On

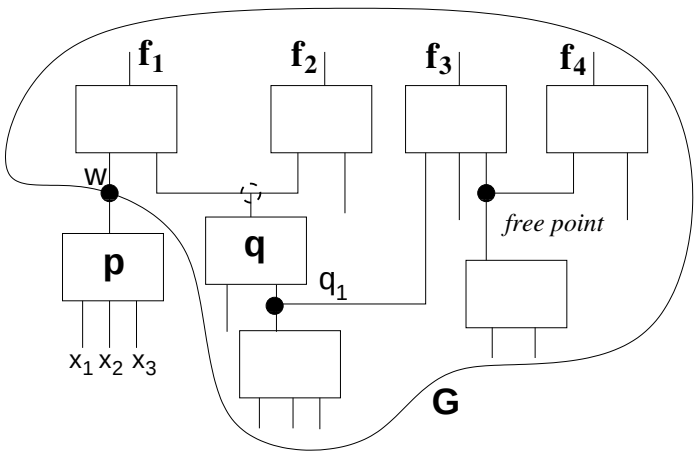

Figure 6.3: A vector function and its free points

the other hand, the dashed circle at the output of q is not a free point since, if we split the graph at that node, the two functions obtained, $\mathbf{H}$ and q with $\mathbf{F} = \mathbf{H} \circ q$, would still share the input q_1.

The following theorem shows that we can use free points to simplify the decomposition graph:

Theorem 6.1. *Given a decomposition graph for a multiple output Boolean function* $\mathbf{F}(x_1, \cdots, x_m) : \mathcal{B}^m \rightarrow \mathcal{B}^n$*, a free point* $p(x_1, \cdots, x_p) : \mathcal{B}^p \rightarrow \mathcal{B}$ *in it, and the function* $\mathbf{G}(p, x_{p+1}, \cdots, x_m) : \mathcal{B}^{m-p+1} \rightarrow \mathcal{B}^n$*, obtained by substituting the function* $p()$ *with the new input variable* p *in the graph of* $\mathbf{F}$*,* $\mathcal{R}(\mathbf{F}) = \mathcal{R}(\mathbf{G})$.

Proof. Consider the function $\mathbf{F}(x_1, \cdots x_m)$ and compute its range by splitting on the input variables [27]:

$$\mathcal{R}(\mathbf{F}) = \mathcal{R}(\mathbf{F}_{x_1=0}) \cup \mathcal{R}(\mathbf{F}_{x_1=1})$$

By applying this equation recursively over all the variables $(x_1, \cdots x_p)$ in the support of p, we obtain:

$$\mathcal{R}(\mathbf{F}) = \bigcup_{(i_1, \cdots, i_p) \in \mathcal{B}^p} \mathcal{R}(\mathbf{F}_{x_1=i_1, x_2=i_2, \cdots, x_p=i_p}) \tag{6.2}$$

Using Equation 6.1:

$$\mathbf{F}_{x_1=i_1,x_2=i_2,\cdots,x_p=i_p} = \mathbf{G}_{p=i_w} \quad \text{where} \quad i_w = p(i_1,\cdots,i_p) \in \{0,1\}$$

since p evaluates to a constant. Substituting in Eq. 6.2 we finally obtain:

$$\mathcal{R}(\mathbf{F}) = \bigcup_{i_w \in \{0,1\}} \mathcal{R}(\mathbf{G}_{p=i_w}) = \mathcal{R}(\mathbf{G})$$

□

Thus, we can substitute all the free points with new variables and generate a new state function **G** with a smaller representation.

A simple traversal of the graph is sufficient to discover all the free points with maximal support, that is, all the free points whose support is not contained in any other free point of the decomposition graph:

Definition 6.2. *A* **free point** $p()$ *is said to have* maximal support *if its support* $\mathcal{S}(p)$ *is not a proper subset of any other free point in the graph.*

The transformation of free sub-graphs with new variables produces a new function **G**, with $|\mathcal{S}(\mathbf{G})| \leq |\mathcal{S}(\mathbf{F})|$.

Example 6.1. *Consider the decomposition graph of Figure 6.4. Figure 6.4.a shows all the free points of the graph with filled circles. The free points surrounded by a dashed circle are also maximal and we can substitute the portion of the graph rooted at these nodes with a new parameter, without affecting the range of the graph. Figure 6.4.b shows the new, reduced graph obtained.*

Note that, anytime we perform a free point reduction we remove a set of input variables from the support of the vector function **F**. Thus, we can reassign any of these variables from a combinational input variable role to a parameter variable role and use it as the parameter assigned to the free point.

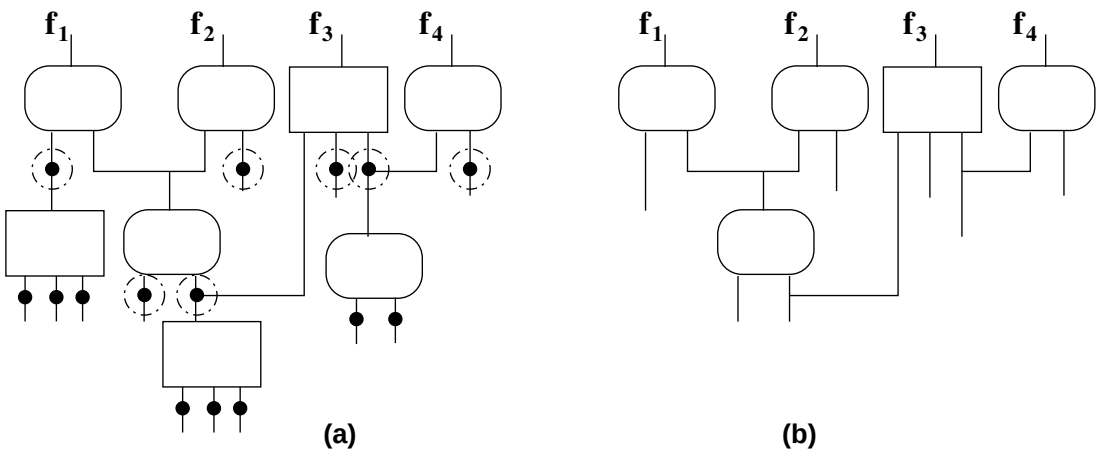

Figure 6.4: Free points elimination for Example 6.1

6.3 Elimination of Prime functions

As mentioned in Section 4.4.3, each block of a decomposition is either termed a *PRIME* function or it is an associative operator. We found that, if a *PRIME* function satisfies certain conditions, we can remove it from the decomposition graph, along with all of its sub-graph and substitute it with a fresh input variable.

In order for the substitution to be acceptable, the output node of the *PRIME* block has to be *almost* a free point, in the sense that up to one input of the *PRIME* block can be a node shared with rest of the decomposition graph. As the proof shows, in this special case, the tree rooted at the *PRIME* block can still be removed. In fact, *PRIME* blocks inherently guarantee that their output cannot be kept constant by assigning any single one of their input signals. It follows that, no matter what is the value for the node that is shared with the rest of the decomposition graph, the output of *PRIME* block can still assume both values 0 and 1, and thus has full range.

Theorem 6.2. *Given a prime function $r(r_1, \cdots, r_r)$ in a decomposition graph* **F***, if all of its inputs, except at most one, are free points, than the decomposition graph G obtained by substituting the new variable r for function $r()$,*

$$\mathbf{F}(x_1, \cdots x_m) = \mathbf{G}(r, \cdots x_m) \circ r(r_1, \cdots r_r)$$

is such that

$$\mathcal{R}(\mathbf{F}) = \mathcal{R}(\mathbf{G}).$$

Proof. We distinguish two cases:

1. All the inputs of the prime block are free points. Then the output of the free block is also a free point and the theorem reduces to the hypothesis of Theorem 6.1.

2. The prime block r has one input that it is not a free-point, say r_1, without loss of generality. All the other inputs to the prime function: $(r_2, \cdots, r_r)$ are still free points and we can assume that have been reduced to input variables by Theorem 6.1.

   In the most general case, r_1 is a single output function of other input variables that are in the support of both $\mathbf{G}$ and r: $\mathcal{S}(r_1) = (a_1, \cdots, a_p)$. The function $\mathbf{F}$ has then the form:

$$\mathbf{F}(a_1, \cdots a_p, r_2, \cdots r_r, \cdots x_m) = \mathbf{G}(r, a_1, \cdots a_p, r_1, \cdots x_m) \circ r(r_1, \cdots r_r) \circ r_1(a_1, \cdots a_p) \quad (6.3)$$

   Let's proceed again by computing the $\mathcal{R}(\mathbf{F})$ by recursively splitting on the input variables:

$$\mathcal{R}(\mathbf{F}) = \bigcup_{(i_1, \cdots, i_p) \in \mathcal{B}^p} \mathcal{R}(\mathbf{F}_{a_1 = i_1, a_2 = i_2, \cdots, a_p = i_p}) \quad (6.4)$$

   For each different assignment $(i_1, \cdots, i_p)$, r_1 evaluates to a constant value:

$$i_r = r_1(i_1, \cdots, i_p) \in \{0, 1\}.$$

   Substituting the expansion of $\mathbf{F}$ as in Eq. 6.3, we obtain:

$$\mathbf{F}_{a_1 = i_1, \cdots, a_p = i_p} = \mathbf{G}_{a_1 = i_1, \cdots, a_p = i_p, r_1 = i_{r_1}} \circ r_{r_1 = i_{r_1}} \quad (6.5)$$

   Note that we cannot drop the cofactors w.r.t. the a_i in $\mathbf{G}$ because r_1 is not a free point and thus

its inputs fan out to other nodes of the graph.

Now, the function $r_{r_1=i_{r_1}}(r_2,\cdots,r_r)$ is a free point and as such it can be substituted by a new free variable r. We show now that it is not possible that $r_{r_1=i_{r_1}}(r_2,\cdots,r_r)$ reduces to a constant for any value of i_{r_1}. In fact, if that was the case, r could be expressed as $r = r_1 \otimes r_{res}(r_2,\cdots,r_r)$, where $\otimes$ is either *AND* or *OR* and $\mathcal{S}(r_1)\cap\mathcal{S}(r_{res}) = \emptyset$. r would then have a disjoint support decomposition through an associative operator and would not be a *PRIME* function.

By carrying on the substitution $r = r_{r_1=i_{r_1}}(r_2,\cdots,r_r)$, Eq. 6.5 reduces to:

$$\mathbf{F}_{a_1=i_1,a_2=i_2,\cdots,a_p=i_p} = \mathbf{G}_{a_1=i_1,a_2=i_2,\cdots,a_p=i_p}$$

which substituted into Eq. 6.4 proves the theorem.

□

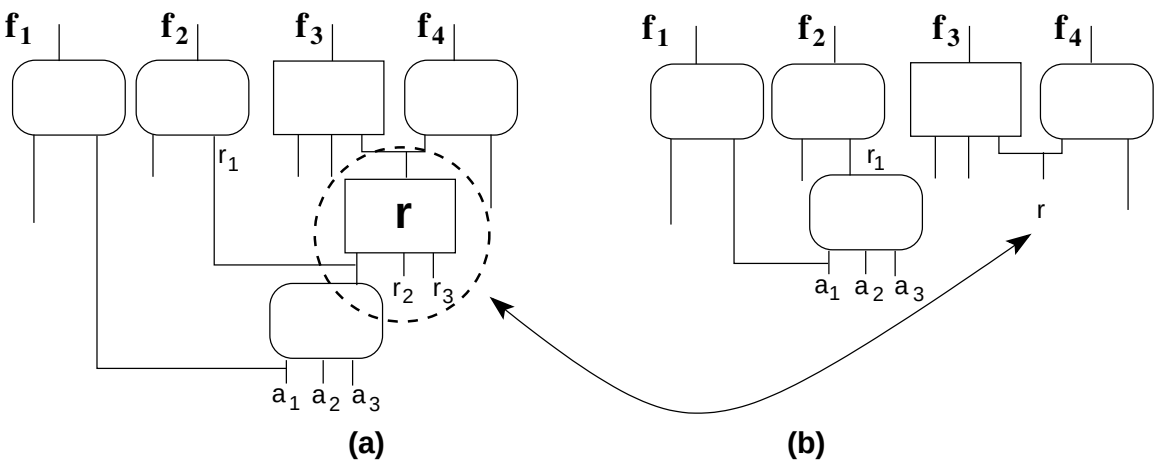

Figure 6.5: General case for prime function elimination: (a) before and (b) after

A possible structure for the graph **F** is represented in Figure 6.5.a: All the inputs to block r are free points, except for r_1. We can then remove the block r and substitute it with a new input variable obtaining the graph in Figure 6.5.b without affecting the range of the function. Note that input variables r_2 and r_3 are not needed anymore.

Example 6.2. *The testbench s1196 from the IWLS suite contains the blocks reported in Figure 6.6 in its next state function at step 10 of symbolic simulation. The variables names are just indices corresponding to the variables in the support of the state function. Since the prime function r has the two inputs x_{35} and x_{39} that are free points and only one input that has multiple fanout, we can completely eliminate this portion of the graph and just substitute it with the input variable r.*

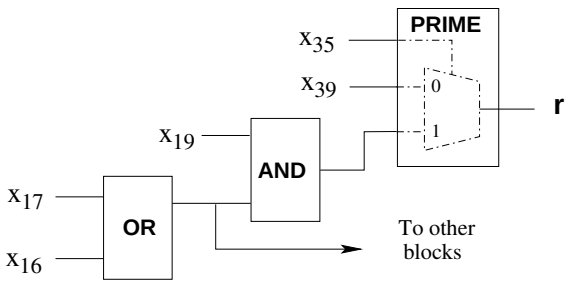

Figure 6.6: Prime elimination for Example 6.2.

6.4 Removal of non-dominant variables

Under certain conditions, an input variable can be removed from the decomposition graph without affecting its range.

Example 6.3. *Consider the following 3-outputs function:*

$$\begin{aligned} f_1 &= AND(b,e) \\ f_2 &= AND(e,OR(a,b,d)) \\ f_3 &= XOR(a,c) \end{aligned}$$

The range of this function is $\mathcal{B}^3 \setminus \{101,100\}$. We can remove the variable a from the function, by cofactoring all the components w.r.t. $a = 0$ without changing the range spanned by **F**. *The result*

is:

$$\begin{aligned} f_1 &= AND(b,e) \\ f_2 &= AND(e, OR(b,d)) \\ f_3 &= c \end{aligned}$$

and it still has range $\mathcal{B}^3 \setminus \{101, 100\}$.

We could do the simplification in the example because the range of the function for $a = 1$ is a subset of the range for $a = 0$. The following definition formalizes the situation:

Definition 6.3. *An input variable of a decomposition graph has a* **non-dominant value 0** *iff it fans out only to blocks that are decomposed through OR or XOR associative operators. It has a* **non-dominant value 1** *iff it fans out only to blocks that are AND or XOR decompositions. Otherwise it does not have a non-dominant value.*

Note in particular that a variable may have a non-dominant value 0 and a non-dominant value 1 simultaneously if it fans out only to XOR decompositions. The theorem below shows that in the most general case, a variable that fans out only to associative operators can be removed from the decomposition graph if it has a unique non-dominant value for the whole graph.

Theorem 6.3. *If a decomposition graph F has an input variable v with non-dominant value* $k \in \{0,1\}$*, and each of the blocks (i.e., intermediate single-output functions) that have v in their fanin have at least one other input in their fanin that is a free point, then:* $\mathcal{R}(\mathbf{F}) = \mathcal{R}(\mathbf{F}_{v=k})$

Proof. For a generic function **F**, we have:

$$\mathcal{R}(\mathbf{F}) = \mathcal{R}(\mathbf{F}_{v=k}) \cup \mathcal{R}(\mathbf{F}_{v=\bar{k}}) \tag{6.6}$$

We now show that under the conditions specified:

$$\mathcal{R}(\mathbf{F}_{v=\bar{k}}) \subseteq \mathcal{R}(\mathbf{F}_{v=k}) \tag{6.7}$$

and Eq. 6.6 reduces to $\mathcal{R}(\mathbf{F}) = \mathcal{R}(\mathbf{F}_{v=k})$.

Let's consider first the case where $k = 0$ and let's label each of the functions that have v in their fanin $x(v, p, x_1, \cdots x_x)$, $y(v, q, y_1, \cdots y_y)$, $w(v, r, w_1, \cdots w_w)$ ... where p, q, r ... are the free points in each of them and x_i, y_i, w_i ... are other variables the functions depend on. The $x()$, $y()$, $w()$, ... functions by hypothesis can only be *OR* or *XOR* decompositions.

We can then express $\mathbf{F}$ using the composition of these functions:

$$\mathbf{F} = \mathbf{G}(x, y, w, \cdots, x_1 \cdots x_x, y_1 \cdots y_y, \cdots w_w, \cdots) \circ x(v, p, x_1, \cdots x_x) \circ y(v, q, y_1, \cdots y_y) \cdots$$

Note that, in general, x_i, y_i, w_i ... are also in the fanin of $\mathbf{G}$. Let's now compute the two cofactors of $\mathbf{F}$ w.r.t. v:

$$\mathbf{F}_{v=0} = \mathbf{G}(x, y, w, \cdots, x_1 \cdots x_x, y_1 \cdots y_y, \cdots w_w, \cdots) \circ x(0, p, x_1, \cdots x_x) \circ y(0, q, y_1, \cdots y_y) \circ \cdots$$

$$\mathbf{F}_{v=1} = \mathbf{G}(x, y, w, \cdots, x_1 \cdots x_x, y_1 \cdots y_y, \cdots w_w, \cdots) \circ x(1, p, x_1, \cdots x_x) \circ y(1, q, y_1, \cdots y_y) \circ \cdots$$

In order to show the inclusion of the ranges of Eq. 6.7, we are going to represent each range as a union of ranges by cofactoring the variables in the support of x, y, w, ... one function at a time starting with $x()$:

$$\mathcal{R}(\mathbf{F}_{v=0}) = \bigcup_{(x_1 \cdots x_x) \in \mathcal{B}^x} \mathcal{R}(\mathbf{G}(x, y, \cdots x_1 \cdots, y_1 \cdots) \circ x(0, p, x_1 \cdots)_{x_1 = i_{x_1}, \cdots})$$

$$\mathcal{R}(\mathbf{F}_{v=1}) = \bigcup_{(x_1 \cdots x_x) \in \mathcal{B}^x} \mathcal{R}(\mathbf{G}(x, y, \cdots x_1 \cdots, y_1 \cdots) \circ x(1, p, x_1 \cdots)_{x_1 = i_{x_1}, \cdots})$$

We distinguish two cases for each x, y, w, ... function:

1. **x is a OR decomposition**. When all the $(x_1, \cdots, x_x)$ are zero, for $\mathbf{F}_{v=1}$, x evaluates to the constant value 1. For $\mathbf{F}_{v=0}$, $x = p$. In all the other cases x evaluates to 1. By grouping all the component ranges so that to distinguish the special case from all the others , we can simplify

the expressions:

$$\mathcal{R}(\mathbf{F}_{v=0}) = \mathcal{R}(\mathbf{G}(p,y,\cdots 0\cdots 0,\cdots)) \bigcup_{(x_1,\cdots,x_x)\neq \mathbf{0}} \mathcal{R}(\mathbf{G}(1,y,\cdots x_1\cdots x_x\cdots)_{x_1=i_{x_1},\cdots}$$
$$\mathcal{R}(\mathbf{F}_{v=1}) = \mathcal{R}(\mathbf{G}(1,y,\cdots 0\cdots 0,\cdots)) \bigcup_{(x_1,\cdots,x_x)\neq \mathbf{0}} \mathcal{R}(\mathbf{G}(1,y,\cdots x_1\cdots x_x,\cdots)_{x_1=i_{x_1},\cdots}$$

It can be easily seen that the first range for $\mathbf{F}_{v=1}$ is a subset of the corresponding range for $\mathbf{F}_{v=0}$, while the rest of the expression is identical.

2. **x is an XOR decomposition**. For the 1-cofactor, $\mathbf{F}_{v=1}$, $x = XNOR(p, x_1, \cdots, x_x)$. In the case of the 0-cofactor, $\mathbf{F}_{v=0}$, x evaluates to the complement: $x = XOR(p, x_1, \cdots, x_x)$. We can again group all the component ranges so that to distinguish the cases where $XOR(x_1, \cdots, x_x) = 0$ from the ones where $XOR(x_1, \cdots, x_x) = 1$:

$$\mathcal{R}(\mathbf{F}_{v=0}) = \bigcup_{XOR(x_1,\cdots x_x)=0} \mathcal{R}(\mathbf{G}(p,y,\cdots x_1\cdots)_{x_1=i_{x_1},\cdots} \bigcup_{XOR(x_1,\cdots x_x)=1} \mathcal{R}(\mathbf{G}(\bar{p},y,\cdots x_1\cdots)_{x_1=i_{x_1},\cdots}$$
$$\mathcal{R}(\mathbf{F}_{v=1}) = \bigcup_{XOR(x_1,\cdots x_x)=0} \mathcal{R}(\mathbf{G}(\bar{p},y,\cdots x_1\cdots)_{x_1=i_{x_1},\cdots} \bigcup_{XOR(x_1,\cdots x_x)=1} \mathcal{R}(\mathbf{G}(p,y,\cdots x_1\cdots)_{x_1=i_{x_1},\cdots}$$

And it can be observed that the two components of each expression match. It follows: $\mathcal{R}(\mathbf{F}_{v=0}) = \mathcal{R}(\mathbf{F}_{v=1})$.

This procedure can be applied recursively for each of the other functions y, w, ..., by computing and grouping all the cofactors for the sets of input variables $(y_1 \cdots y_y)$, $(w_1 \cdots w_w)$, ...

For the case where $k = 1$, the functions x, y, w, ... can now only be *AND* or *XOR* decompositions. The corresponding proof can be obtained by substituting *AND* for *OR* and 1 for 0 in the proof just discussed. Finally, for the case where the input variable v has both a non- dominant value 0 and 1, we can just use any of the two value- specific proofs. □

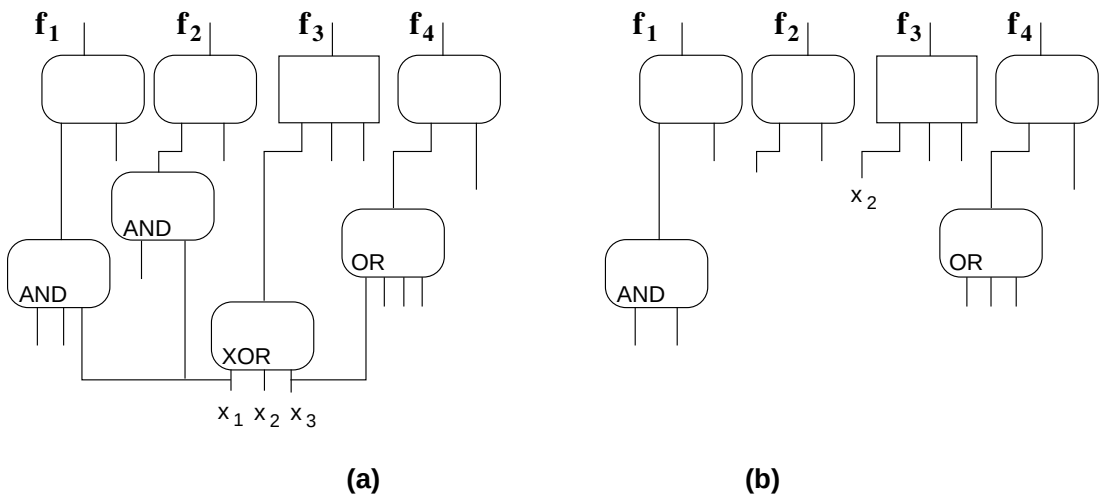

Figure 6.7: Non-dominant variable removal for Example 6.4

Example 6.4. *Figure 6.7.a shows a system with two non-dominant variables: x_1 has a non-dominant value 1, since it only fans out to AND and XOR nodes, while x_3 has a non-dominant value 0, since it fans out to OR and XOR. After removing of these two non-dominant variables and eliminating the nodes left with only one input, we obtain the system in Figure 6.7.b. Note that at this point we can apply the free point reduction technique to the graph of* **F**.

6.5 DSD-SS Implementation

Our implementation of the Disjoint Support Decomposition based Symbolic Simulator performs the parameterizations at the end of each symbolic simulation step. We first generate the decomposition graph for the state vector $\mathbf{S}_{@\mathbf{k}}$ and then attempt the three transformations described above. Often, the graph produced by applying one of the transformations enables further simplifications through some of the other transformations.

Even when all of the transformations fail, we still want to maintain a compact representation for the state function $\mathbf{S}_{@\mathbf{k}}$, so that we can make further progress with the simulation. Thus, when the state function exceeds a threshold value, we choose a variable to set to a constant value. The variable with fanout to the maximum number of blocks is selected becuase by simplifying this variable we

eliminate the largest interdependency among the nodes of the graph and thus we maximize the likelyhood of creating a graph where our techniques can be applied in future simulation steps. When computing the fanout of a primary variable that is candidate for elimination, we only consider those decomposition blocks which have other input variables in their fanin. The intuition behind this choice is that those blocks are closer to become free points, since some of their inputs are already free points.

We found experimentally that often, after eliminating a variable by setting it to constant as described, we could discover additional free points or variables with non-dominant values.

6.6 Experimental results

The algorithm presented in this chapter was implemented in a C++ program called Disjoint Support Decomposition based Symbolic Simulator (DSD-SS). We tested this approach on the largest sequential circuits from the Logic Synthesis Benchmarks suite [68] and the ISCAS'89 Benchmark Circuits [16], including their 1993 additions, as we did for the previous CBSS technique in Chapter 3. Table 6.1 reports results on all but the smallest testbenches of the two suites (we excluded from the table the circuits with less then 20 memory elements). The testbenches are grouped by benchmark suite. The experiments were run on a Linux PC equipped with a Pentium 4 processor running at 2.7Ghz and 2GB of memory and 512Kb of cache. We linked the DSD-SS to the CUDD package [29] as the underlying BDD manipulation library for the combinational portion of the simulation and a proprietary BDD package for the parameterizations. We set the reordering threshold in CUDD to 80,000 nodes. Each testbench is run for 100 simulation steps and, at the end of each step, DSD-SS performs the decomposition of the next state symbolic vector and applies the transformations described in Sections 6.2-6.4. Whenever the transformations are not sufficient to provide an exact small representation for the state vector, we resort to pick a variable to evaluate to a constant value, in order to guarantee a compact representation. The variable is chosen based on the criteria described in the previous section. After a few experiments, we chose 2,500 nodes as a reasonable

value to use for the upper limit for the size of the state vector. We noticed that, generally speaking, this value can be used to trade-off simulation breadth vs. time.

For each circuit, the table reports first the same relevant metrics that we presented before in Table 3.1: the number of inputs In, outputs Out, memory elements FF, and internal network gates Gates.

The next three columns report how many times we were able to apply our transformations: FP is the cumulative number of free point substitutions, PE is the number of prime function eliminations, NVD the number of non-dominant variables removals over all the symbolic simulation steps. The next column of this group, Null, counts the cumulative number of times where no exact transformation could be applied, but the state vector was within the limit size (of 2,500 nodes), and thus DSD-SS advanced to the next step of simulation without applying any parameterization. Note that during a single simulation step we may apply more than one technique until we reduce the state vector within limits or until no additional exact parameterization is possible. The values of Table 6.1 indicate that the conditions that allow an exact parameterization of the state vector are frequently met in almost all the circuits. In particular, in most cases the transformations can be applied successfully multiple times during each same simulation step. Free point elimination is the parameterization that achieves the best results across all the testbenches producing a total 2,417 exact simplifications over 4,200 simulation steps (42 testbenches, each run for 100 steps). The second most successful technique appears to be the non-dominant variable removal, which was applied for a total of 1,243 times, while prime function elimination satisfied the necessary conditions for exact parameterization only 139 times.

The purpose of the next group of columns is to compare the breadth of the state exploration between DSD-SS and a pure symbolic simulator that does not include parameterization. To this end, we built a plain symbolic simulator and we constrained it to have the same upper bound for the size of the state vector at the end of each simulation cycle as DSD-SS. While the only reduction technique available to the plain symbolic simulator was approximation of the state vector by evaluating symbolic variables to constant values, DSD-SS would attempt first exact parameterization,

and default to approximation only as a backup method. The number of variables approximated to constant provides an indication of how much the search breadth has been restricted: every time a variable is set to constant, we cut in half the amount of equivalent simulation traces checked by the exploration. Thus, in this section of the table, a bigger value indicates a more aggressive approximation and a smaller breadth of search. DSD-SS greatly outperformed a pure symbolic simulator in all but three testbenches. The situation of a test such as *s635*, can arise because DSD-SS chooses the variable to approximate so to maximize the chance of being able to perform additional exact parameterizations. This may not be the choice that leads to the smallest BDD vector size with the least number of approximations. However, in all the other cases, even with this disadvantage, DSD-SS avoids the elimination of many symbolic variables and propagates through the simulation a factor of 2 to 10 times more symbols over a plain symbolic simulator, when the same amount of memory is available. The situation of test *bigkey* is exceptional in this sense: because of the exact parameterizations, DSD-SS could avoid the evaluation to constant of more than 11,000 symbols over a plain symbolic simulator.

The last column reports the execution times of DSD-SS. The current implementation of DSD-SS at this point is fairly poor, since we need to transfer the data back and forth between the two BDD packages many times during the simulation. The proprietary BDD package that we currently use to perform the parameterizations has special functionalities for linking to the Disjoint Support Decomposition library. Execution times are also penalized by multiple variable reorderings in the CUDD package that are triggered by many of the testbenches. We hope in the near future to be able to directly link the DSD library to the CUDD package; we expect this connection to provide great improvements in the performance of DSD-SS. At this point, the plain symbolic simulator executes faster than DSD-SS since it can rely simply on the usage of the CUDD package. Still, in a few cases DSD-SS can gain enough advantage from a compact representation to be faster than the plain simulator, for instance, in the case of test *bigkey*.

Finally, the testbenches with a "-" mark indicate that either the plain symbolic simulator or DSD-SS run out of the allotted time of one hour of execution. For these testbenches we only report

Circuit	In	Out	FF	Gates	Par.techniques FP	PE	NDV	Null	Symbol reductions DSD-SS	PlainSym.	Time (s)
Logic Synthesis '91 - FSM tests											
ex1	9	19	20	622	0	0	0	100	0	0	0.3
s1423	17	5	74	830	5	0	3	18	156	659	48.78
s838	35	2	32	596	0	1	1	98	0	0	7.98
s953	16	23	29	658	67	0	1	6	555	677	108.37
Logic Synthesis '91 - Addition '93											
bigkey	262	197	224	9211	28	0	47	1	178	11781	30.17
clma	382	82	33	24482	9	0	20	69	12	10	17.46
dsip	228	197	224	3893	24	0	0	1	395	13043	428.48
mm9a	12	9	27	639	0	0	14	27	110	71	11.93
mm9b	12	9	26	786	2	0	3	7	211	277	62.18
mult16b	17	1	30	284	63	0	78	1	1185	1229	107.55
mult32a	33	1	32	715	0	0	0	1	2003	-	9507.86
s38417	28	106	1465	23771	49	1	13	6	148	867	7.75
s38584	38	304	1426	20281	138	1	34	9	516	1755	86.18
s5378	35	49	163	3232	136	0	30	1	636	1145	615.63
s838	34	1	32	618	0	0	0	51	52	61	66.21
s9234	36	39	135	3019	156	1	117	0	297	477	48.69
sbc	40	56	27	1143	183	1	28	1	1086	1314	244.3
ISCAS '89 - FSM tests											
s13207.1	62	152	638	9539	53	1	12	8	607	1080	35.1
s13207	31	121	669	9539	27	0	2	31	96	189	26.15
s1423	17	5	74	830	5	0	3	18	156	637	52.17
s15850.1	77	150	534	11316	103	23	100	0	994	2615	326.22
s15850	14	87	597	11316	2	0	98	0	55	120	9.93
s35932	35	320	1728	23085	0	0	0	16	245	1183	18.68
s38417	28	106	1636	27648	47	1	12	6	155	1293	6.52
s38584.1	38	304	1426	24619	124	0	51	9	500	1624	61.15
s38584	12	278	1452	24619	21	0	0	9	141	458	19.27
s5378	35	49	179	3973	150	0	58	1	680	1027	388.27
s838	34	1	32	626	0	0	0	51	52	61	72.76
S9234.1	36	39	211	6585	204	1	104	3	682	1096	110.46
s9234	19	22	228	6585	88	0	11	18	311	437	41.78
s953	16	23	29	658	38	0	1	7	547	764	110.97
ISCAS '89 - Addition '93											
prolog	36	73	136	1845	130	6	10	0	459	1201	169.99
s1269	18	10	37	771	13	0	3	2	-	1306	-
s1512	29	21	57	990	136	95	113	0	278	931	34.54
s3271	26	14	116	2166	0	0	1	16	714	1469	23.37
s3330	40	73	132	2020	144	5	51	0	472	1460	83.77
s3384	43	26	183	1734	61	2	3	4	1551	2565	297.18
s4863	49	16	104	2492	163	0	149	0	-	2400	-
s635	2	1	32	382	31	0	0	35	82	5	55.74
s6669	83	55	239	3272	17	22	0	1	-	6262	-
s938	34	1	32	626	0	0	0	51	52	61	70.76
s967	16	23	29	677	0	0	50	50	0	731	2.56

Table 6.1: Disjoint Support Decomposition-based simulation results

the number of transformations that we were able to complete.

6.7 Summary

This chapter introduced a new parameterization technique for symbolic simulation, DSD-SS. This work was published in [11]. Its core contribution is in exploiting the disjoint support decomposition properties of the state vector to generate a compact parameterization during symbolic simulation.

The major advantage of this approach is that it is a loss-less transformation, that means we can generate a compact representation of the state vector, without losing any of the information it carries between simulation steps. Results show that, within a fixed amount of memory resources dedicated to represent the frontier set, we can keep a much broader search space than pure symbolic simulation.

Chapter 7

Conclusion

This thesis presents two major theoretical contributions:

1. Techniques were developed for the parameterization of Boolean functions, so that, given a vector function, we can generate an alternative more compact representation that spans the same range as the original one, and

2. Novel contributions were made to the theory of disjoint support decomposition of Boolean functions by providing a new canonical form to represent the unique maximal disjoint decomposition of a logic function and a novel and efficient algorithm that can automatically discover this decomposition in polynomial time.

Both of these theoretical contributions were applied to the problem of formal verification of digital systems and two new techniques were developed for the use symbolic simulation in verifying digital systems. These approaches expand the robustness and scalability of symbolic simulation and expand its accessibility for adoption in current industrial design practices.

7.1 Parameterized approaches in symbolic simulation

Parametric representations help to control BDD explosion in symbolic simulation, which is the central problem in formal verification. Based on our parameterization techniques, we have shown

simulation results on industrial design blocks that show the significant performance gains that these techniques can achieve both over logic simulation and symbolic simulation. In particular we have shown how CBSS (Cycle-Based Symbolic Simulation) provides many orders of magnitude better performance, measured in test vectors simulated per second, over logic simulation. At the same time, CBSS does not require any change in the verification user model, and thus can replace logic simulation in a transparent fashion. The same checks, or assertions, that are used to verify the outputs of logic simulation – see Section 1.1 – can be used in the context of CBSS to verify the expressions for the output of the digital system. DSD-SS (Disjoint-Support Decomposition based Symbolic Simulation) provides an exact parameterization technique, and we have shown that for a fixed amount of memory resource it can explore a much broader search space compared to a pure symbolic simulation approach. DSD-SS also provides a significant performance gain over logic simulation.

The two new solutions presented can be viewed as trade-off points in a search breadth vs. simulation scalability plane as indicated qualitatively in Figure 7.1: logic and symbolic simulations are at the extremes of scalability and search breadth, while CBSS and DSD-SS provide additional trade-off points between the two parameters. A major advantage of DSD-SS is that the memory resources required for the simulation can be traded off for reduced search breadth.

7.2 Disjoint support decompositions

Disjoint support decompositions (DSD) are a useful property of Boolean functions that can be used in many areas of computer-aided design automation. The ability to partition a function into blocks that depend only on a small portion of the support set of the function is valuable for decreasing the computational complexity of the function. Applications of DSD to the synthesis domain span from the routing arena, where the decomposition suggests the clustering of input signals that only affect a portion of the circuit, to multi-level synthesis, where it provides an automatic way of moving from a flat representation to a hierarchical one based on the decomposition tree. In this thesis we explored

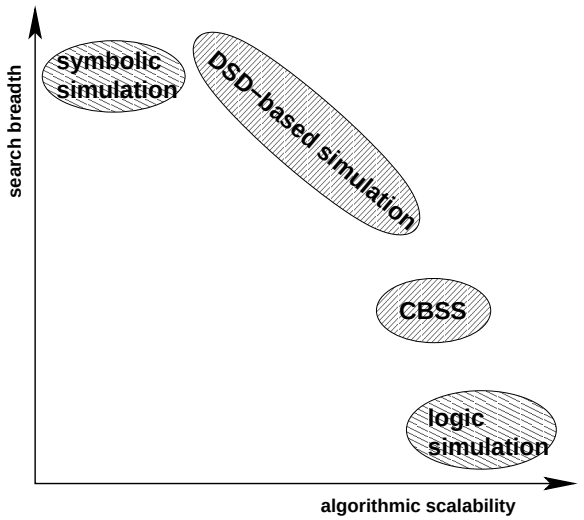

Figure 7.1: Trade-offs of in the breadth vs. scalability plane

some aspects of the benefits that decompositions can provide in verification and, in particular, simulation. Here, we exploited the fact that DSD exposes the inherently parallel components in the computation of a function and we used this fact to generate a simplified function. DSDs can also be used to generate a good initial variable order for the BDD of a function, by clustering inputs togheter that affect the same blocks.

In our experiments with the algorithm we introduced, we found that most functions, at least among the ones related to industrial designs, have meaningful decompositions, which is a promising starting point for the possible applications we have outlined and others that we have not thought about.

7.3 The future of this work

This work, as with most research, suggests future directions of research. From a practical standpoint, the efficiency of the implementations can be considerably improved. Such improvements would not only increase the robustness of the algorithms, but also provide even better results when comparing the performance to other simulators. Currently, the front-end of the simulators is limited to a simple

logic intermediate language. Supporting standard hardware description languages would increase the ease of experimenting with current industrial designs and provide better insights on the strengths and weaknesses of our techniques.

The robustness of the parameterization techniques introduced could only be evaluated accurately by building a usage framework around the raw simulations engines we have, so that a user can provide assertions, or specify correct behaviour for the design under simulation, and available test vectors to direct the search when the simulator needs to approximate.

Parameterization is a general technique that can be applied in other phases of the simulation flow. We would like to explore the additional robustness that we could obtain by deploying these or other parameterizations in the simulation of the combinational network in order to reduce the memory resources required there.

The application of the theory of disjoint support decompositions to other areas of design automation such as the ones indicated above is a direction that promises to lead to a broad range of problems and solutions. The few cases where DSD can not break a function into sufficiently small components suggest that there are more complex types of decomposition for which a canonical form has yet to be defined.

In a broader spectrum, the verification of digital designs is becoming the hardest problem in design automation and the immediate bottleneck to remove in order to maintain the growth trends that the IC industry has seen in the past 40 years. Current verification methods are struggling to handle the complexity of a single component module in a digital system on a chip design. We believe that the answer to this problem will come from two directions: improved scalability of verification algorithms on one end, and a modular verification methodology on the other. Parameterization techniques are one way of improving such scalability; however, we can se how, even only within the symbolic simulation approach, other techniques are needed to improve scalability and robustness during the whole verification flow.

Bibliography

[1] Mark Aagaard, Robert Jones, and Carl-Johan Seger. Formal verification using parametric representations of Boolean constraints. In *DAC, Proceedings of Design Automation Conference*, pages 402–407, June 1999.

[2] Aharon Aharon, Dave Goodman, Moshe Levinger, Yossi Lichtenstein, Yossi Malka, Charlotte Metzger, Moshe Molcho, and Gil Shurek. Test program generation for functional verification of PowerPc processors in IBM. In *DAC, Proceedings of Design Automation Conference*, pages 279–285, June 1995.

[3] Robert L. Ashenhurst. The decomposition of switching functions. In *Proceedings of the International Symposium on the Theory of Switching*, Part I 29, pages 74–116, 1957.

[4] Zeev Barzilai, J. Lawrence Carter, Barry K. Rosen, and Joseph D. Rutledge. HSS - a high-speed simulator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 601–617, July 1987.

[5] Janick Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2nd edition, 2003.

[6] Jules Bergmann and Mark Horowitz. Improving coverage analysis and test generation for large designs. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 580–583, November 1999.

[7] Valeria Bertacco and Maurizio Damiani. Boolean function representation based on disjoint-support decompositions. In *ICCD, Proceedings of the International Conference on Computer Design*, pages 27–33, October 1996.

[8] Valeria Bertacco and Maurizio Damiani. Boolean function representation using parallel-access diagrams. In *Proceedings of the Sixth Great Lakes Symposium on VLSI*, March 1996.

[9] Valeria Bertacco and Maurizio Damiani. The disjunctive decomposition of logic functions. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 78–82, November 1997.

[10] Valeria Bertacco, Maurizio Damiani, and Stefano Quer. Cycle-based symbolic simulation of gate-level synchronous circuits. In *DAC, Proceedings of Design Automation Conference*, pages 391–396, June 1999.

[11] Valeria Bertacco and Kunle Olukotun. Efficient state representation for symbolic simulation. In *DAC, Proceedings of Design Automation Conference*, June 2002.

[12] Beate Bollig, Martin Löbbing, and Ingo Wegener. Simulated annealing to improve variable orderings for OBDDs. In *International Workshop on Logic Synthesis*, pages 5.1–5.10, May 1995.

[13] Karl Brace, Richard Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *DAC, Proceedings of Design Automation Conference*, pages 40–45, 1990.

[14] Robert K. Brayton and Curt McMullen. The decomposition and factorization of boolean expressions. In *ISCAS, Proceedings of the International Symposyium on Circuits and Systems*, pages 49–54, 1982.

[15] Robert K. Brayton, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert R. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(6):1062–1081, November 1987.

[16] Franc Brglez, David Bryan, and Krzysztof Koźmiński. Combinational profiles of sequential benchmark circuits. In *ISCAS, Proceedings of the International Symposyium on Circuits and Systems*, pages 1929–1934, May 1989.

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

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

[19] Randal E. Bryant. Symbolic boolean manipulation with ordered binary–decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[20] Randal E. Bryant, Derek Beatty, Karl Brace, Kyeongsoon Cho, and Thomas Sheffler. COSMOS: A compiled simulator for MOS circuits. In *DAC, Proceedings of Design Automation Conference*, pages 9–16, June 1987.

[21] Jerry R. Burch, Edward M. Clarke, David E. Long, Ken L. MacMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[22] Jerry R. Burch and David E. Long. Efficient boolean function matching. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 408–411, November 1992.

[23] Gianpiero Cabodi, Paolo Camurati, Luciano Lavagno, and Stefano Quer. Disjunctive partitioning and partial iterative squaring: an effective approach for symbolic traversal of large circuits. In *DAC, Proceedings of Design Automation Conference*, pages 728–733, June 1997.

[24] Gianpiero Cabodi, Paolo Camurati, and Stefano Quer. Improved reachability analysis of large finite state machine. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 354–360, November 1996.

[25] Srihari Cadambi, Chandra S. Mulpuri, and Pranav N. Ashar. A fast, inexpensive and scalable hardware acceleration technique for functional simulation. In *DAC, Proceedings of Design Automation Conference*, pages 570–575, June 2002.

[26] A. Chandra, V. Iyengar, D. Jameson, R. Jawalekar, I. Nair, B. Rosen, M. Mullen, J. Yoon, R. Armoni, D. Geist, and Y. Wolfsthal. AVPGEN - a test generator for architecture verification. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(2):188–200, June 1995.

[27] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In *Automatic Verification Methods for Finite State Systems, International Workshop*, volume 407 of *Lecture Notes in Computer Science*, pages 365–3. Springer, June 1989.

[28] Olivier Coudert and Jean Christophe Madre. Implicit and incremental computation of primes and essential primes of Boolean functions. In *DAC, Proceedings of Design Automation Conference*, pages 36–39, June 1992.

[29] CUDD-2.3.1. *http://vlsi.Colorado.edu/~fabio*.

[30] Herbert A. Curtis. *A New Approach to the Design of Switching Circuits*. Van Nostrand, Princeton, N.J., 1962.

[31] Charles J. DeVane. Efficient circuit partitioning to extend cycle simulation beyond synchronous circuits. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 154–161, nov 1997.

[32] Masahiro Fujita, Hisanori Fujisawa, and Nobuaki Kawato. Evaluation and improvements of boolean comparison method based on binary decision diagrams. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 2–5, November 1988.

[33] Craig Hansen. Hardware logic simulation by compilation. In *DAC, Proceedings of Design Automation Conference*, pages 712–716, June 1988.

[34] Faisal I. Haque, Khizar A. Khan, and Jonathan Michelson. *The Art of Verification with Vera*. Verification Central, 2001.

[35] Scott Hauck. *Multi-FPGA Systems*. PhD thesis, University of Washington, Dept. of Computer Science and Engineering, 1995.

[36] Pei-Hsin Ho, Thomas Shiple, Kevin Harer, James Kukula, Robert Damiano, Valeria Bertacco, Jerry Taylor, and Jiang Long. Smart simulation using collaborative formal and simulation engines. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 120–126, November 2000.

[37] Yoav Hollander, Matthew Morley, and Amos Noy. The *e* language: A fresh separation of concerns. In *Technology of Object-Oriented Languages and Systems*, volume TOOLS-38, pages 41–50, March 2001.

[38] Alan J. Hu and David L. Dill. Reducing BDD size by exploiting functional dependencies. In *DAC, Proceedings of Design Automation Conference*, pages 266–271, June 1993.

[39] Gérard Huet. Higher order unification 30 years later. In *Theorem Proving in Higher Order Logics*, volume 2410 of *Lecture Notes in Computer Science*, pages 3–12. Springer-Verlag, August 2002.

[40] Prabhat Jain and Ganesh Gopalakrishnan. Hierarchical constraint solving in the parametric form with applications to efficient symbolic simulation based verification. In *ICCD, Proceedings of the International Conference on Computer Design*, pages 304–307, October 1993.

[41] Prabhat Jain and Ganesh Gopalakrishnan. Efficient symbolic simulation-based verification using the parametric form of boolean expressions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13:1005–1015, August 1994.

[42] Steven Johnson. View from the fringe of the fringe. In *CHARME*, volume 2144 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, September 2001.

[43] Michael Kantrowitz and Lisa M. Noack. I'm done simulating; now what? verification coverage analysis and correctness checking of the DECchip 21164 Alpha microprocessor. In *DAC, Proceedings of Design Automation Conference*, pages 325–330, June 1996.

[44] Richard M. Karp. Functional decomposition and switching circuit design. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):291–335, 1963.

[45] Kevin Karplus. Representing boolean functions with if-then-else dags. Technical Report UCSC-CRL-88-28, Baskin Center for Computer Engineering & Information Sciences, 1988.

[46] Kevin Karplus. Using if-then-else dags for multi-level logic minimization. In *Proceedings of Advanced Research in VLSI*, pages 101–118, 1989.

[47] Kevin Karplus. Using if-then-else dags to do technology mapping for field-programmable gate arrays. Technical Report UCSC-CRL-90-43, Baskin Center for Computer Engineering & Information Sciences, 1990.

[48] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[49] Tommy Kuhn, Tobias Oppold, Markus Winterholer, Wolfgang Rosenstiel, Marc Edwards, and Yaron Kashai. A framework for object oriented hardware specification, verification and synthesis. In *DAC, Proceedings of Design Automation Conference*, pages 413–418, June 2001.

[50] Oded Lachish, Eitan Marcus, Shmuel Ur, and Avi Ziv. Hole analysis for functional coverage data. In *DAC, Proceedings of Design Automation Conference*, pages 807–812, June 2002.

[51] Frédéric Mailhot and Giovanni DeMicheli. Algorithms for technology mapping based on binary decision diagrams and on boolean operations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12:599–620, May 1993.

[52] Sharad Malik, Albert Wang, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 6–9, November 1988.

[53] Patrick C. McGeer, Jagesh V. Sanghavi, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. ESPRESSO-SIGNATURE: A new exact minimizer for logic functions. In *DAC, Proceedings of Design Automation Conference*, pages 618–624, June 1993.

[54] In-Ho Moon, James Kukula, Kavita Ravi, and Fabio Somenzi. To split or to conjoin: The question in image computation. In *DAC, Proceedings of Design Automation Conference*, pages 23–28, June 2000.

[55] Rajeev Murgai, Yoshihito Nishizaki, Narendra V. Shenoy, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Logic synthesis for programmable gate arrays. In *DAC, Proceedings of Design Automation Conference*, pages 620–625, June 1990.

[56] Gregory F. Pfister. The yorktown simulation engine: Introduction. In *DAC, Proceedings of Design Automation Conference*, pages 51–54, January 1982.

[57] Kavita Ravi and Fabio Somenzi. High density reachability analysis. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 154–158, November 1995.

[58] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 42–47, November 1993.

[59] Tsutomu Sasao. Totally undecomposable functions: Applications to efficient multiple-valued decompositions. In *ISMVL*, pages 59–65, 1999.

[60] Tsutomu Sasao and Munehiro Matsuura. DECOMPOS: An integrated system for functional decomposition. In *International Workshop on Logic Synthesis*, pages 471–477, 1998.

[61] Claude E. Shannon. The synthesis of two-terminal switching circuits. *Bell Systems Technical Journal*, 28(1):59–98, 1949.

[62] V. Yun-Shen Shen, Archie C. McKellar, and Peter Weiner. A fast algorithm for the disjunctive decomposition of switching functions. *IEEE Transactions on Computers*, C-20(3):304–309, 1971.

[63] Theodore Singer. The decomposition chart as a theoretical aid. Technical Report BL-4, Sec.III, Harvard Computational Laboratory, 1953.

[64] Hervé Touati, Hamid Savoj, Bill Lin, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 130–133, November 1990.

[65] C. A. J. van Eijk and Jochen A. G. Jess. Exploiting functional dependencies in finite state machine verification. In *ED&TC, Proceedings of the European Design and Test Conference*, pages 9–14, March 1996.

[66] Laung-Terng Wang, Nathan E. Hoover, Edwin H. Porter, and John J. Zasio. SSIM: A software levelized compiled-code simulator. In *DAC, Proceedings of Design Automation Conference*, pages 2–8, June 1987.

[67] Chris Wilson and David L. Dill. Reliable verification using symbolic simulation with scalar values. In *DAC, Proceedings of Design Automation Conference*, pages 124–129, June 2000.

[68] Saeyang Yang. Logic synthesis and optimization benchmarks user guide, version 3.0. Technical report, Microelectronics Center of North Carolina, January 1991.

[69] Jun Yuan, Kurt Schultz, Carl Pixley, Hiller Miller, and Adnan Aziz. Modeling design constraints and biasing using bdds in simulation. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 584–590, November 1999.