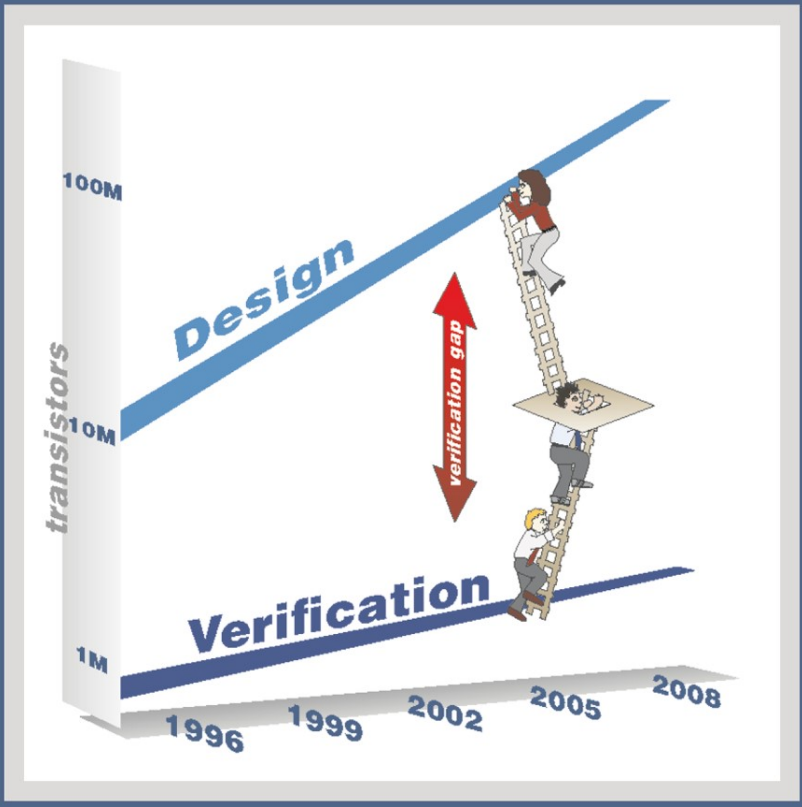
Scalable Hardware Verification with Symbolic Simulation

Valeria Bertacco



Springer

SCALABLE HARDWARE VERIFICATION WITH SYMBOLIC SIMULATION

SCALABLE HARDWARE VERIFICATION WITH SYMBOLIC SIMULATION

VALERIA BERTACCO
University of Michigan

Valeria Bertacco
The University of Michigan
Advanced Computer Architecture Lab
Department of EE & CS
1301 Beal Avenue, Room 2224
Ann Arbor, MI 48109
U.S.A.

Scalable Hardware Verification with Symbolic Simulation

Cover design by S. Alexander Garcia

Library of Congress Control Number: 20055934803

ISBN-10: 0-387-24411-5
ISBN-13: 9780387244112
ISBN-10: 0-387-29906-8 (e-book)
ISBN-13: 9780387299068 (e-book)

Printed on acid-free paper.

Printed in the United States of America.

9 8 7 6 5 4 3 2 1 SPIN 11325031

springeronline.com

To Roberta, Bruno,
Livio and Todo.

Contents

List of Figures

List of Tables

Preface

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

The design of digital systems is an intricate and time consuming process that progresses through various phases and levels of abstraction relying heavily on CAD (Computer-Aided Design) software tools. Within this context, ensuring the correctness of these digital systems is a critical consideration, especially because failure costs are becoming increasingly high. One of the most famous, recent examples of the importance of correct design is the Intel Pentium flaw in the floating point divide unit in 1994 that eventually forced Intel to replace many of the Pentium chips that were already in the market. In many cases, the possibility of failure is plainly unacceptable. Examples of these applications are transportation systems, medical applications and financial systems. Driven by the importance of correct design, the cost of verification in modern computing systems has grown to dominate the cost of system design in terms of the time and human resources dedicated to it. In contrast, even though guaranteeing the correctness of a design is such a central aspect of 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 to verification, yet are still unable to guarantee correct functionality over the entire design space.

In industry, the scalability, flexibility and predictable run-time behavior of logic simulation makes it the most widely accepted technique for ensuring the correctness of digital ICs. The 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 to see or described in a specification document. However, logic simulation can usually 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 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 runtime behavior. Thus, the scalability of this approach has been the main limiting factor to its mainstream deployment, with the consequence that, thus far, its scope has been limited to small systems.

About this book

This book presents recent advancements in symbolic simulation-based solutions which radically improve scalability. We overview current verification techniques, both based on logic simulation and on formal verification methods, and we describe in detail the baseline technique of symbolic simulation. The core of this book focuses on new techniques that narrow the performance gap between the complexity of digital systems and the limited ability to verify them. In particular we cover a range of solutions that exploit approximation and parametrization methods in order to achieve this goal. In the direction of approximation techniques, we comprehensively cover quasi-symbolic simulation – an aggressive technique aiming at simulating only the portion of the design necessary for the verification goal at hand – and cycle-based symbolic simulation – 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. Cycle-based symbolic simulation is a hybrid solution that uses both approximation and parametrization to attain its scalability goal. Its 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 logic simulation techniques, while achieving better efficiency.

In the realm of parametric solutions, we discuss a range of approaches, including various applications of parametric symbolic simulation to industrial microprocessor designs. An in-depth analysis is dedicated to another solution that we recently proposed, disjoint-support decomposition-based symbolic simulation, where the parametrization makes use of the disjoint-support decomposition properties of a Boolean function. This simulation technique is

rooted on a novel 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, compared to that of previous solutions which had exponential complexity in the number of input variables of the function. We deploy this algorithm to decompose of the state functions in symbolic simulation. Then, by restructuring the next-state functions using their disjoint components, it is possible to transform them into a simpler parametric form without sacrificing simulation accuracy. Results show that this technique is effective in reducing the memory requirements of symbolic simulation while maintaining exact state exploration. When the design complexity becomes overwhelming, it can trade-off search breadth for performance, and proceed to simulate very large trace sets in parallel, thus maintaining a simulation speed and memory profile that are close to logic simulation.

In structuring this book, the hope was to provide an interesting reading for a broad range of readers. Chapters 1, 2 and 3 constitute a panoramic flight over the world of digital systems' design and, in particular, verification. Chapter 3 reviews some of the mainstream symbolic techniques in formal verification, dedicating most of the focus to symbolic simulation.

We use Chapter 4 to cover the necessary principles of parametric forms and disjoint-support decompositions. In particular, we attempt to keep the material at a level that facilitates understanding, but without too many formal details. While there is a range of resources discussing parametric forms and parametrizations for Boolean functions, we felt that the topic of disjoint-support decompositions was not as readily available. For that reason Appendix A complements Chapter 4 in providing a more formal presentation of the topic and derivation of the theoretical results.

Chapters 5 and 6 focus on a range of recent symbolic simulation techniques, which we grouped in approximate solutions, and exact parametrizations. Finally, Chapter 7 wraps up the presentation and outlines possible future research directions.

Acknowledgments

I would like to acknowledge several people who were critical in making this book a reality through their work, advice and support. The main solutions discussed by this book were developed during my Ph.D. work at Stanford, with the supervision of Kunle Olukotun, who had always been prompt and available in supporting whatever direction of research, and of life, I pursued. In our technical interactions, he would always move straight to the results of my work and challenge me on their practical contribution to improve the quality of verification in industrial designs. David Dill has been the person I could always go to bounce ideas off of and to have illuminating technical discussions. When my ideas could survive his dissecting analysis, I knew I could publish them.

My years at 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 Vera Group team, who gave me early opportunities to interact with customers. Those customer meetings have always been enlightening in my quest towards understanding the needs of hardware designers; Pei-Hsin Ho, my manager in the Advanced Technology Group of Synopsys, showed me how to efficiently achieve technology transfers, by taking academic research and deploying it in software solutions for the hardware-design community. 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.

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 the pursuit of both education and industry experience. I also want to thank my family for supporting my choices in my path through life. My brother, Livio, provided all sorts of technical support and advice and solved many sys-

tem crashes, most often connecting from some remote location in Europe. My colleagues and my students at the University of Michigan have been an important source of support in continuing this research and in helping shuffle all the work and deadlines that are always overlapping: Kai-hui Chang, Steve Plaza, Smitha Shyam and Ilya Wagner. I would also like to thank Todd Austin for the numerous technical discussions and for taking the time to review this book multiple times. In addition, I would like to thank Gloria Cadavid, Alex Garcia, Tim Wright and Azita Emami.

Finally, I would like to express gratitude to my editors Michael Hackett and Alex Greene and their collaborators, Rose Antonelli, Melissa Guasch and Rebecca Olson at Springer for their help in preparing the final manuscript and keeping me on task, particularly during a year of transition and growth in the company.

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. In the same respect, 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 human 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 reasonable confidence 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 yet 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 large 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 to 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 limited.

1.1 Functional validation

Designers normally try to ensure correctness by developing multiple sets of tests that stimulate a digital design and by later 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 handcrafted by the designers to cover those areas of the design that they wish to validate. For a large, complex system it is impossible to test and/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 consequently verified, by 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 the overall state space. Furthermore, simulation inputs are usually based on the design specification and, therefore, 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 in 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. Consequently these cases do not get checked, while they may likely 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 the design's specification. In a formal verification approach, the desired functionality of the system is completely specified, then a formal model of the system is 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 will hold 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 mainly been 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 an increased interest in this approach of verification which has led to new algorithmic solutions and to new approaches that compromise 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, thus far these tools have not shown the required robustness to be included in the industry mainstream verification methodology, and have only been applied to experimental projects. One of the main limitations shown by these first attempts is 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 expressions representing, implicitly, all configurations – or sets 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, simultaneously, under all possible inputs. Hence, it has the potential of 1) verifying many configurations of the design in parallel and providing much better coverage than traditional logic simulation, and 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 exhausts the memory resources of the host computer even on designs of limited complexity.

This book addresses the robustness and scalability limitations of symbolic simulation and discusses recent algorithms that dramatically reduce the memory requirements compared to traditional techniques. The algorithms presented attack the scalability problem either through approximation or by using reparametrization, or both.

Among the techniques that use both approximation and parametrization we present a solution called cycle-based symbolic simulation. This technique simplifies the Boolean expressions involved in symbolic simulation by substituting the state vector with an alternate parametric form whose co-domain spans a very large subset of the original set of states described by the state vector. The resulting simulator maximizes the level of parallelism achievable with a limited amount of memory. Cycle-based symbolic simulation performs well from a scalability standpoint, and achieves a high level of parallelism, in terms of test vectors that are run through the simulator, while maintaining a low memory profile.

Another approximation technique presented here is quasi-symbolic simulation, which achieves scalability by avoiding the computation of complex expressions for internal nodes that are not involved in the verification of the specific aspect of the design under inspection. The selection of the nodes whose computation can be avoided is automatically estimated by the simulator and adjusted during re-simulation if the estimate is found over-conservative.

In the quest for effective parametrization techniques, we found that exploiting the decomposition properties of Boolean functions had the potential for an exact, yet computationally efficient parametrization. Hence, we introduce the theory of disjoint-support decompositions and present a new, efficient algorithm that exposes all the decompositions of a 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 that decompose F into disjoint support blocks: $F(x_1, \cdots, x_m) = G(H(x_1, \cdots, x_h), x_{h+1}, \ldots, x_m)$. An exact solution to this problem that had exponential complexity in the number of variables of the function,

was proposed in the late 1950's. The algorithm we present in this book takes, as input, a binary decision diagram (BDD) representation of a Boolean function and restructures it into its disjoint decomposition components. The worst-case complexity of the algorithm we propose is only quadratic in the size of the BDD, making it well suited for use with complex functions. In discussing exact parametrizations, we apply this algorithm to transform and simplify the Boolean expressions involved in symbolic simulation so that the simulation requires fewer memory resources while continuing to provide the original quality of results. Experimental results show that these solutions provide often more than ten orders-of-magnitude better performance (in test vectors per second) than a logic simulator, while, at the same time, they improve the design complexity that can be addressed by symbolic simulation by enabling simulations with up to thousands more symbolic variables.

1.3 Organization of the book

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. Chapters 2 and 3 present the models of digital systems used in verification and the main algorithms for both traditional simulation and formal verification. In particular, Chapter 3 focuses on symbolic simulation and closely related techniques. In addition, it covers the history of symbolic simulation, and provides a very broad presentation of the ongoing research on the subject.

Chapter 4 focuses on two background theories that are key to the solutions proposed in this book: parametrizations and disjoint-support decompositions. We introduce these topics from a practical point of view and through many examples. In discussing disjoint-support decompositions we present our BDD-based algorithm and show, with experimental results, that many Boolean functions arising in digital designs are non-trivial decompositions. The Appendix complements the discussion with a formal presentation of decompositions, a proof of the uniqueness of the maximal decomposition, and of the completeness of the BDD-based algorithm.

We then dive in the presentation of recent, effective solutions in the symbolic simulation space. Chapters 5 and 6 group these solutions into approximation-based and parametric techniques. Cycle-based symbolic simulation and quasi-symbolic simulation are discussed in Chapter 5. The chapter provides a presentation of the algorithms and simulation results comparing the performance of cycle-based simulation to a plain logic simulator.

Chapter 6 is dedicated to the discussion of exact parametric solutions. In particular, disjoint-support decomposition-based symbolic simulation exploits the decomposition properties of the state vector functions arising in simulation. We compare the results of this approach to a plain symbolic simulator and to cycle-based simulation, showing that DSD-based simulation can address much

more complex designs within the same budget of memory resources. Another solution presented in Chapter 6 is concerned with the application of symbolic simulation to microprocessor-design verification.

We conclude the book with Chapter 7, which includes a review of the methods presented and some directions for future research. In each chapter we attempt to provide, first, a general overview of the topic discussed and the related research work that has been published on the subject. The reference list should support the enticed reader in studying more in depth a topic of interest.

Chapter 2

DESIGN AND VERIFICATION OF DIGITAL SYSTEMS

Before delving 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 2.5) that are required to present the core algorithms involved in verification.

The remaining sections present the algorithms that are at the core of the current technology in design verification. Section 2.6 presents 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 2.7 provides an overview of formal verification and a few of the solutions in this space; we leave the discussion of symbolic simulation and other symbolic techniques to Chapter 3.

2.1 The design flow

Figure 2.1 presents a conceptual design flow from the specifications to the final product. The flow in the figure shows a top-down approach that is very simplified – as we discuss later in this section, the reality of an industrial development is much more complex, involving many iterations through various portions of the flow in the figure, until the final design converges to a form that meets the 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

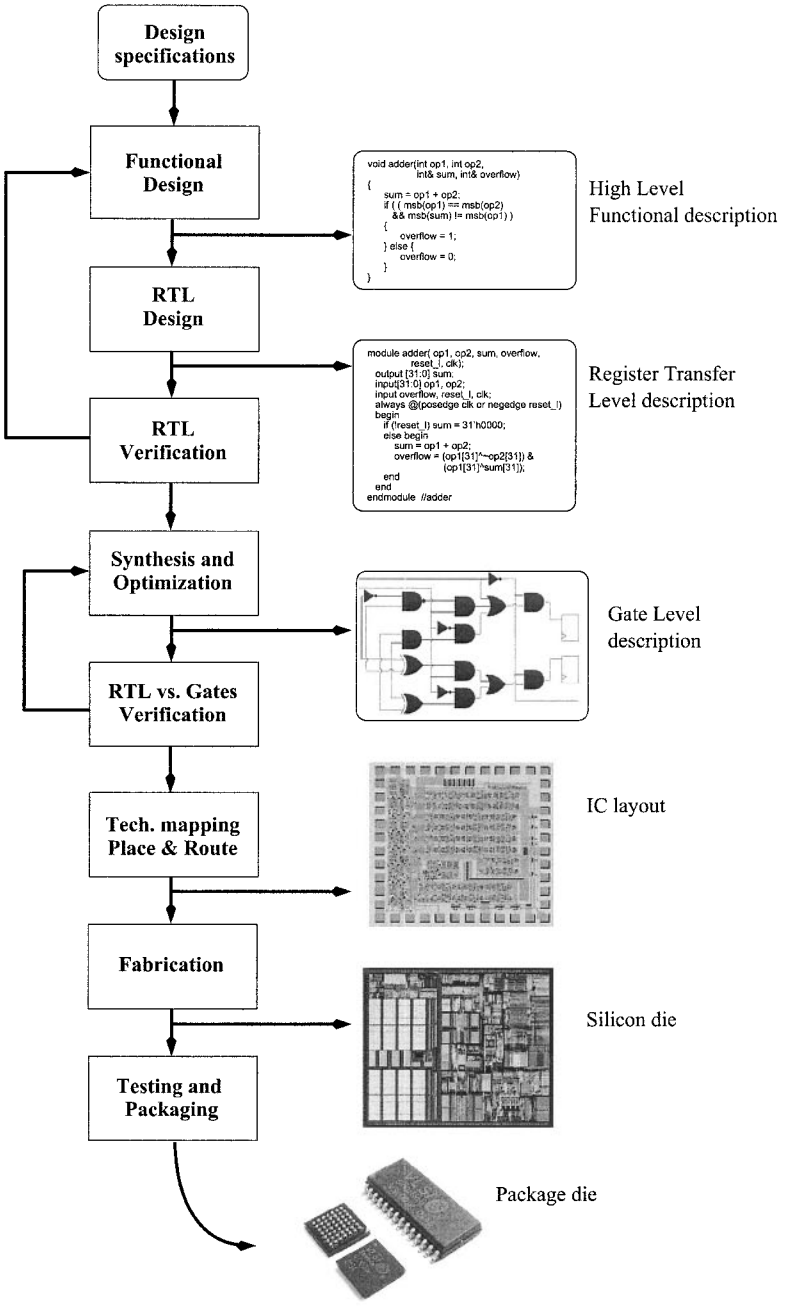


Figure 2.1: Conceptual design flow of a digital system

that it must satisfy. In this context, the *functional design* is the initial process of deriving a potential and realizable solution from these specifications and requirements. This is sometimes referred to as *modeling* and includes such activities as hardware/software tradeoffs and 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 in a 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 elements and functional components of each model are designed using a Hardware Description Language (HDL). This phase also entails the development of the clocking system of the design and architectural trade-offs such as speed and power.

With the RTL design, the functional design of our digital system ends and its verification begins. *RTL verification* consists of acquiring 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 phase of 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 incongruous or overlooked aspects in the original set of specifications and it is found that the specification document is to be updated instead of the RTL description.

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 circuit, 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*, in addition to memory elements. Optimizing the netlist, or gate-level description, for constraints such as timing and power requirements is an increasingly challenging aspect of current developments and it usually involves multiple iterations of trial-and-error attempts before reaching a solution that satisfies the requirements. Such optimizations may, in turn, introduce functional errors that require additional RTL verification.

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. Consequently, they absorb a preponderant fraction of the time and cost involved in developing a digital system. Starting with 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 equivalence 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 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 versions of the design needs to be put again through all of the subsequent phases. One of the main challenges faced by design teams, for instance, is satisfying the ever-increasing market pressure to produce digital systems with better and better performance. These challenging specifications force engineering teams to push the limits of their designs by optimizing them at every level: architectural, component (optimizing library choice and sizing), placement and routing. Achieving timing closure, that is, developing a design that satisfies the timing constraints set in the specifications while still operat-

ing correctly and reliably, most often requires optimizations that go beyond the abilities of automatic synthesis tools and pushes engineers to intervene manually, at least in 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 must devise alternative optimizations or architectural changes and create a new design model that must be put through the complete design flow all over again.

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, are very expensive. At the same time, RTL verification, that is, verifying the correctness of an RTL description, 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 ad-hoc tools developed by design and verification teams specifically for the present design effort. In the best scenarios, the development of this verification infrastructure 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 plan of attack, 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 not only 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, but it is clear why verification is, thus, the bottleneck in the time-to-market odyssey for integrated circuit development [Ber03a].

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 then 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 hardware) 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 of the other modules. 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. These tests also include 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. Recently, a few CAD tools have become available to support functional validation, meaning, they mainly provide more powerful and compact language primitives to describe the test patterns and to check the outputs of the module, thereby saving some test development time [HKM01, HMN01, Ber03a].

During chip-level validation, the design is verified as a whole. Often, this is done after sufficient confidence is obtained regarding the correctness of each single module. The focus is mainly in verifying the proper interaction between modules. This phase, while more computationally intensive, has the advantage of being carried on in a semi-automatic fashion. In fact, input test patterns are often randomly generated, with the only constraint being that they must be compatible with what the specification document defines to be the proper input format for the design. During chip-level validation, it is usually possible to use a golden model for verification. That is, run in parallel the simulation of both the RTL and a high-level description of the design, 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 (this is called lock-step).

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 [KN96, LMUZ02]. Functional validation can provide only partial coverage because of its approach. The objective therefore is 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 common metric is *state coverage*, which measures the number of all the possible configurations of a design that have been simulated (*i.e.* validated). This measure is particularly valuable when an estimate 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 has been verified.

With the increasing complexity of industrial designs, the fraction of design space that the functional validation approach can explore is becoming vanishingly small, indicating 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[1].

[1]The state space of a system doubles for each additional state bit added. Since, as we discussed earlier, the area available doubles every 18 months, and assuming that a fixed fraction of this area is dedicated to memory elements, the overall complexity growth is exponential.

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, therefore, 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 the qualitative sketch in Figure 2.2 shows, with logic simulation we probe the system with a few handcrafted stimuli which are sent through the system and produce an output that must be interpreted in order to establish the correctness of the system for that specific setting. On the other hand, with formal verification the correctness of a design is shown by generating an analytical proof that the system is compatible with each of the properties derived from the specification. Compared to a functional validation approach, this is equivalent to simulating a design with all possible input stimuli, thus providing 100% coverage.

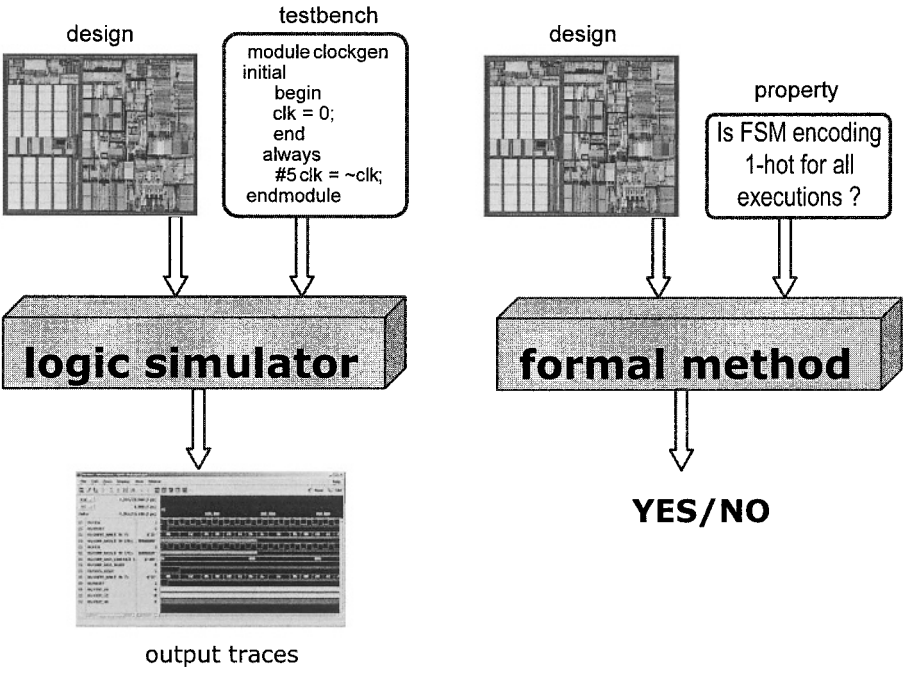


Figure 2.2: Approaches to verification: validation vs. formal verification

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. So far they generally have not been part of the mainstream verification methodology.

The next sections review some of the model abstractions for digital systems in use, in the context of functional verification. We then present, in depth, an algorithm underlying the mainstream approach of functional validation: logic simulation.

2.3 Boolean functions and their representation

Boolean functions are the most common vehicle to describe the functionality of a digital block. We dedicate this section to review a few basic aspects of Boolean algebra and Boolean functions. The concepts outlined here will be referenced throughout the book.

We use the symbol $\mathcal{B}$ to denote the Boolean algebra defined over the set $\{0,1\}$. A **symbolic variable** is a variable defined in $\mathcal{B}$. A **logic function**, or Boolean function, is a mapping $\mathbf{F} : \mathcal{B}^n \rightarrow \mathcal{B}^m$. In the attempt to ease the reading of the theoretical presentations in this book, we use lower-case letters to denote symbolic variables, and upper-case to denote functions. In addition, **scalar** functions, $F(x_1, \cdots, x_n) : \mathcal{B}^n \rightarrow \mathcal{B}$ are represented by regular face literals, while vector-valued functions are represented in boldface. The majority of our presentation will be concerned with scalar functions. The i^{th} component of a vector function $\mathbf{F}$ is indicated by F_i.

An important aspect of a logic function is its **support**, that is, the set of variables the function effectively depends on. For instance the support of the function $F(a,b,c,d) = a + b$ is $\mathcal{S}(F) = \{a,b\}$. To find which variables are in the support of a function, we need to use the concept of cofactor:

Definition 2.1. *The* **1-cofactor** *of a function F with respect to 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.*

By computing the cofactors w.r.t. (with respect to) c for the function used in the previous example, we can easily find that $F_c = F_{\overline{c}} = a + b$. Here is a formal definition of support:

Definition 2.2. *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 iff $F_{x_i} \neq F_{\overline{x}_i}$. 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 vector 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.* $\mathcal{S}(F) \cap \mathcal{S}(G) = \emptyset$. The concept of disjoint support is the core of the presentation in Chapter 4.

Another aspect of Boolean functions which is central to this entire book is **range**, that is, the co-domain spanned by a logic function. Using again our little example, the range of $F = a + b$ is $\{0, 1\}$. When discussing vector functions the concept of range becomes more meaningful, since each output value is a Boolean vector. The notion of range is relevant to our discussion for the following reason: in symbolic simulation, each cycle computes a symbolic vector to represent the next states of the design. This vector is effectively a Boolean function, say **NS**. The next step of simulation transfers this vector to the present state and then proceeds by simulating the combinational logic of the design. However, as we point out again in later chapters, the only relevant information to be transferred among symbolic steps is the range spanned by **NS**, because this range describes the set of states that have been visited by the simulator up to that point. The key advantage of parametrization techniques is that of considering the **NS** vector function and devising an alternative vector, **P**, which will span the same range but will involve smaller functions. This is a formal definition for the range of a function:

Definition 2.3. *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* **F**, *and is denoted by $\mathcal{R}(\mathbf{F})$:*

$$\mathcal{R}(\mathbf{F}) = \{y \in \mathcal{B}^m | \exists x \in \mathcal{B}^n, \mathbf{F}(x) = y\} \tag{2.1}$$

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

A special class of functions that will be used frequently is that of *characteristic functions*. Characteristic functions are scalar functions that represent sets implicitly – they are asserted if and only if their input value belongs to the set represented. Characteristic functions can be used, for instance, to describe implicitly all the states of a system that have been explored by a symbolic technique.

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

$$\chi_V(x) = \begin{cases} 1 & \textit{when} \quad x \in \mathcal{V} \\ 0 & \textit{otherwise} \end{cases} \tag{2.2}$$

For example, the characteristic function of the set $S = \{00, 11\}$ is $\chi_S(x_1, x_2) = x_1x_2 + \overline{x_1}\,\overline{x_2}$. When sets are represented by their characteristic function, the operations of set intersection, union and set complement correspond to the *AND*, *OR* and *NOT* respectively, on their corresponding functions.

We conclude this section by defining two additional operations between Boolean functions. These are function composition and generalization of cofactor and will be useful in the presentation of the disjoint-support decomposition algorithm in Chapter 4. Its application will be discussed in Chapter 6.

Definition 2.5. *Given two functions $F(x_1, \cdots, x_n)$ and $X_i(y_1, \cdots, y_n$, the* **function composition** *$F \circ X_i$ is the function obtained by replacing the function X_i in F for each occurrence of the variable x_i.*

Definition 2.6. *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 the generalized cofactor. Moreover, if F and G have disjoint supports, then one possible solution for F_G is the function F itself.

2.3.1 NP-equivalence

NP-equivalence, or negation-permutation equivalence, is a family of transformations among Boolean functions. They are of interest in the Computer-Aided Design world because two functions that are NP-equivalent can be realized by the same circuit. The implementation of the two NP-equivalent functions will differ only in the mapping of the function's inputs on the inputs of the circuit, and in the need of complementing some of the circuit's inputs. The difficulty, however, lies in identifying when two functions are NP-equivalent. Canonical data structures, such as BDDs (see Section 2.4) provide many benefits by recognizing easily when two functions are identical. On the other hand, two NP-equivalent functions may have completely different BDD representations and can be, therefore, very hard to identify.

An NP-function is a function that can be connected in front of another function F to generate a function G that is NP-equivalent to F [BL92, MD93]:

Definition 2.7. *A function $\mathbf{F}(x_1, \cdots, x_n)$: $\mathcal{B}^n \rightarrow \mathcal{B}^n$ is termed an* **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 $\mathcal{S}(F_i) \cap \mathcal{S}(F_k) = \emptyset, i \neq k$.*

In other words, an NP-function can only permute and/or complement its inputs.

Definition 2.8. *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*

$$F(x_1,\cdots,x_n) = G(\mathbf{NP}(x_1,\cdots,x_n)) \tag{2.3}$$

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

2.4 Binary decision diagrams

Binary Decision Diagrams (BDDs) are a compact and efficient way of representing and manipulating Boolean functions. Because of this, BDDs are a key component of all symbolic techniques of verification. They form a canonical representation, making the testing of functional properties, such as satisfiability and equivalence, straightforward. BDDs are *directed acyclic graphs* that satisfy a few restrictions for canonicity and compactness. BDDs have two terminal nodes, 0 and 1, all other internal nodes are labeled by a symbolic variable and have two outgoing edges: one corresponding to the 0-cofactor w.r.t. the variable labeling the node, and one corresponding to the 1-cofactor. Each path from root to leaves, in the graph, corresponds to an evaluation of the Boolean function for a specific assignment of its input variables. An important factor in the success of BDDs is the ease of manipulating Boolean functions through this representation: the *apply operation* is implemented by a simple recursive function which traverses one or more BDDs (the operands) bottom-up and applies a specified Boolean operator at each intermediate node. We provide here a brief presentation. The interested reader is referred to [Bry86, Bry92] for an in-depth introduction and an overview of their applications.

Example 2.1. *Figure 2.3.a represents the BDD for the function* $F = (\overline{x}+\overline{y})pq$. *Given any assignment for 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 if the corresponding variable has a value* 0*, the* 1*-edge otherwise.*

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.*

For a given ordering of the variables, it was shown in [Bry86] that a function has a unique BDD representation. Therefore, checking the identity of two functions corresponds to checking for BDD identity, which is accomplished in constant time. The following definition formalizes the structure of BDDs:

Definition 2.9. *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*

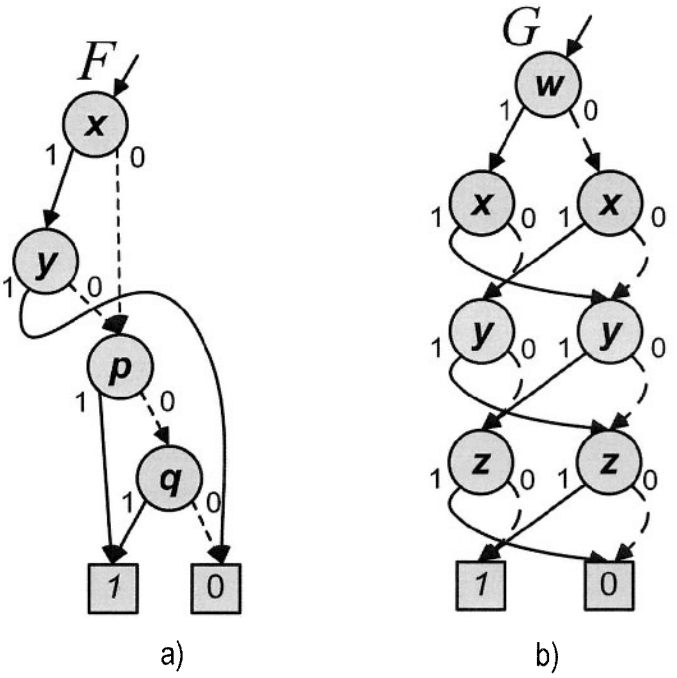


Figure 2.3: Examples of binary decision diagrams

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. 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* [BRB90]. 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 may 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. Because of its impact, much research has been devoted to finding algorithms that can provide a good variable order. While finding the optimal order is an intractable problem, many heuristics have been suggested that find sufficiently good orders, from static approaches based on the underlying logic network structure in [MWBSV88, FFK88], to dynamic techniques that change the variable order whenever the size of the BDD grows beyond a threshold [Rud93, BLW95].

Moreover, much research work has been dedicated to the investigation of alternative representations to BDDs. For instance BMDs [BC01] target the representation of circuit with multiplicative cores, Zero-Suppressed BDDs [Min93] are suitable for the representation of sets, and MTBDDs [FMY97] can represent multi-valued functions. An example application which uses MTBDDs is presented in Chapter 5.

Binary decision diagrams are used extensively in symbolic simulation. The most critical drawback of this method is its high demand on memory resources, which are mostly used for BDD representation and manipulation. This book discusses recent techniques that transform the Boolean functions involved in symbolic simulations through parametrization and approximation. The objective of parametrization is to generate 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 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.

2.5 Models for design verification

The verification techniques that we present in this book 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 latches or banks of registers. Such circuits store state information about the system. Hence, 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.

2.5.1 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 that we use are: *AND*, *OR*, *NOT* or *XOR*. Figure 2.4 reproduces the graphic symbol for each of these types.

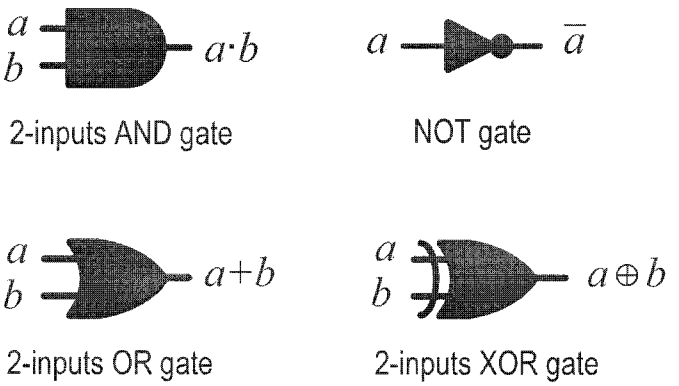


Figure 2.4: Graphic symbols for basic logic gates

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, also called netlist.

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 three bits representing the current value of the counter. The clock input is assumed implicitly, and not represented in the figure. This system has four memory elements that store the current counter value and control 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. Subsequently, it is decremented down to zero. Whenever the reset signal is held high, the counter is reset to zero.*

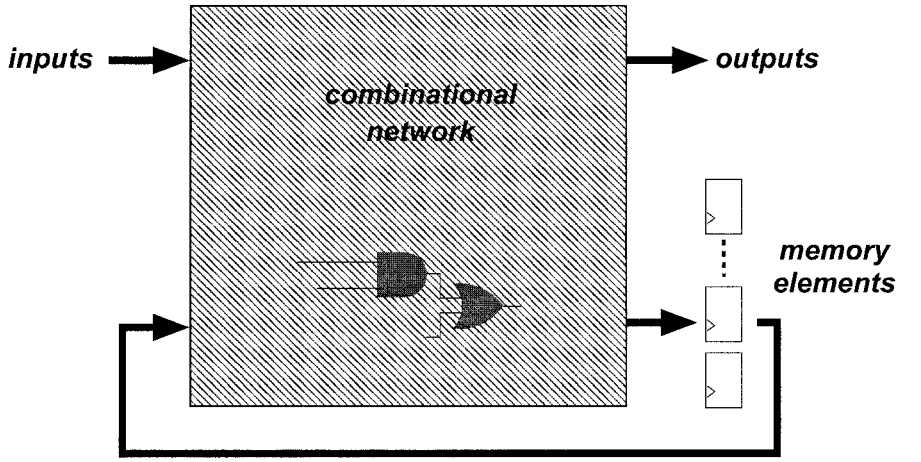


Figure 2.5: Structural network model schematic

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

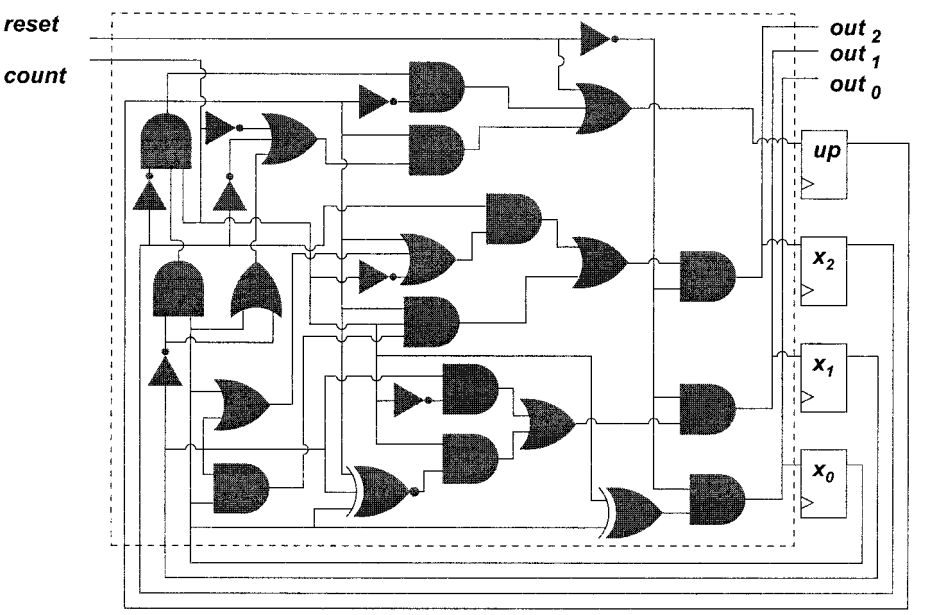


Figure 2.6: Network model of a 3-bits up/down counter with reset

2.5.2 State diagrams

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

Such a 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 combinations which would cause the transition in a single clock cycle. State diagrams present only the functionality of the design, while the details of the implementation are not considered. 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.*

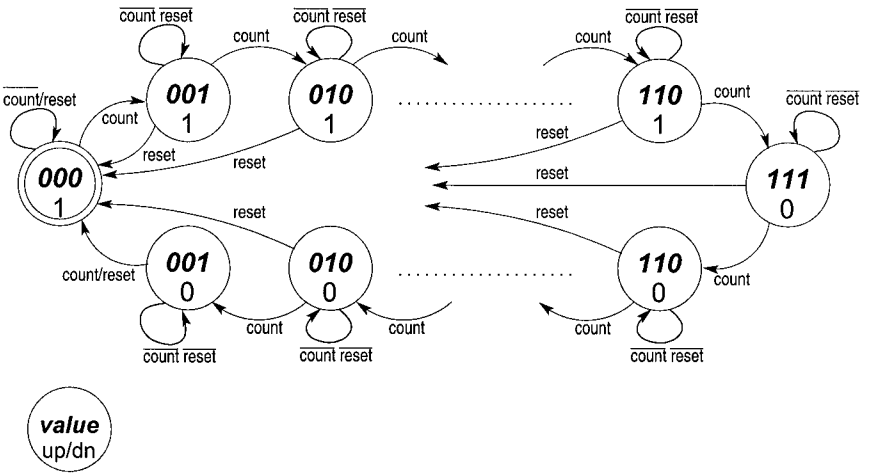


Figure 2.7: State diagram for a 3-bits up/down counter

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

Example 2.4. *Figure 2.8 represents the finite state machine for 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.*

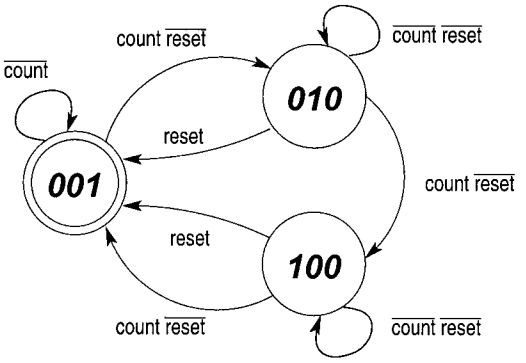


Figure 2.8: State diagram for a 1-hot encoded 3-bits counter

2.5.3 Mathematical model of finite state machines

An alternative way of describing a Finite State Machine (FSM) 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 is defined by a 6-tuple:

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

where:

- I is an ordered set $(i_1, \ldots, i_m)$ 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 means of building 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.

2.6 Functional validation

This section is dedicated to the presentation of an algorithm for cycle-based logic simulation, a mainstream technique in functional validation. The next section outlines some of the techniques used in formal verification.

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 [BCRR87, Han88, WHPZ87].

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 of the gates in "levels". 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 instructions' inputs are ready when the program counter reaches a specific instruction. This assembly block constitutes the internal representation of the circuit in the 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*

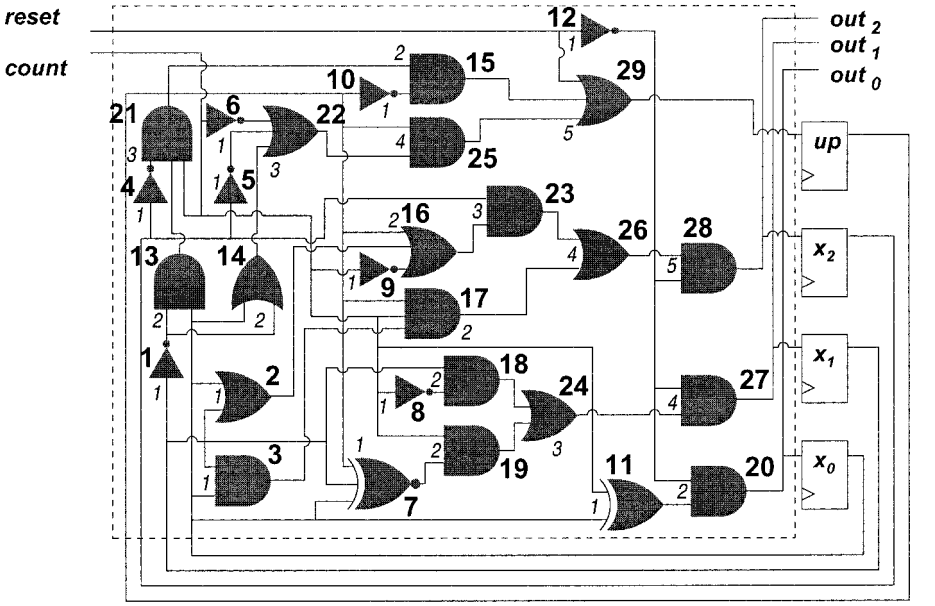


Figure 2.9: Compiled logic simulator

number (italicized in the graphic) based on its distance from the inputs of the design. Subsequently, gates have been numbered sequentially (gates numbers are in boldface), in a way compatible with this partial order. From this diagram it is possible to write the corresponding assembly block:

Note, that there is a one-to-one 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 input 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. Figure 2.10 shows an outline of the algorithm, where the core loop includes the assembly code generated from the network.

```
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: Pseudo-code for a cycle-based logic simulator

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.

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 also facilitates simulating 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 [DeV97]. When deployed in a digital-system development context, simulators constitute the core engine of the functional validation process. However, the bulk of the time spent in verification is in the development of meaningful test sequences. 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 then generate procedures to automatically check the correctness of the output values [HKM01, KOW+01]. 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. Coverage engines collect analytical data on the portions of the circuit that have 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 [AGL+95, CIJ+95, YSP+99]. 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 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 just producing many 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 [Pfi82, Hau95, CMA02]. While emulators can perform one to two orders of magnitude faster than software-based simulators, they constitute a very expensive solution. It is expensive because of the high raw cost of acquisition and the time-consuming process of configuring them for a specific design, which usually requires several weeks of engineering effort. Because of these reasons, emulators are mostly used for IC designs with a large market.

Even if design houses put forth great 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

can, therefore, potentially miss subtle design errors that might only surface under particular sets of rare conditions.

2.7 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 machine's representations. The main successful results of these methods are based on *symbolic state traversal* algorithms which allow the full exploration of digital systems, although with very limited scalability. Typical design sizes that can be handle by these solutions are up to a few hundreds latches, which is far from what is needed in an industrial context. 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*symbolic state traversal,state explosion problem and it is the main reason for the very limited applicability of the method. At the same time, the approach has the advantage of being fully automatic. A variant within this family is *bounded model checking*, which allows for the handling of much more complex systems. Although, as the name suggests, it has a limited (or bounded) depth of analysis.

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. The next chapter is dedicated to discuss in depth symbolic simulation and a range of related solutions.

Symbolic approaches are also at the base of *equivalence 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 heuristic solutions that are scalable to industrial-size circuits, thereby achieving full industrial acceptance. The success of scalable symbolic solutions in the domain of equivalence checking, gives hope that symbolic techniques will be also the basis for viable industrial-level solutions for formal verification.

A different family of approaches, *proof-theoretic methods*, are based on abstractions and hierarchical techniques aimed at proving the correctness of a system [Hue02, Joh01]. Verification within this framework uses theorem-proving software to provide support in reasoning and deriving proofs about the specifications and the implementation model of a design. They use a variety of logic representations, called theories. 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.

We conclude the section by describing in more detail one of the techniques outlined above, to give the reader a sense of the computational procedures involved in formal verification. Symbolic simulation techniques will be described in the next chapter.

2.7.1 Symbolic finite state machine traversal

One approach used in formal verification is to focus on a property of a circuit and to 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 microprocessors, one property could be that any issued instruction completes within a finite number of clock cycles. The proof of properties such as these, requires, 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. A *reachable state* is just a state 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*. The approach, described below, has been published in seminal papers by Madre, Coudert and Berthet [CBM89] and later in [TSL+90, BCL+94].

In the context of FSMs, reachable-state computations are based on implicit traversal of the state diagram (Section 2.5.2). 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 of these states. The image of the set* $\{110-1,111-0\}$ *is* $\{000-1,110-1,111-0,110-0\}$.

The following definition formalizes the operation of image computation:

Definition 2.10. *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 2.5.3, 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 is asserted when there is some input i such that $\delta(s,x) = s'$. This relation is defined by existentially quantifying the inputs from $\delta()$:

$$TR(s,s') = \exists i \left[\bigwedge_{k=1}^{n} \delta_k(s,i) \equiv s'_k \right] \tag{2.4}$$

where δ_k represents the transition function for the k-th bit. As it could be imagined, the transition relation can be represented by a corresponding characteristic function – see Definition 2.4 – χ_{TR} which equals 1 when $TR(s,s')$ holds true.

Finally, the image of a pair $\langle \mathcal{M},R \rangle$ can be defined 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 the characteristic function:

$$\chi_{Img}(s') = \exists s \ \ (\chi_{TR}(s,s') \cdot \chi_R(s)) \tag{2.5}$$

Symbolic FSM traversal performs image computations iteratively starting from the initial state. At each steps it accumulates the states visited (that is, the images) into a *reached* set. The traversal ends when a fixed point is found, which is detected when the reached set does not grow from iteration to iteration. At the end of the computation, the reached set represents the characteristic function of all the states that can be reached by the system, and it can be used to prove properties specified over the states of the design. The fixed point computation of symbolic FSM traversal will be discussed in more detail in Section 3.4.1.

2.8 Summary

This chapter presented an overview of the design and verification flow involved in the development of a digital integrated circuit. It discussed the main techniques used to verify such circuits, namely functional validation (by means of logic simulation) and formal verification, using a range of techniques.

We reviewed basic concepts and representations for Boolean functions and for sequential systems, and described how a logic simulator works. The models discussed in the earlier sections will be needed to present all the main techniques in the later chapters. The last part of the chapter was dedicated to skim over a range of formal verification techniques, and give a sense of this methodology through the presentation of symbolic FSM traversal. The next chapter covers in great detail another technique, symbolic simulation, and draws the similarities between that and reachability analysis.

References

[AGL+95] 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.

[BC01] Randal E. Bryant and Yirng-An Chen. Verification of arithmetic circuits using binary moment diagrams. *International Journal on Software Tools for Technology Transfer*, 3(2):137–155, 2001.

[BCL+94] Jerry R. Burch, Edward M. Clarke, David E. Long, Ken L. McMillan, 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.

[BCRR87] 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.

[Ber03] Janick Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2nd edition, 2003.

[BL92] 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.

[BLW95] 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.

[BRB90] 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.

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

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary–decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[CBM89] 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.

[CIJ+95] 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.

[CMA02] 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.

[DeV97] 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, November 1997.

[FFK88] 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.

[FMY97] Masahiro Fujita, Patrick McGeer, and Jerry Yang. Multi-terminal binary decision diagrams: An efficient datastructure for matrix representation. *Formal Methods in System Design*, 10(2-3):149–169, 1997.

[Han88] Craig Hansen. Hardware logic simulation by compilation. In *DAC, Proceedings of Design Automation Conference*, pages 712–716, June 1988.

[Hau95] Scott Hauck. *Multi-FPGA Systems*. PhD thesis, University of Washington, Dept. of Computer Science and Engineering, 1995.

[HKM01] Faisal I. Haque, Khizar A. Khan, and Jonathan Michelson. *The Art of Verification with Vera*. Verification Central, 2001.

[HMN01] 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.

[Hue02] 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.

[Joh01] 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.

[KN96] 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.

[KOW+01] 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.

[LMUZ02] 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.

[MD93] 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.

[Min93] S.-I. Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *DAC, Proceedings of Design Automation Conference*, pages 272–277, June 1993.

[MWBSV88] 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.

[Pfi82] Gregory F. Pfister. The yorktown simulation engine: Introduction. In *DAC, Proceedings of Design Automation Conference*, pages 51–54, January 1982.

[Rud93] 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.

[TSL+90] 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.

[WHPZ87] 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.

[YSP+99] 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.

Chapter 3

SYMBOLIC SIMULATION

This chapter introduces symbolic simulation, a verification technique that has been in use for the past 25 years. We overview here its evolution, the baseline algorithms that fall under the symbolic simulation umbrella and a range of techniques that have been explored to improve performance and its ability to tackle complex designs.

3.1 The origins of symbolic simulation

Symbolic simulation was first explored in the late 1970's as an answer to the increasing inadequacy of logic simulators to explore and verify all the possible execution paths in digital designs. Different decade, same problem of today. Designs were reaching sizes comprised of tens of thousands of transistors, simulating hosts could only run at a few Megahertz, and the time required by functional verification was becoming a burden.

The key idea of symbolic simulation consists in the use of mathematical techniques to represent symbolically the logic values at internal nodes of a digital circuit during its simulation. A preliminary work in this area is by King in [Kin76], where he proposes a method of symbolic execution to verify software programs. One of the first hardware symbolic simulators was developed by IBM in 1979 [CJB79] to verify instruction micro-code in their processor designs. The main idea was to provide a simulator system where symbolic variables could be mixed with constant values at the inputs of the design under test. Symbolic expressions would then be propagated through the design schematic and the expressions obtained at the output would be evaluated for correctness by means of theorem proving techniques. However, this first symbolic simulator was lacking some important features of modern tools: it could only handle combinational circuits, algebraic expressions could not rely on a compact data-structure for their representation (BDDs were not discovered,

yet), and algebraic minimization capabilities were, well, at a minimum, leaving the bulk of the verification work to theorem proving. Because of the lack of specialized techniques to simplify expressions, the IBM simulator could handle both bit-level and word-level symbolic variables uniformly.

The first pre-cursor to modern symbolic simulators is MOSSYM, a software package developed by Randy Bryant in the mid 1980's [Bry85]. MOSSYM was the first tool to include a Boolean-expression manipulation package, a precursor of the Binary Decision Diagrams (BDDs) – see Section 2.4. Because of this, it also required symbolic variables to be associated with single-bit signals, so that the resulting expressions could be handled by the Boolean library. In 1987, the ideas of MOSSYM carried on to COSMOS [BBB+87], a symbolic simulator for CMOS designs. COSMOS uses full-fledged BDDs as the underlying Boolean expression package to represent and manipulate the symbolic expressions associated with the nodes of the circuit. In addition, initial techniques to simulate specialized types of sequential circuits, such as pipelined designs, are explored in the same period by [BF89, BBS90].

3.2 Symbolic simulation of a logic gate

As described in Section 2.6, a logic simulator uses a gate-level representation of a circuit and performs the simulation by manipulating the Boolean scalar values, 0 and 1. Symbolic simulation differs from logic simulation because it builds Boolean expressions rather than scalar values, as a result of circuit simulation.

Figure 3.1: Comparison of logic and symbolic simulation

Consider the two *OR* gates in Figure 3.1. 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 variables a and b, and the result placed at the output node is the Boolean expression $a+b$. By composing the expressions generated at the output of simple gates, it is possible to compute the functionality of group of gates, as shown with two pictorial examples in Figure 3.2. By extension, when a distinct Boolean symbol is assigned to each input net of a gate-level netlist, the functionality of the entire combinational logic portion can be computed.

This approach is very powerful in two ways. First, at the completion of the symbolic simulation, we have a Boolean expression that represents the full

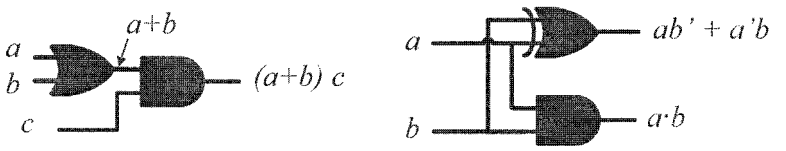


Figure 3.2: Simulation of a netlist by composition of symbolic expressions

functionality of the circuit (in this example, two tiny two-gate circuits). This expression can be compared and verified against a formal specification of the desired outputs. Expressions obtained by symbolic simulation become quickly very complex. Hence, the comparison is only feasible for very small designs. In addition, a great deal of effort is spent, and has been spent, in devising techniques to represent the expressions compactly and keeping them in simple canonical form.

Second, symbolic simulation can be seen as a way of executing multiple logic simulations in parallel, one for each possible assignment of the symbolic variables. The number of equivalent logic simulations is given by 2^n where n is the number of symbolic variables introduced. Because values are propagated symbolically, the expressions obtained at the circuits' outputs encode the output values under all possible assignments. For instance, the symbolic simulation of Figure 3.1 is implicitly applying four test vectors in parallel, corresponding to $\{a = 0, b = 0\}$, $\{a = 1, b = 0\}$, $\{a = 0, b = 1\}$ and $\{a = 1, b = 1\}$. This aspect of symbolic simulation is also interesting because it can be easily integrated in a logic simulation methodology where the amount of parallelism in the input test vector can be tuned by substituting constant values with Boolean variables and *vice versa*, based on the resources available in the simulating host.

3.3 Symbolic simulation, time frame-by-time frame

We now describe a full-fledged symbol simulation algorithm. The frame-by-frame approach is a mainstream technique to symbolically simulate sequential circuits. We use this algorithm as the baseline for the discussion of related techniques, performance improvements and more scalable variations that are the topic of the subsequent sections and chapters of this book. Because it is our reference technique, when we use the term *symbolic simulation* in this book, we are referring to this frame-by-frame algorithm.

In frame-based simulation, the state space of a synchronous circuit is explored iteratively, with reference to a gate-level description of the digital design. At each step of simulation, a distinct Boolean expression is assigned to each input signal and present-state signal. They can vary from complex expressions to extremely simple ones, 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 2.6, 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.

With reference to Figure 3.3, the algorithm operates as follows: At time step 0, the gate-level network model is initialized with the initial assignment S_0 for 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. 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 a levelized order compatible with their distance from the input nodes, similarly to what is done in compiled-level logic simulation (see Section 2.6). At the end of each step, the Boolean expressions obtained for the primary outputs are used to evaluate the correctness of the design (for instance by checking against a bounded property). The expressions computed for the memory elements' inputs are fed back to the state inputs of the circuit, and the next step of simulation starts.

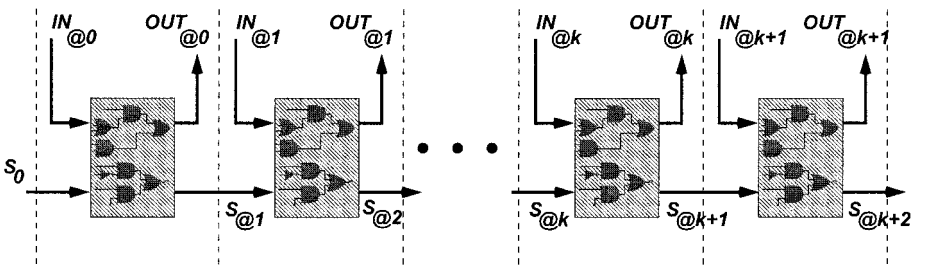


Figure 3.3: Schematic of the iterative model of symbolic simulation

Example 3.1. *Assume that 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* $S_0 = \{000 - 1\}$ *as shown in Figure 2.7. We then use two symbolic variables* r_0 *and* c_0 *for the set* $IN_{@0}$ *on the two input lines.*

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

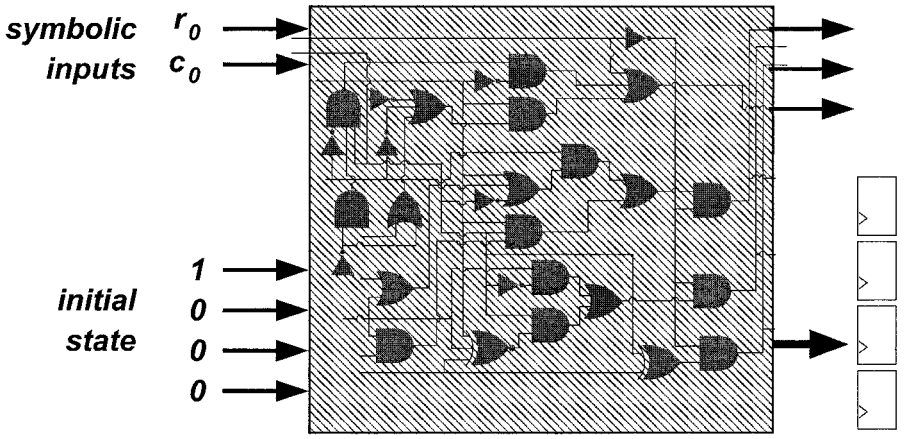


Figure 3.4: Symbolic simulation for Example 3.1 - Initialization phase

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 the primary outputs and the flip-flops' 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 $IN_{@1}$ as suggested by Figure 3.5. At completion of the second simulation step, we obtain the following expressions:

$$\begin{aligned} out_0 &= x_0 = ((\overline{r_0}c_0) \oplus c_1)\overline{r_1} \\ out_1 &= x_1 = \overline{r_0}r_1c_0c_1 \\ out_2 &= x_2 = 0 \\ up &= 1 \end{aligned}$$

Notice that new Boolean variables are created at every simulation step, one for each of the primary inputs of the network. Thus, the expression obtained for the primary outputs and the state signals at the end of each step k will be

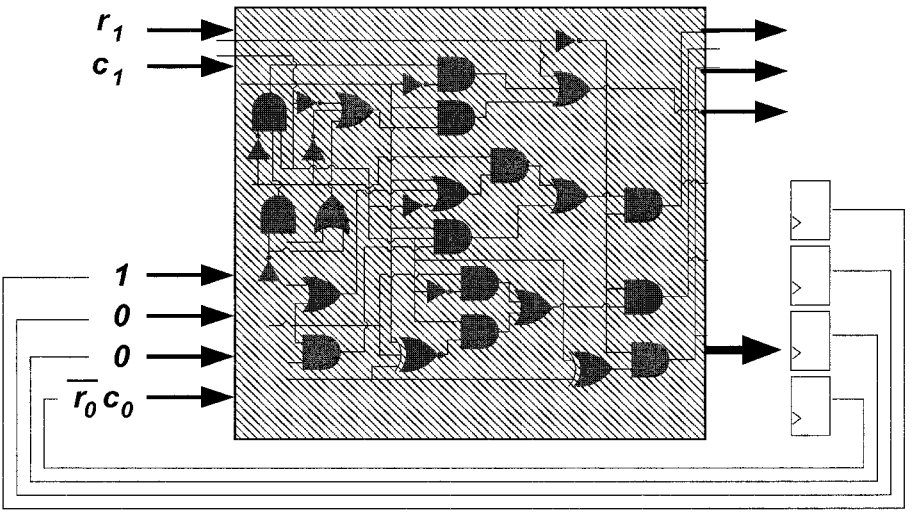


Figure 3.5: Symbolic simulation for Example 3.1 - Simulation Step 2

functions of variables in $\{IN_{@0}, \cdots, IN_{@k}\}$. The vector of Boolean functions obtained for the state symbols $\mathbf{ST_{@k}} : \mathcal{B}^{mk} \rightarrow \mathcal{B}^n$ represents all the states that can be visited by the circuit at step k. The state symbols $\mathbf{ST_{@k}}$ 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. *Vice versa*, each state that is k steps away from the initial state and corresponds to at least one evaluation of the symbolic variables. Note also, from the example, how the expressions involved in the simulation become increasingly complex at each time step.

The procedure just described is equivalent to propagating the symbolic expressions through a time-unrolled version of the circuit,symbolic simulation,time-unrolled circuit where the combinational portion is duplicated as many times as there are simulation steps. Figure 3.3 shows this iterative model and input and output nodes for the symbolic expressions.

A simple pseudo-code of the verification algorithm just described is shown in Figure 3.6.

3.3.1 Symbolic simulation to expose design flaws

By running a frame-based symbolic simulation, we generate a Boolean expression for each output and each simulation step. If some of those outputs represent the outcome of checkers or assertions embedded in the system, we

```
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 3.6: Pseudo-code for frame-by-frame symbolic simulation

can then use these expressions to check if we exposed a design error, and also to produce a trace (Boolean test vector) that leads the system from an initial state to the flawed configuration. Such trace can then be executed by a logic simulator and used to debug the design. If no error is exposed by the output expressions, then the simulator proceeds to the next step, searching for bugs embedded deeper in the design's state space.

Specifically, at each step, the functions $\mathbf{OUT}_{@\mathbf{k}} : \{in_{@0}, \cdots, in_{@k}\} \rightarrow \mathcal{B}^p$ represent a set of legal values for the outputs of the circuit. If, for instance, one of these outputs corresponds to the outcome of a checker that we expect to be always at 1 under correct functionality, then 1) a bug is exposed if we see that the corresponding $\mathbf{OUT}_{@\mathbf{k}i}$ expression differs from the constant value 1, and 2) a trace can be generated by finding an assignment for the inputs of the expression $\mathbf{OUT}_{@\mathbf{k}i}$ so that it evaluates to the desired value.

In the most general case, if the expected output vector at time k is given by the Boolean vector $C \in \mathcal{B}^p$, all the assignments that falsify the expression *Valid*:

$$Valid = \bigwedge_{i=1}^{p} (\mathbf{OUT}_{@\mathbf{k},i} = C_i) \tag{3.1}$$

are valid test sequences that expose the design error. Once *Valid* is computed, it can either be a constant 1 expression, or it is simple to find a falsifying expression by just traversing its BDD representation from the top down until

a 0 leaf is reached. This approach can be easily generalized to verify more complex properties where the output signals must satisfy complex relations expressed in terms of the input symbols.

3.4 Close relatives of symbolic simulation

The algorithm just described plays a key role in many formal verification techniques. We briefly overview here two important formal verification techniques that use symbolic simulation: 1) In reachability analysis, the transition relation of the circuit is typically computed using symbolic simulation, and 2) symbolic simulation is also the underlying engine of symbolic trajectory evaluation. The next section will outline some of the main improvements proposed for symbolic simulation.

Symbolic reachability analysis uses symbolic computation to generate an implicit function representing of all the reachable states of a finite state machine. Once the reachable state set is known it is straightforward to verify if a property holds by simply validating it for all the reachable states. A cornerstone in the evolution of reachability analysis was the work by Coudert and Madre in [CBM89], where the authors discovered a technique to compute the reached set using the same explicit Boolean expressions that are used in symbolic simulation, making reachability analysis a more scalable solution. Their technique can also be seen as a way of reparametrizing the symbolic output functions, a very fruitful research direction for symbolic simulation, as we will see in the later sections.

The other technique that we cover here is *Symbolic Trajectory Evaluation* (STE). STE is a way of focusing symbolic simulation on the verification of a specific property, by restricting the symbolic test vector to specify only the input sequences of interest. STE can tackle more complex problems (in terms of design size), especially when the property to be verified requires a very narrow (very specific) stimulus.

3.4.1 Symbolic reachability analysis

The objective of symbolic reachability analysis is to determine the set of states that a system can reach after an arbitrary number of transitions, starting from an initial state (or set of states) S_0. The system is described by a finite state machine, and the mathematical model of the FSM is used to run the analysis. The key step of reachability analysis is the operation of image computation, as we described it in the previous chapter, Section 2.7.1, that is the operation of computing all the states that can be reached in one transition, starting from a start state, under any possible input.

The set of reachable states can be computed by a *symbolic breadth-first traversal* where all operations are performed with characteristic functions. Dur-

ing 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 included 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.

```
Symbolic_Reachability (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 3.7: Pseudo-code for symbolic reachability analysis

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 reachability analysis is at the core of the *symbolic model checking* technique for verification. The basic idea underlying this method is to use BDDs (Section 2.4) to represent all the functions involved in the process and the set of states that have been visited during the exploration. Once again, 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 challenges is still the subject of intense research, in particular, various solutions have been proposed that try to contain the size of the BDDs involved [RS95, CCQ96] and to reduce the complexity of performing the image computation operation [CCLQ97, MKRS00]. Recently, Goel and Bryant [GB03] proposed a technique to simplify the complexity of computing

the reached set of states from the symbolic state vector, by defining set operations directly on the vector, obtaining a more complex representation of the reached set and bypassing complex and time-consuming BDD operations.

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 not trivial to construct an input trace that exposes it.

3.4.2 Symbolic trajectory evaluation

Symbolic Trajectory Evaluation, or STE, strives to improve the scalability of *symbolic model checking* by restricting how properties are described, so that a frame-based symbolic simulator can be used, instead of performing full-blown reachability analysis.

In symbolic trajectory evaluation [SB95], the property to be checked is restricted to be expressible in the form $A \Rightarrow C$. This type of properties states that whenever the design behavior matches a pattern described by A, then it must also satisfy the pattern specified by C. The formulas A and C describe the desired behavior by specifying the values of circuit nodes for a finite number of clock cycles into the future. However, only the node values that are relevant for the property condition or assertion must be specified. These types of formulas are called "trajectory formulas". Although trajectory formulas seem difficult to express, STE software typically provides features to ease the laboriousness of this task. More importantly, trajectory formulas lack the expressivity of other property specification languages (*e.g.*, CTL). For instance, it is not possible to describe a trajectory as the negation of a pattern, nor as the disjunction of two trajectories; it is also not possible to describe an event that may happen "eventually" (with the meaning of CTL) in the future, since trajectories describe events over a finite interval of time (this last limitation has been overcome with the introduction of "generalized STE" in [YS03]).

These limitations are offset by an important benefit: all the circuit behavior described in a trajectory formula can be expressed by a unique symbolic simulation vector, where at each cycle the input signals are assigned to 0, 1, a symbolic variable, or the value 'X' to indicate 'unspecified'. Hence, a trajectory formula can be verified with a single run of symbolic simulation by symbolically simulating the vector for A, and after each simulation cycle, by checking that the circuit state is consistent with the corresponding part of C.

It can be noted that this approach is usually much faster than traditional model checking, which requires the computation of a fixpoint for reachability analysis. The downside is the limited expressiveness of trajectory formulas discussed previously. In addition, this technique may suffer the same problems of traditional symbolic simulation in terms of performance and memory resource demands.

3.5 Enhancements and optimizations

While theoretically a symbolic simulation can proceed indefinitely, practically the representation of the Boolean expressions involved will eventually require memory resources beyond those available in the simulation host. Moreover, for the more complex designs, the complexity of the Boolean expressions impacts, very early on, the amount of memory resources needed to proceed. The consequence is that both the poor performance and the low scalability of plain symbolic simulation are a bottleneck for this verification solution.

Many techniques to improve the applicability of symbolic simulation have been proposed in the past fifteen years. A few of them are briefly outlined here below, mainly to give a flavor of how broad and diversified this research has been. The next chapters are dedicated to cover, in depth, a few solutions that make aggressive use of both reparametrization and approximation, in order to show the effectiveness of these techniques in making symbolic simulation a scalable verification approach.

In the area of microprocessor verification, Velev, *et al.* [VBJ97] presented a technique to symbolically simulate memories. Memories are common in microprocessor designs and present an added difficulty for a symbolic simulator because it needs to store a distinct symbolic function for each storage bit of a memory. In Velev's solution a memory is represented by a black box. Write accesses are tracked through a queue storing symbolic addresses and data in proper time order. Read accesses compute the proper data function by composing the superimposition effect of all the write operations. In general, this technique makes symbolic simulation more amenable for systems with large memory blocks, since the complexity is proportional to the number of write operations, as opposed to the size of the memory. It is true that read operations over time could entail very long computations, because they require to process all the write accesses to a portion of the memory (as specified by the symbolic address of the operation). However, a baseline symbolic simulator involves more and more complex symbolic functions over time, too. The Velev approach has the advantage that the first few accesses are very simple to process by the simulator, and the complexity only grows over time, with the growing number of read accesses. To address the verification of microprocessors, Ritter, *et al.* proposed a technique that combines hierarchical equivalence checking with symbolic simulation to prove the equivalence between the specification and an implementation of a microarchitecture [REH99]. Jones [AJS99] also dedicated much work to the verification of microprocessors, and addressed the problem by using hybrid techniques and parametrization. We will cover some of this work in Chapter 6.

Kölbl, *et al.* proposed a method to deploy symbolic simulation in the context of event-driven simulation [KKD01]. To this end, simulation time is also handled symbolically, and simulation advances by processing a queue of sym-

bolic events. The attractiveness of this solution lies in the ability to simulate a design that has not been synthesized, hence, this formal technique can be applied in the very early stages of system development. However, it is an added level of complexity in the use of symbolic time. As a result, all of the different possible ordering among simulation events must be considered and evaluated. The authors present a solution to merge events by identifying when they occur at the same time, however, in general the sequencing of the events is a source of considerable complexity.

Techniques that directly address the performance of simulation include the work of Want *at al.*, [WCZK01], where the authors attack the problem by simulating the circuit's gates in an order that minimizes the overall size of the BDDs in use at any point in time. The technique uses a *min-cut linear arrangement* graph algorithm adjusted to take into account the topological constraints of the circuit. In addition, the graph edges are weighted by an estimate of the BDD size at each internal circuit node. This estimate can be derived from the size at the previous simulation step, and does not keep track of the sharing potential among BDDs. Another example is [HSH+00], where an attempt is made to overcome the limitations of plain symbolic simulation by parallelizing it with other collaborative engines: symbolic simulation interacts with logic simulation to achieve higher coverage within boundaries of time and memory usage. Simultaneously, symbolic state traversal is used together with abstraction techniques to prune unreachable portions of the design's state space, and thus simplify the simulator's work. The result is an integrated software tool that supports the designer in "classifying" the state space of the IC design into reachable and unreachable sets, and produces efficient and compact tests to visit the relevant portions of a design.

Earlier in this chapter we mentioned that symbolic simulation can be viewed as running an exponential number of logic simulators in parallel: this gives symbolic simulation a notable advantage. A fraction of this advantage could be traded to gain scalability (through simpler Boolean expressions) and obtain viable and efficient symbolic solutions for functional verification. For instance, in [WD00], the trade-off is between the breadth of the search and a better ability to simulate complex designs. Wilson, *et al.* present 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 step is then re-simulated with the other constant value for each of the simplified symbolic variables, until complete expressions are obtained for the outputs of the network. Another technique that we developed [BDQ99] makes a hybrid use of reparametrization and approximation: at each step of simulation it reparametrizesreparametrization the state space by deploying a technique that constrains the parametric functions to be

extremely compact. If a solution under this constraint is not viable, the state set is under-approximated to what is representable by the parameters. Both these solutions are covered in detail in Chapter 5.

Chapter 6 is dedicated to reparametrization solutions and it presents the work by Jones in relation to microprocessor verification [AJS99]. The chapter also introduces a technique we developed in this research space, which makes use of disjoint-support decompositions to generate an efficient and exact parametrization [BO02].

The research overviewed in this section is just a small sample of the work that has been developed in this area. While we did not attempt to provide an exhaustive presentation, the objective was to show the broad range of enhancements proposed.

3.6 The challenge in symbolic simulation

Even if some of these efforts provide a major contribution in making symbolic simulation much more attractive for use in industrial settings, the functional verification of digital systems remains a challenge for every hardware engineering team. There are two core observations underlying the solutions that we discuss in the rest of this book. First, symbolic simulation traverses the states of a digital system carrying forward, at each simulation step, much more information than 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 step and how they relate to the symbolic input variables. However, much less information is actually needed to achieve the objective of verification, that is, having the ability to identify, globally, which states are visited at each step. In fact, it is possible to re-encode the relevant information in more compact parametric forms. This observation leads us to explore parametric techniques to generate effective encodings.

The second observation is that when attempting to expose a bug, only a small portion of the signals during a simulation are relevant, while the baseline frame-based symbolic simulation algorithm outlined above computes exact Boolean expressions for all intermediate nodes. If we could select the correct portion of the design up front, the remaining part could be approximated and it would be possible to save a lot of computational work.

The next chapters discuss solutions and theoretical work to restrict the computational effort to only the relevant part of the simulation, and also to discover new and efficient encodings of the state sets involved in simulation. These new encodings, or parametrizations, have representations that are more compact than the original ones, and thus allow for a memory efficient symbolic simulation, that presents much better robustness and scalability characteristics.

3.7 Summary

This chapter covered symbolic simulation, its evolution and the baseline algorithm of reference. We also covered other related symbolic techniques and drew the analogy to simulation. Finally the last part of the chapter was dedicated to discuss some of the ongoing research in this area.

The next chapter will take a step back from the verification problem, and focus on introducing parametrizations and disjoint-support decompositions. Both these theoretical aspects will be deployed in the symbolic simulation solutions presented later in this book.

References

[AJS99] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Formal verification using parametric representations of Boolean constraints. In *DAC, Proceedings of Design Automation Conference*, pages 402–407, June 1999.

[BBB+87] 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.

[BBS90] Derek L. Beatty, Randall E. Bryant, and Carl-Johann H. Seger. Synchronous circuit verification by symbolic simulation: An illustration. In *Proceedings of Sixth MIT Conference on Advanced Research in VLSI*, pages 98–112, 1990.

[BDQ99] 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.

[BF89] Soumitra Bose and Allan Fisher. Verifying pipelined hardware using symbolic logic simulation. In *ICCD, Proceedings of the International Conference on Computer Design*, pages 217–221, October 1989.

[BO02] Valeria Bertacco and Kunle Olukotun. Efficient state representation for symbolic simulation. In *DAC, Proceedings of Design Automation Conference*, June 2002.

[Bry85] Randall E. Bryant. Symbolic verification of MOS circuits. In *Proceedings of 1985 Chapel Hill Conference on VLSI*, pages 419–438, May 1985.

[CBM89] 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.

[CCLQ97] 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.

[CCQ96] 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.

[CJB79] William C. Carter, William H. Joyner, and Daniel Brand. Symbolic simulation for correct machine design. In *DAC, Proceedings of Design Automation Conference*, pages 280–286, June 1979.

[GB03] Amit Goel and Randal E. Bryant. Set manipulation with Boolean functional vectors for symbolic reachability analysis. In *DATE, Design, Automation and Test in Europe Conference*, pages 10816–10821, March 2003.

[HSH+00] 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.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[KKD01] Alferd Kolbl, James Kukula, and Robert Damiano. Symbolic RTL simulation. In *DAC, Proceedings of Design Automation Conference*, pages 47–52, June 2001.

[MKRS00] 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.

[REH99] Gerd Ritter, Hans Eveking, and Holger Hinrichsen. Formal verification of designs with complex control by symbolic simulation. In *CHARME, Proceedings of Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 234–249. Springer-Verlag, 1999.

[RS95] 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.

[SB95] Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, 1995.

[VBJ97] Miroslav N. Velev, Randal E. Bryant, and Alok Jain. Efficient modeling of memory arrays in symbolic simulation. In *CAV, Proceedings of International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 388–399. Springer-Verlag, June 1997.

[WCZK01] Dong Wang, Edmund Clarke, Yunshan Zhu, and Jim Kukula. Using cutwidth to improve symbolic simulation and Boolean satisfiability. In *HLDVT, IEEE International High Level Design Validation and Test Workshop*, pages 165–170, November 2001.

[WD00] 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.

[YS03] Jin Yang and Carl-Johan H. Seger. Introduction to generalized symbolic trajectory evaluation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(3):345–353, 2003.

Chapter 4

COMPACTING INTERMEDIATE STATES

This chapter is dedicated to introducing two techniques that enable the transformation of intermediate states of symbolic simulation into compact parametric vectors. The two techniques are parametrization and disjoint-support decomposition. We first introduce the basic concepts of parametrization through simple examples, and we discuss the motivation for this direction of work in symbolic simulation, namely, a reduced memory profile in simulation and a more intuitive description of the state of the design. We then introduce disjoint-support decompositions, also called DSDs, and present the key aspects and benefits of this representation for Boolean functions. We attempt to provide a broad introduction to disjoint-support decomposition and omit much of the formal derivation. The interested reader is referred to the Appendix for a detailed analysis of the canonicity of disjoint-support decompositions and the construction algorithm. The two chapters that follow will use one or both of these techniques to create scalable and compact symbolic simulation solutions.

4.1 Parametric transformations

A parametrization is a transformation which considers the range of a vector function **F** (see also Definition 2.3), and encodes this range through a new vector function **P** over a fresh set of parametric variables, so that **P** spans exactly the same range. The hope when performing a parametrization is that the parametric vector has a more compact representation than the original vector.

Example 4.1. *Consider the vector function* $\mathbf{F} : \mathcal{B}^5 \rightarrow \mathcal{B}^5$*:*

$$\begin{aligned} \mathbf{F}(i_1, \cdots, i_5) \;=\; & < i_1\overline{i_2}, \quad i_2 i_3 + i_2 i_4 i_5 + \overline{i_1}(i_3 + i_4 i_5), \\ & \overline{i_1} + i_2, \quad i_1\overline{i_2 i_3}(\overline{i_4} + \overline{i_5}), \quad i_3 + i_4 i_5 > \end{aligned} \tag{4.1}$$

The range of **F** *can be computed by trying all the possible assignments to the input variables* i_i*, as shown below:*

input	*output*	*input*	*output*	*input*	*output*	*input*	*output*
00000	*00100*	*01000*	*00100*	*10000*	*10010*	*11000*	*00100*
00001	*00100*	*01001*	*00100*	*10001*	*10010*	*11001*	*00100*
00010	*00100*	*01010*	*00100*	*10010*	*10010*	*11010*	*00100*
00011	*01101*	*01011*	*01101*	*10011*	*10001*	*11011*	*01101*
00100	*01101*	*01100*	*01101*	*10100*	*10001*	*11100*	*01101*
00101	*01101*	*01101*	*01101*	*10101*	*10001*	*11101*	*01101*
00110	*01101*	*01110*	*01101*	*10110*	*10001*	*11110*	*01101*
00111	*01101*	*01111*	*01101*	*10111*	*10001*	*11111*	*01101*

By collecting all of the output vectors we gather the range of **F***:*

$$\mathcal{R}(\mathbf{F}) = \{< 00100 >, < 01101 >, < 10010 >, < 10001 >\} \tag{4.2}$$

Since the cardinality of $\mathcal{R}$ *is only four, we should be able to find an encoding of the elements of this set using only two parameters. One possible such encoding is:*

$$\mathbf{P}(p_1, p_2) =< p_1, p_2\overline{p_1}, \overline{p_1}, , p_1\overline{p_2}, p_2 > \tag{4.3}$$

By similar enumeration, it is possible to check that the parametric vector **P** *spans exactly the same range as* **F***, even though each distinct output value is not repeated at multiple input combinations. Moreover,* **P** *has the advantage of using simpler expressions and fewer symbolic variables.*

In the context of symbolic simulation, one key observation that was made very early on is that one of its sources of complexity is the memory required to represent the Boolean expressions at the circuit's outputs and internal nodes. However, each time a simulation step is completed, the only information that is carried forward is the range spanned by the next-state vector. Hence, if we could find simpler parametric Boolean expressions, such as those in the example, that span exactly the same range, we could simplify the simulation job without sacrificing its breadth.

4.1.1 A formal definition

Consider a Boolean vector function $\mathbf{V}(i_1, \cdots, i_n) : \mathcal{B}^n \leftarrow \mathcal{B}^m$. For each distinct assignment to the input variables i, the vector will evaluate to an m-bits Boolean value. Collectively, all the values generated constitute the range of **V**, $\mathcal{R}(\mathbf{V})$. In the most general context, a parametric transformation is a transformation that generates another Boolean vector **P**, whose range is identical to the one of **V**:

Definition 4.1. *A* **parametric transformation** *is a mapping from a Boolean vector* $\mathbf{V}(i_1, \cdots, i_k)$ *to another Boolean vector* $\mathbf{P}(j_1, \cdots, j_r)$ *such that the range (indicated by the symbol* $\mathcal{R}$*) of the two vectors is identical:*

$$\mathcal{R}(\mathbf{V}(i_1, \cdots, i_k)) = \mathcal{R}(\mathbf{P}(j_1, \cdots, j_r)) \tag{4.4}$$

If we assume that Boolean vectors are represented by BDDs, then interesting parametrizations are those that generate vectors **P** which have more compact BDDs than their **V** counterpart, and have possibly fewer inputs, that is $k \leq r$.

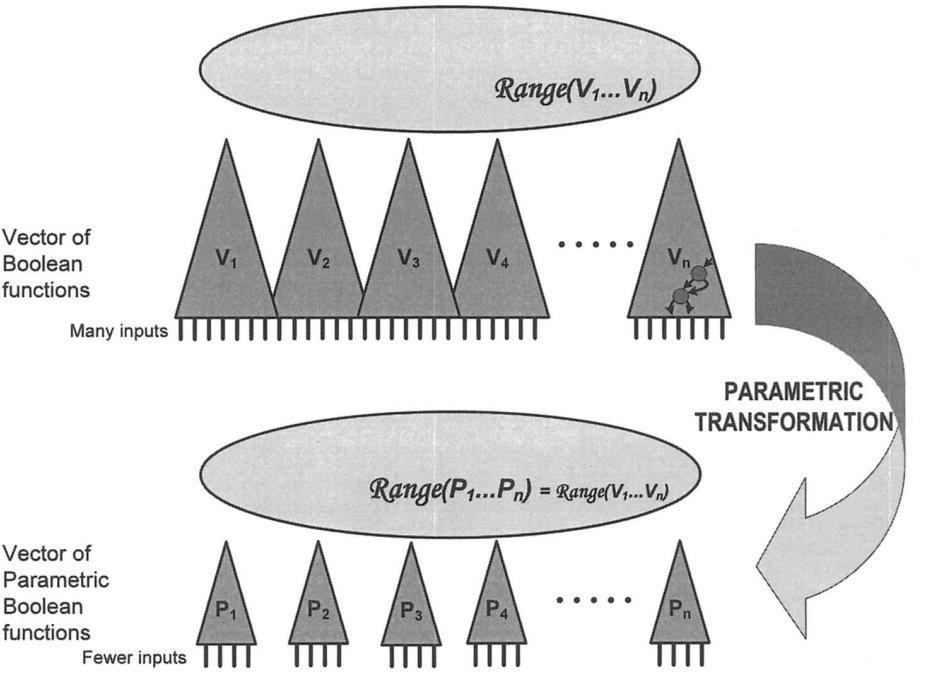


Figure 4.1: Schematic of the impact of parametric transformations

Figure 4.1 illustrates the concept pictorially: on the top part of the diagram is the original Boolean vector **V**, represented by complex BDDs, on the bottom part, the parametric vector **P**, which is described by smaller BDDs, with fewer inputs.

4.1.2 Applications to symbolic simulation

The central observation underlying the work presented in the next chapter of this book is that the expressions involved in a symbolic 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 3.3, 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 $\mathbf{S}_{@\mathbf{k}}$ is most often redundant. In fact, the only information that must be transferred

across simulation steps is the set of states that have been reached in the previous step. Therefore, it can be generally possible to define a more compact encoding of this set description, that is, a new *parametrization* of the state set. We can then use this new implicit description for the next simulation step. Figure 4.2 shows where the transformation takes place in the simulation flow. Note that the parametrization phase does not need to applied at each cycle, it is possible to have an evaluation mechanism that triggers the parametrization only when needed, for instance because of scarce resources available.

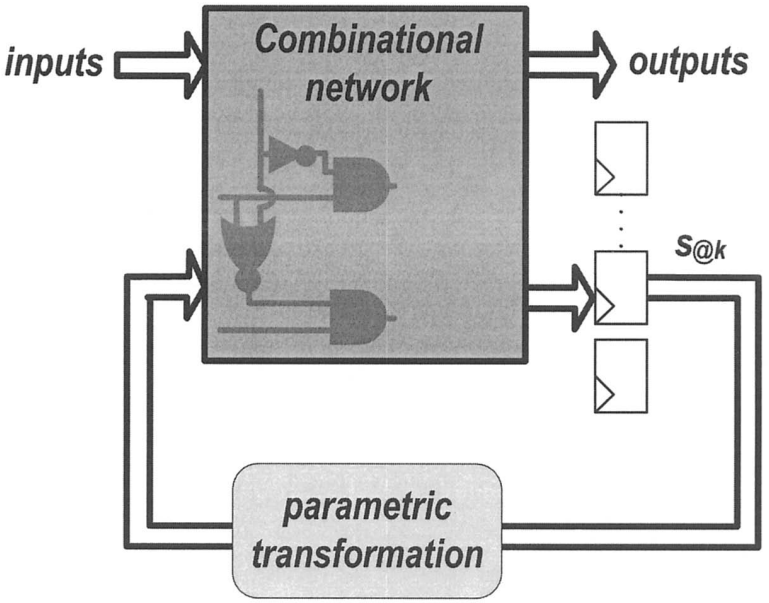


Figure 4.2: Parametrization of the state vector during symbolic simulation

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

Example 4.2. *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 4.3 - 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 parametrization 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 4.3 - 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. Once 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:*

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

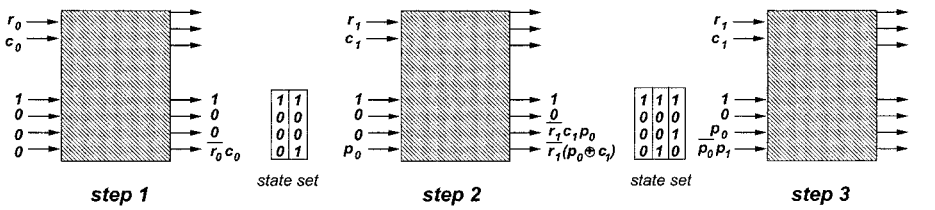


Figure 4.3: Three steps of symbolic simulation for the counter of Example 2.2 and possible parametrizations of the reached state sets

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 memory representations allowed by

parametrization, we need to deploy an efficient algorithm that can discover good parametrizations automatically.

4.1.3 A brief history of parametric solutions

An entire branch of research on symbolic simulation has concerned itself with finding efficient and compact parametrizations for the next-state functions $ST_{@k}$. We overview here some of the solutions proposed, while the following chapters will cover in depth a few of them.

Some of the first parametrization work was developed by Jain and Gopalakrishnan in [JG92, JG94], where they stated the relevance of parametrization in symbolic simulation and evaluated a range of techniques for a few specialized applications. Other solutions rely on user interaction to suggest candidate relations among the signals of the systems under verification, that will then be explored when parametrizing the simulation. For instance, in [HD93], the authors exploit the dependencies among state variables to simplify the traversal of a FSM. Such dependencies are suggested by the designer and are verified for correctness during the simulation of the system.

The work on reachability analysis by Coudert and Madre [CBM89, CM90] can also be viewed as a parametric transformation (which they call functional parametrization) from an explicit next-state vector to an implicit reached set. In contrast, Aagard, *et al.*, [AJS99] addressed the scalability of simulation by both partitioning the search space and using a parametrization technique where an implicit function is transformed to an explicit set of functions, with an overall more compact representation. The structural partitioning is based on a design-derived case splitting, so that each individual case can be fairly compact and self-contained. Van Eijk, *et al.*, in [vEJ96], automatically discover dependencies among state variables during finite state machine 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. Consequently, the copious computation required offsets the benefits of the more compact parametric representation. Another contribution in this area is by Moon, *et al.*, in [MKK+02], who developed an alternative technique by creating a compact, functionality-preserving representation for a circuit portion, which reduces the overall complexity of combinational equivalence checking.

A research direction related to symbolic simulation and parametrizations 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, parametrization techniques have been used to express the conditions of each sub-problem in the partitioned constraints. In particular, Jain, *et al.*, considered in [JG93] a variety of Boolean formula representations for the constraints and proposed a method to obtain parametric solutions. Aa-

gaard, *et al.*, [AJS99] introduced an alternative method where the case splitting on the constraints is based on Shannon decomposition. Our techniques from [BDQ99] and [BO02] are also in the realm of parametric solutions, the former in relation to fast approximations, and the latter in conjunction to disjoint-support decompositions. All these solutions will be discussed in depth in the following chapters.

The next few sections of this chapter are dedicated to present the theory of disjoint-support decompositions. We discuss the theoretical background and present a recent algorithm that enables the computation of decompositions for complex functions of thousands of variables. We will see in the next chapters how this algorithm and parametrizations techniques are used to improve the scalability of simulation.

4.2 Disjoint-support decompositions

Disjoint-support decomposability is an intrinsic property of Boolean functions. Most functions arising in practical design contexts present some amount of decomposability, that is, they can be split in two or more sub-functions, which do not share any input signals.

In general terms, 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 with 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.* A qualitative representation of this process is depicted in Figure 4.4.

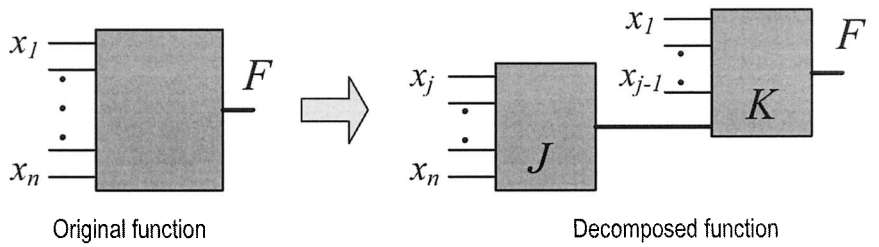


Figure 4.4: General form of a disjoint-support decomposition (DSD)

The following example shows how a functions has in general multiple distinct disjoint-support decompositions. If that is the case, there is always a "common-denominator" decomposition with a finer granularity than all the other decompositions.

Example 4.3. *The function* $F = (x_1 + x_2)(x_3 \oplus x_4)$ *has a simple disjoint-support decomposition where* $K_1 = J_1(x_3 \oplus x_4)$ *and* $J_1 = x_1 + x_2$ *as in Figure 4.5.a. The decomposition* $K_2 = (x_1 + x_2)J_2$ *and* $J_2 = x_3 \oplus x_4$ *is also a simple disjoint sup-*

port (Figure 4.5.b). Note that it is possible to combine these two decompositions and represent the function as $F = K_3(J_1, J_2)$ *where* $K_3 = j_1 j_2$ *(Figure 4.5.c).*

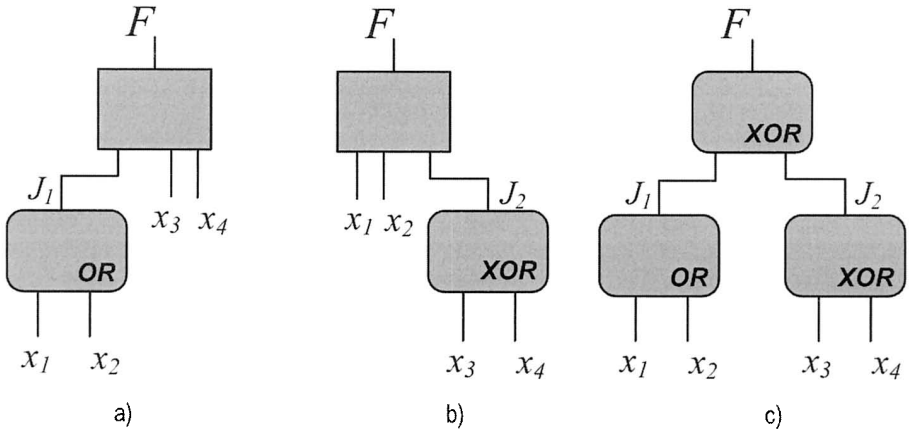


Figure 4.5: Three different disjoint-support decompositions for Example 4.3

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 4.5.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 requirement of non-intersection. We can also guarantee that they belong to no less than one function. In fact, if that was not the case, the variable x_i would not be part of the support of F, against the assumption that F depended on x_i (see Section 2.3 for a definition of support of a function). 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 4.3 is a maximal decomposition.

The following sections, complemented by the Appendix, will unfold the theory behind disjoint-support decompositions, their uniqueness, and a canonical form that we defined in recent work [BD97, Ber03b]. We will also a outline an algorithm that can create the maximal decomposition tree for any Boolean function, starting from its BDD representation. The algorithm has worst-case complexity quadratic in the size of the input BDD. A symbolic simulation solution discussed in Chapter 6 shows that the decomposed form can be used to

generate efficiently an exact parametrization of the intermediate state vectors of simulation.

4.2.1 A canonical form of DSDs

In the most general terms, the disjoint-support decomposition of a scalar function $F : \mathcal{B}^n \rightarrow \mathcal{B}$ consists of finding other, simpler functions L and A_i such that:

$$F(x_1,x_2,..,x_n) = L(A_1(x_1,..,x_{A_1}),A_2(x_{A_1+1},..,x_{A_2}),..,A_k(...,x_n)) \quad (4.5)$$
$$\text{with} \mathcal{S}(A_i) \cap \mathcal{S}(A_j) = \emptyset, \forall i,j$$

The decomposing function L is called a *divisor* of F. In addition, functions can be grouped into two different categories based on how they decompose. A function for which no disjoint-support decomposition exists is said to be **prime**. For instance $F = ac + b\overline{c}$ is prime, since it cannot be decomposed by any simpler function. Another example of prime function is the majority function: $F = ab + bc + ca$, which has no decomposition through disjoint-support components. A **maximal decomposition** is a decomposition where each component cannot be further decomposed, for instance, the decomposition in Figure 4.5.c is maximal. In a maximal decomposition, the component function blocks of the decomposition can be either prime functions or associative operators (*OR*, *AND*, *XOR*). In terms of the Equation 4.5, this is equivalent to say that L can only fall into one of the two categories above. It is relevant to note that, because of this classification, prime functions must always have three or more inputs, otherwise they would be an associative operator.

4.2.2 Decomposition trees

Note that each function A_i in Equation 4.5 can also be decomposable. Hence, by applying the decomposition recursively, we obtain a tree structure, called a **decomposition tree**. 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 the type of function L that divides the subfunction rooted at that node of the tree. Each node has k incoming edges, one for each A_i function. The function elements of the decomposition are the **actuals list**, while the variables in the support of the dividing function L constitute the **formals list**. When the decomposition is maximal, the internal nodes can be either prime functions or associative operators, and the tree is called **maximal decomposition tree**.

Example 4.4. *Consider the function* $F = \overline{h}((a \oplus b) \cdot MAJ(c,d,e+f) \cdot g) + h(k + j \cdot m)$. *Figure 4.6 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 increasing left to right for the edges in the picture. The node la-*

beled MAJ corresponds to the function majority. The other nodes corresponds to AND, OR and XOR functions.

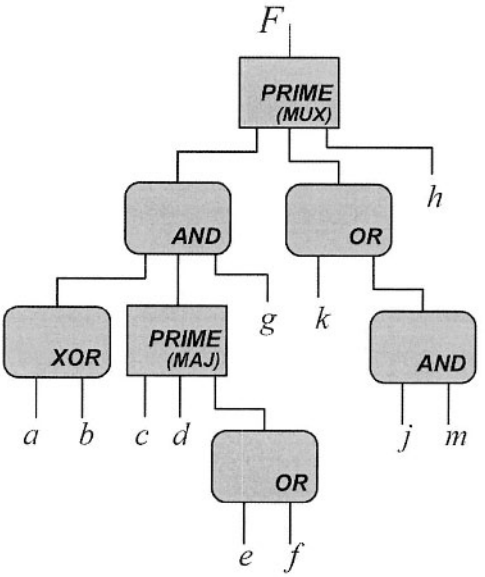


Figure 4.6: A decomposition tree for Example 4.4

We prove in the Appendix that the maximal decomposition tree of a function is unique and that distinct decomposition trees correspond to different functions. The only two constraints we must impose to construct this canonical form are that:

1 when a function is decomposed by an associative operator, the cardinality of its actuals list, that is k, must be maximal, and

2 each node of the decomposition tree must be maximally decomposed.

In a maximal decomposition tree, each of the internal nodes corresponds to a **kernel** function, which is a function that cannot be further decomposed, and abides the canonicity constraint 1. Moreover, any cut through the decomposition tree represents a valid disjoint-support decomposition for the function F. Normalization rules related to the use of complement edges allow to further compact the representation, while maintaining a canonical form, in a fashion similar to what is discussed in [BRB90] for BDDs.

Example 4.5. *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 4.7. 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.*

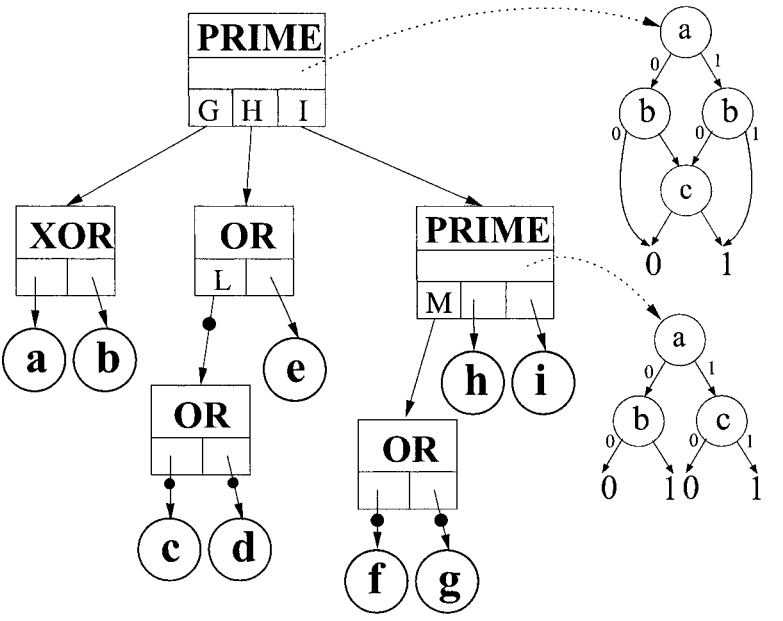


Figure 4.7: Decomposition data structure for the function of Example 4.5

4.3 A BDD-based algorithm to extract DSDs

In the previous section we presented a canonical data structure to represent all the disjoint-support decompositions of a Boolean function. We showed there and in the Appendix that this structure, the decomposition tree, is unique for each function, because of the uniqueness of the maximal decomposition.

To be able to explore the decomposability of functions, we need an algorithm that can build the decomposition tree. Historically (see Section 4.5 for more details), algorithms to this end would attempt to find decompositions by exploring all the possible partitions of the support of a function F, consequently their complexity would be exponential in the number of input variables of F. Because of this, only very small functions with a few inputs could be decomposed. The algorithm that we present here, uses a completely different approach. It starts by considering a function in its BDD form, and it proceeds by traversing the BDD tree bottom-up and constructing the decomposition of each intermediate node, until eventually it obtains the decomposition of the root node, that is the whole function. Moreover, the complexity of the algorithm is only quadratic in the size of the BDDs, which enables the construction of the decomposition tree for functions with thousands of variables. The careful reader may notice that this does not constitute a major breakthrough in the theory of computer science. In fact, it is known that in the worst case scenarios, BDDs have exponential complexity over the number of variables of the function they represent (Section 2.4). In those worst case scenarios, our algorithm is no better than the original solutions attempting all the partitioning of the support set. However, from a practical standpoint, many functions involved in the design of integrated circuits have much simpler BDDs, often of complexity just linear on the size of support set. In addition, the advent of BDDs has enabled the representation and manipulation of Boolean functions much more complex than what was possible before. For all these functions, the algorithm presented below is a viable technique to extract all the disjoint-support decompositions.

4.3.1 Building decompositions bottom-up

The decomposition algorithm, which we call DEC, operates recursively on the BDD of the function, in a bottom-up fashion. At each intermediate node it discovers the decomposition of the sub-function corresponding to that node, deriving it from the decompositions of the two children of the node, that is, the two cofactors. This construction is done by considering a list of cases that may arise and matching the correct situation to the node at hand. After the decomposition of a node is completed, the algorithm moves on to compute its parent's decomposition. Here we provide a high level overview of the cases that may arise, and we then discuss the various components of the DEC algo-

rithm. The Appendix reports a detailed discussion of all these cases and proves the correctness of the analysis.

To set the stage, assume that we are decomposing a function F, whose BDD has the variable z at the root node. We already know the decomposition tree of the two cofactors, $DT(F_0)$ and $DT(F_1)$, and we want now to compute the decomposition tree for F, $DT(F)$. In principle, one could build $DT(F)$ by running a case analysis based on the decomposition type of F_0, F_1, that is, *OR*, *AND*, *XOR* or *PRIME*. Example 4.6 below, however, indicates that this information alone may not be enough, and additional comparisons need to be carried out on $DT(F_0), DT(F_1)$:

Example 4.6. *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:*

$$\begin{aligned} F_0 &= G \\ F_1 &= G+H \end{aligned}$$

That is, F_0 has a PRIME decomposition, while F_1 has an OR decomposition. The decomposition for F can be found as follows:

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

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:

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

and K_F is a PRIME function. Thus, functions with different decomposition types may 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 would involve additional comparisons on the actual lists that are often numerous and complex. Therefore, we present here a different solution, based on the observation made by the following example:

Example 4.7. *Suppose that F has a decomposition with K_F a PRIME function:*

$$F = K_F(A_1(z), A_2, \cdots, A_l). \tag{4.6}$$

The two cofactors will then have decomposition

$$\begin{aligned} F_0 &= K_F(A_1(z=0), A_2, \cdots, A_l) \\ F_1 &= K_F(A_1(z=1), A_2, \cdots, A_l). \end{aligned} \tag{4.7}$$

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 is shown in Appendix, Section A.4.1.1 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 4.7 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 require a simpler analysis to identify the decomposition of the function F. If we impose that both cofactors of A_1 should be constant, then A_1 must have a single variable in its support and it must be $A_1 = z$ or $A_1 = \bar{z}$. 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 construction. We refer to the other situations, when at least one of the cofactors of A_1 is not constant, as an *inherited decomposition*, since in this case we are expanding a block that exists already in the decomposition of the cofactors of F by "adding" the variable z to it. The definition below formalizes the classification of new and inherited decompositions:

Definition 4.2. *We say that the decomposition of F: $\langle K_F, F/K_F \rangle$ is* **inherited** *if $|S(A_1)| \geq 2$. It is termed* **new** *otherwise.*

the Appendix shows that in an inherited decomposition, F shares the kernel (and some actuals) with at least one of its cofactors. However, in a new decomposition, this is not guaranteed to happen, and the kernel of F may be completely different from either of the cofactors.

4.3.2 Putting it all together: The DEC procedure

We detail now the decomposition procedure. This description sets the stage for the complexity analysis presented in Section 4.3.3. 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* [1] pointer (`DEC *`) to the root of its decomposition tree. With the reference to the pseudo-code in Figure 4.8, the function `GetDecomposition` simply extracts the `DEC` pointer from a BDD node, and complements it if the BDD node was negated. The call to `decompose` is the decomposition procedure proper as shown in Figure 4.9.

[1] because of the use of complemented edges – see also the Appendix at Section A.3

We attempt the decomposition as an inherited or new decomposition. Each subroutine then considers all the corresponding cases that are presented in the Appendix.

```
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;
}
```

Figure 4.8: Pseudo-code for the `decompose_node` procedure

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

Figure 4.9: Pseudo-code for the `decompose` procedure

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 A_i.

It is worth noting that `decompose_INHERITED` and `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 useful in building inherited decompo-

sitions as shown in Figure 4.10 where the case numbering is made with reference to the presentation in the Appendix. Figure 4.11 shows the pseudo-code to build new 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);
}
```

Figure 4.10: Pseudo-code for `decompose_INHERITED`

```
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);
}
```

Figure 4.11: Pseudo-code for `decompose_NEW`

Since the maximal decomposition is unique, the calling order of the various sub-procedures 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 called 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, our implementation of `decompose_node` executes, first of all,

`decompose_NEW_OR` and `decompose_NEW_XOR`, since those are the simplest situations to evaluate.

4.3.3 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 $|S_F|$ has **#VAR** variables. The reader may want to use the Appendix as a reference in following the discussion of all the possible case analysis that may arise when decomposing a function.

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, $|F/K_F| \leq |S_F|$. We now analyze the complexity of each procedure in `decompose`.

The 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, an 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: $O(|S_F|) = O(\#\mathbf{VAR}) + O(|S_F| - 1)$.

`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 four generalized cofactors. As we showed in Section A.5.1.3, these are special cofactor 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: $O(|S_F|) = O(\#\mathbf{VAR}) + 4 \cdot O(\#\mathbf{BDD}) + O(|S_F| - 2)$.

The decomposition functions `decompose_INHERITED_PRIME_2.a` and `decompose_INHERITED_PRIME_3.a` require a list intersection, a number of cofactor 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 recursion 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 call to the decomposition function `decompose_INHERITED_PRIME_3.a`. 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 had exponential complexity in the size of 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.

4.3.4 Decomposability experiments

The algorithm described in this chapter was implemented in a C++ program and tested on the circuits from the Logic Synthesis Benchmarks suite [Yan91] and the ISCAS'89 Benchmark Circuits [BBK89], 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 BDDs 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 Table 4.1. 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 these 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 was 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 large 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 of the table provide performance information. The first time and memory pair reports the time in seconds and the amount of memory in kilobytes required to produce the BDDs 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 BDDs. 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 BDD package. In particular, our BDD package records the support of the functions associated to each BDD 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.

In most cases the additional time to decompose a function is small compared to the time required to build the initial BDDs. 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 [Yan91]. By inspecting the two circuits we found that

Table 4.1: Disjoint-support decomposition results (Part 1)

Circuit	Outputs	DEC	Dec Cof	Inputs	FanIn	Blks	BDD perf		DEC perf	
							Time (s)	Mem (kb)	Time (s)	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
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

Table 4.2: Disjoint-support decomposition results (Part 2)

Circuit	Outputs	DEC	Dec Cof	Inputs	FanIn	Blks	BDD perf		DEC perf	
							Time (s)	Mem (kb)	Time (s)	Mem) (kb)
Logic Synthesis '91 - FSM tests (cont.)										
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

Table 4.3: Disjoint-support decomposition results (Part 3)

Circuit	Outputs	DEC	Dec Cof	Inputs	FanIn	Blks	BDD perf		DEC perf	
							Time (s)	Mem (kb)	Time (s)	Mem (kb)
Logic Synthesis '91 - Multi level tests (cont.)										
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
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

Table 4.4: Disjoint-support decomposition results (Part 4)

Circuit	Outputs	DEC	Dec Cof	Inputs	FanIn	Blks	BDD perf		DEC perf	
							Time (s)	Mem (kb)	Time (s)	Mem (kb)
Logic Synthesis '91 - Multi level tests (cont.)										
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
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

Table 4.5: Disjoint-support decomposition results (Part 5)

Circuit	Outputs	DEC	Dec Cof	Inputs	FanIn	Blks	BDD perf		DEC perf	
							Time (s)	Mem (kb)	Time (s)	Mem (kb)
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
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

Table 4.6: Disjoint-support decomposition results (Part 6)

Circuit	Outputs	DEC	Dec Cof	Inputs	FanIn	Blks	BDD perf		DEC perf	
							Time (s)	Mem (kb)	Time (s)	Mem (kb)
ISCAS '89 - Addition '93 (cont.)										
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

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 A.5.2.2, made it 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 4.1 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 BDDs 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. Summarizing the results we found that we could decompose 16,472 functions out of a total of 18,584. The total time spent constructing the BDDs was 79.63s, while the time spent after that to attempt the functions' decompositions was 209.11s.

4.4 On the decomposability of Boolean functions

Shannon proved in [Sha49] that for a sufficiently large support size, $|S()|$, 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, $|S()|$, that are decomposable approaches 0 as $|S()|$ approaches infinity. Sasao provided some quantitative results on how fast this limit is approached in [Sas99]. He reports there that at $|S()| = 5$, 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 composing simpler functions.

4.5 Evolution of disjoint-support decompositions

Algorithms for extracting disjunctive decompositions are a classic research subject of switching theory. Ashenhurst and Singer [Ash57, Sin53] developed the first theoretical framework in the '50s. In particular, Ashenhurst presented in [Ash57] 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 of partitioning the support variables of a function in two sets A and B and detecting if there exists 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 [Ash57] 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 explored applications for the theory in the area of synthesis of digital circuits in [Cur62, Kar63].

In the early '70s, Shen *et al.*, [SMW71], 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 [SM98].

Alternative simplified techniques, for instance, algebraic factorization, as proposed in [BM82], 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 [BM82] is a form of disjunctive decomposition. In algebraic factorization, one attempts to decompose a two-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 [BRSVW87]. Logic synthesis have been also attempted starting directly from BDD representations: In [CM92] 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 [MSBSV93]. Links between BDDs and multiple-level logic have been explored in [Kar89, Kar88, Kar90]. In [Kar90], 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 [MNS+90] and function representation [BD96b, BD96a]. In [BD96a] we proposed a new function representation that merges BDD and a restricted type of decomposition. In that work, 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:

$$F = (f_1 + \cdots + f_n)' \tag{4.8}$$

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. [BD97] overcomes this limitation and constitutes a preliminary version of the material presented in this chapter. Subsequent work has refined the algorithm and provided techniques to perform Boolean operations directly on the decomposition data structure, hence eliminating the need to build the initial un-decomposed BDD [PB05b, PB05a].

4.6 Summary

This chapter discussed parametrizations and disjoint-support decompositions, two techniques that have been used effectively in improving the scalability of symbolic simulation. We conveyed the main ideas of parametrization through examples related to simulation. For disjoint-support decompositions, we overviewed the main aspects of this theory, and we refer the interested reader to the formal presentation in the Appendix. We also presented the DEC algorithm, which can expose the maximal disjoint-support decomposition of a Boolean function represented by its BDD and which has quadratic complexity. Results show that it is very fast in practice as we were able to obtain the decomposition of most testbenches in a period of time comparable to the construction

of their BDD. Experimental results also indicate that the majority of functions representing the behavior of digital systems are indeed decomposable and the maximal disjoint decomposition has, in fact, a fine granularity, as indicated by the support size of the biggest component block. The next two chapters will exploit these techniques in devising new solutions for symbolic simulation.

References

[AJS99] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Formal verification using parametric representations of Boolean constraints. In *DAC, Proceedings of Design Automation Conference*, pages 402–407, June 1999.

[Ash57] 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.

[BBK89] 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.

[BD96a] 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.

[BD96b] Valeria Bertacco and Maurizio Damiani. Boolean function representation using parallel-access diagrams. In *Proceedings of the Sixth Great Lakes Symposium on VLSI*, March 1996.

[BD97] 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.

[BDQ99] 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.

[Ber03] Valeria Bertacco. *Achieving Scalable Hardware Verification with Symbolic Simulation.* PhD thesis, Stanford University, 2003.

[BM82] 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.

[BO02] Valeria Bertacco and Kunle Olukotun. Efficient state representation for symbolic simulation. In *DAC, Proceedings of Design Automation Conference*, June 2002.

[BRB90] 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.

[BRSVW87] 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.

[CBM89] 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.

[CM90] Olivier Coudert and Jean Christophe Madre. A unified framework for the formal verification of sequential circuits. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 126–129, November 1990.

[CM92] 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.

[Cur62] Herbert A. Curtis. *A New Approach to the Design of Switching Circuits*. Van Nostrand, Princeton, N.J., 1962.

[HD93] 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.

[JG92] Prabhat Jain and Ganesh Gopalakrishnan. Some techniques for efficient symbolic simulation-based verification. In *ICCD, Proceedings of the International Conference on Computer Design*, pages 598–602, October 1992.

[JG93] 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.

[JG94] 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.

[Kar63] Richard M. Karp. Functional decomposition and switching circuit design. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):291–335, 1963.

[Kar88] Kevin Karplus. Representing Boolean functions with if-then-else dags. Technical Report UCSC-CRL-88-28, Baskin Center for Computer Engineering & Information Sciences, 1988.

[Kar89] Kevin Karplus. Using if-then-else dags for multi-level logic minimization. In *Proceedings of Advanced Research in VLSI*, pages 101–118, 1989.

[Kar90] 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.

[MKK+02] In-Ho Moon, Hee Hwan Kwak, James Kukula, Thomas Shiple, and Carl Pixley. Simplifying circuits for formal verification using parametric representation. In *FMCAD, Proceedings of International Conference on Formal Methods in Computer-Aided Design*, pages 52–69. Springer-Verlag, 2002.

[MNS+90] 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.

[MSBSV93] 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.

[PB05a] Stephen Plaza and Valeria Bertacco. Boolean operations on decomposed functions. In *International Workshop on Logic Synthesis*, pages 310–317, June 2005.

[PB05b] Stephen Plaza and Valeria Bertacco. STACCATO: Disjoint support decompositions from BDDs through symbolic kernels. In *ASPDAC, Proceedings of the Asia South Pacific Design Automation Conference*, pages 276–279, January 2005.

[Sas99] Tsutomu Sasao. Totally undecomposable functions: Applications to efficient multiple-valued decompositions. In *ISMVL*, pages 59–65, 1999.

[Sha49] Claude E. Shannon. The synthesis of two-terminal switching circuits. *Bell Systems Technical Journal*, 28(1):59–98, 1949.

[Sin53] Theodore Singer. The decomposition chart as a theoretical aid. Technical Report BL-4, Sec.III, Harvard Computational Laboratory, 1953.

[SM98] Tsutomu Sasao and Munehiro Matsuura. DECOMPOS: An integrated system for functional decomposition. In *International Workshop on Logic Synthesis*, pages 471–477, 1998.

[SMW71] 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.

[vEJ96] 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.

[Yan91] Saeyang Yang. Logic synthesis and optimization benchmarks user guide, version 3.0. Technical report, Microelectronics Center of North Carolina, January 1991.

Chapter 5

APPROXIMATE SIMULATION

In this chapter we present approximate solutions for symbolic simulation. In particular we discuss an approximate reparametrization technique that we developed which can simulate much more complex designs than can the traditional symbolic methods described in Chapter 3, while still using a bounded amount of memory resources. The main focus of this algorithm, called cycle-based symbolic simulation, is to achieve as much breadth of traversal as possible with the constraint of maintaining the scalability and economic memory requirements of logic simulation. In the best situation, the solution presented below achieves the same breadth of traversal as a pure symbolic simulation algorithm. However, this breadth can be reduced if required in order to limit memory usage. Thus, cycle-based symbolic simulation, or CBSS, can be viewed as a hybrid approach that exploits the trade-offs between symbolic search and logic simulation. The results compare CBSS to logic simulation and show that each CBSS simulation step generates the equivalent results of multiple logic simulation test vectors. Ultimately, even if a step of CBSS is more time consuming, as it requires operating on Boolean expressions instead of constant values, overall the gain in parallelism offsets this cost, producing an improvement in performance by orders of magnitude over plain logic simulation.

In addition, we overview another approximate technique: quasi-symbolic simulation. In this case the goal is to avoid computing the symbolic value of intermediate nodes, unless strictly necessary. This simulator embeds an automatic evaluation of the nodes to be computed, and a feedback loop to adjust in the case of miscalculation.

5.1 Cycle-based symbolic simulation flow

Cycle-based symbolic simulation is a hybrid approach in the sense that the values propagated through the network can be either symbolic expressions or constant Boolean values. Section 2.4 showed that BDDs can be used to represent both data efficiently. Our algorithm, which was first published in [BDQ99], adds a *reparametrization phase* at the end of each symbolic simulation step, in line with the structure presented in 4.1.1. The CBSS reparametrization transforms the state vector into a new parametric state vector made of smaller 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 parametrization) and complexity of the expressions produced (which we want to keep at a minimum). When we under-approximate, we settle for simulating only a subset of the state set. In this situation, the objective is to select a parametrization that maximizes the amount of states represented given the amount of memory used.

5.2 The CBSS algorithm

A Cycle-Based Symbolic Simulation is initialized by setting the state of the circuit to the initial constant vector S_0 (see Section 2.5.3 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 parametrization, a minimal number of symbolic variables could be set to constants. This could be the case if the state set that the state vector spans is not representable by our simple and compact parametrization. If this happen, we under-approximate the state set that we represent, so that it coincide with one of the parametric forms that we can generate. 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 representation.

In addition, during the simulation, we do not compute a `reached` set as in symbolic reachability analysis (Section 3.4.1). This computation is one of the main causes of the limited 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 this symbolic analysis. 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 reachability analysis can address. Moreover, we use a novel reparametrization algorithm that does not require `reached` set information.

After reparametrization, the newly generated functions are used as the present state to start the next step of simulation. Figure 5.1 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 3.3. 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 $\{IN_{@0}, \cdots, IN_{@k}\}$. Notice also that by doing so we are simulating in parallel many distinct input vectors, in other words we are performing the equivalent of many logic simulation steps.

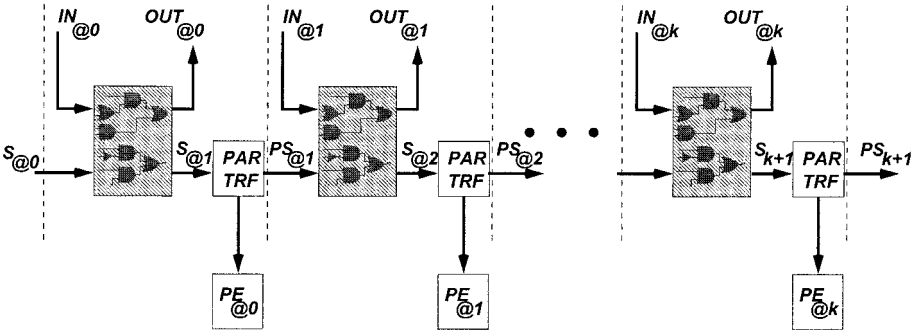


Figure 5.1: Flow of cycle-based symbolic simulation algorithm

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 simulate any neighborhood of an input trace generated by the testbench.

5.3 The reparametrization phase

The parametrization technique is based on the following observation. In symbolic Finite State Machine traversal, the next-state function δ can be often very complex. We overcome this bottleneck using the following observation. The next-state functions of symbolic simulation at time step 0, $\mathbf{S_1}$, can be de-

```
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 parametric eqns */
    parametric_state_set = Parameterize(state_symbols, step);
    assign(present_state_signals, parametric_state_set);
  }
}
```

Figure 5.2: Pseudo-code for cycle-based symbolic simulation

rived from δ as:

$$\mathbf{S_1}(i_{@0}) : I : \mathcal{B}^m \rightarrow S : \mathcal{B}^n = \{\delta(s,i) | s \in S_0, I \equiv IN_{@0}\}, \tag{5.1}$$

that is, by evaluating the state variables of the δ function to the initial state values, and substituting the combinational input variables with the input variables of time step 0. The initial state of digital system is often a single well-defined state, or it entails a few states. The result is that the initial state vector is made mostly of constant values. Hence, because the state variables in the δ function become, in turn, constant values, the resulting components of $\mathbf{S_1}$ are very simple, such as constants, copies of an input, or complements 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, parametrized 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 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 $\mathbf{S}_{@\mathbf{k}}$ at time k to build a parametrized version, $\mathbf{PS}_{@\mathbf{k}}$, for time $k+1$. If, in spite of our efforts, $\mathbf{PS}_{@\mathbf{k}}$ 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.

5.3.1 Using functional dependencies

We discover and exploit functional dependencies using a parametric representation of the next-state set. Figure 5.1 illustrates the approach. We introduce some *intermediate* variables p_i. At a generic clock tick k, we inspect the BDDs of $\mathbf{S}_{@\mathbf{k}}$ and build a function $\mathbf{PS}_{@\mathbf{k}}$ such that (see also Definition 2.3):

$$\mathcal{R}(\mathbf{PS}_{@\mathbf{k}}) = \mathcal{R}(\mathbf{S}_{@\mathbf{k}}). \tag{5.2}$$

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

$$\mathcal{R}(\mathbf{PS}_{@\mathbf{k}}) \subseteq \mathcal{R}(\mathbf{S}_{@\mathbf{k}}). \tag{5.3}$$

The state space that is not visited because of the approximation, can be absorbed later in the simulation, once the state set has a shape that can be parametrized by CBSS.

The set $\mathbf{PS}_{@\mathbf{k}}$ that we generate has cardinality that is 2^p, where p is the number of parameters we introduce during the parametrization phase. The diagram in Figure 5.3 shows the relation between the whole state space of the system, $\mathbf{S}_{@\mathbf{k}}$ and $\mathbf{PS}_{@\mathbf{k}}$.

Section 5.3.2 provides the details on $\mathbf{PS}_{@\mathbf{k}}$ 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:

$$\mathbf{S}_{\mathbf{k+1}}(i_{@k+1}) : I : \mathcal{B}^m \rightarrow S : \mathcal{B}^n = \{\delta(s,i) | s \in \mathbf{PS}_{@\mathbf{k}}, I \equiv IN_{@k+1}\} \tag{5.4}$$

and a new $\mathbf{PS}_{\mathbf{k+1}}$ can then be constructed by parametrizing $\mathbf{S}_{\mathbf{k+1}}$. Notice that the state variables are effectively replaced by the parametric variables p_i.

In addition, we build a second mapping $\mathbf{PE}_{@\mathbf{k}}$. This second mapping expresses each p_i as a function of inputs and intermediates at the previous tick. $\mathbf{PE}_{@\mathbf{k}}$ 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 $k-2, \cdots, 0$.

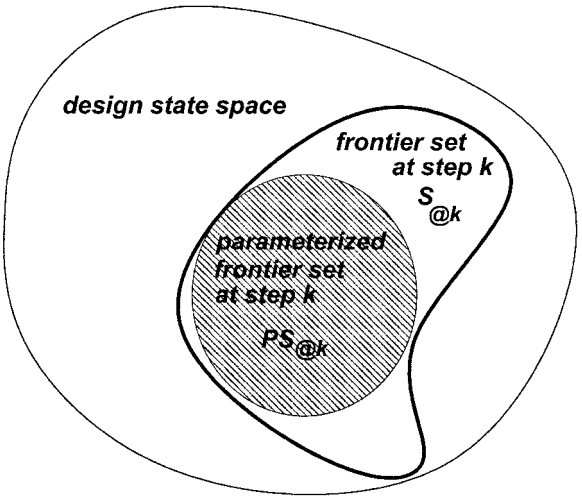


Figure 5.3: The parametrized frontier subset $\mathbf{PS_{@k}}$

The parametric transformation develops in two phases: the first phase identifies *simple* variables, while the second phase parametrizes *unbound* functions. The pseudo-code of the **Parametrize** function is shown in Figure 5.4. It guarantees that $\mathcal{R}(\mathbf{S_k})$ can be parametrized 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 constant the variable corresponding to input x, we could extract the value assigned at input x in the associate 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 path corresponding to a logic simulation run on the same testbench. In addition, if there is a testbench that drives the design to a specific corner case to be checked, 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 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 pseudo-code of the parametrization phase is shown in Figure 5.4. 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.

```
Parametrize(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 5.4: Pseudo-code for the `Parametrize` function of CBSS

5.3.2 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 $\mathbf{S}_{@\mathbf{k}}$.

Below we classify the symbolic variables in the support (see Definition 2.2) of the state vector as simple or complex. We consider having multiple support sets, one for each bit-function of the state vector. Simple variables are those variables that appears alone in at least one of the support sets. Once simple variables are identified, we can simplify complex variables, which are all those variables that share a support set with a simple variable. For instance if one support set is $\{a\}$, and another is $\{a, b\}$, but there is not set with b alone, then a is simple and b is complex and must be evaluated to a constant value. If there were a third set including b alone, than both a and b would be simple. The two definitions below formalize this description.

Definition 5.1. *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 5.2. *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 $\mathcal{S}(\mathbf{V}_i) \cap \overline{\mathcal{S}i} \neq \emptyset$.

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

Once simple and complex variables are classified, there may be additional state functions which include variables that are not classified. Actually, from the definition of simple and complex variable, we can guarantee that all the remaining non-parametrized functions contain only unclassified variables. To handle these functions we group into equivalence classes, where we assign to a class all the occurrence of a given function and its complement. The variables in these remaining functions can be classified as bound or shared. They are bound is they belong to the support of only one function, and thus equivalence class. They are shared if they belong to more than one equivalence class. Once again, we evaluated shared variables to constant, since that makes the functions disjoint support and straightforward to parametrize.

Definition 5.3. *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 5.4. *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 5.1. *Consider the following function* $\mathbf{S}_{@\mathbf{k}}$*:*

$$\mathbf{S}_{@\mathbf{k}}(x,y) : \mathcal{B}^2 \rightarrow \mathcal{B}^7 = (x,\bar{x},y,0,f(x,y),g(x,y),\bar{y}) \tag{5.5}$$

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:

$$\mathbf{PS}_{@\mathbf{k}} = (p_0,\overline{p_0},p_1,0,f(p_0,p_1),g(p_0,p_1),\overline{p_1}) \tag{5.6}$$

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 5.2. *Consider a system with the following state function at state k:*

$$\mathbf{S}_{@\mathbf{k}}(q,r,s,x,y) = (x,y,x+y+q+r,s+xq). \tag{5.7}$$

$\mathbf{S}_{@\mathbf{k},0}$ *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:*

$$\mathbf{PS}_{@\mathbf{k}}(p_0, p_1, p_2) = (p_0, p_1, 1, p_2). \tag{5.8}$$

Simple and complex variables (and functions) are identified in a two-pass scan of the BDDs of $\mathbf{S}_{@\mathbf{k}}$. Figure 5.5 shows the pseudo-code 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 5.5: Pseudo-code for classifying simple and complex support variables

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. 5.9 below:

$$\mathbf{S}_{@\mathbf{k}} = (f(p,q),x,y,g(x,y)). \tag{5.9}$$

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

$$\mathbf{PS}_{@\mathbf{k}} = (p_0,p_1,p_2,g(p_1,p_2)) \tag{5.10}$$

where

$$p_0 = f(p,q); \quad p_1 = x; \quad p_2 = y. \tag{5.11}$$

Consider now the more general situation:

$$\mathbf{S}_{@\mathbf{k}} = (f(p,q),\overline{f(p,q)},x,y). \tag{5.12}$$

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

Definition 5.4 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 5.3. *Consider a system with the following state set at step k:*

$$\mathbf{S}_{@\mathbf{k}} = (x+y+z,\overline{xyz},\bar{z}w,\bar{z}w). \tag{5.13}$$

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

$$\mathbf{S}_{@\mathbf{k},z=0} = (x+y,\overline{x+y},w,w) \tag{5.14}$$

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

$$\mathbf{PS}_{@\mathbf{k}} = (p_0,\overline{p_0},p_1,p_1) \tag{5.15}$$

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

Figure 5.6 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 accordingly.

```
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;
      }
    }
    foreach (class) in (EC)
    {
      foreach (x) in (support(class))
      {
        if (tag(x) == shared) Sh= Sh ∪ {x};
      }
    }
  }
  return <Sh, EC>;
}
```

Figure 5.6: Pseudo-code for classyfing shared support variables

5.3.3 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 parametrization, for each parameter, there is at least one equation whose expression is simply the parameter variable.

Example 5.4. *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:

$$\begin{array}{llll} s_0 = p_0 & s_3 = p_0 p_1 & s_6 = p_2 & s_8 = 1 \\ s_1 = p_0 & s_4 = 0 & s_7 = \overline{p_2} & s_9 = p_3 \\ s_2 = p_1 & s_5 = p_1 & & \end{array}$$

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 parametrized state vector. The* $\mathbf{PE}_{@\mathbf{k}}$ *equations are:*

$$p_0 = a \qquad p_1 = b \qquad p_2 = d+e \qquad p_3 = h \tag{5.16}$$

Note that the number of parameters that are needed during each parametrization 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 parametrized state equation $\mathbf{PS}_{@k,j}$ such that $\mathbf{PS}_{@k,j} = p_i$.

5.4 Implementation and insights

In implementing the algorithm we made some observations that made possible to use the Boolean variables needed for efficient simulation. 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 parametrization.

Second, we noticed that at the end of each parametrization 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 - $O(n)$.

5.4.1 Experimental results

The CBSS algorithm was implemented in a C++ program and tested on the largest sequential circuits from the Logic Synthesis Benchmarks suite [Yan91] and the ISCAS'89 Benchmark Circuits [BBK89], including their 1993 additions. Table 5.1 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, [CUD99], 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 parametrization 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 parametrizations. 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 generate during a parametrization 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

Table 5.1: Cycle-based symbolic simulation results (Part 1)

					Parameterization			Time (s)		Efficiency	Memory (KB)	
Circuit	In	Out	FF	Gates	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	$1.89 \cdot 10^{29}$	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	-

Table 5.2: Cycle-based symbolic simulation results (Part 2)

					Parameterization			Time (s)		Efficiency	Memory (KB)	
Circuit	In	Out	FF	Gates	Params	Ass.d	Symbols	CBSS	Logic	ratio	CBSS	Logic
ISCAS '89 - FSM tests (cont.)												
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	-

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 parametrization each symbol doubles the number of states spanned by the parametrized 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 2.6 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 $2^{\text{Symbols}} \cdot (\text{Time-logic/Time-CBSS})$. 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 parametrizations 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 parametrization 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 parametrizations.

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.

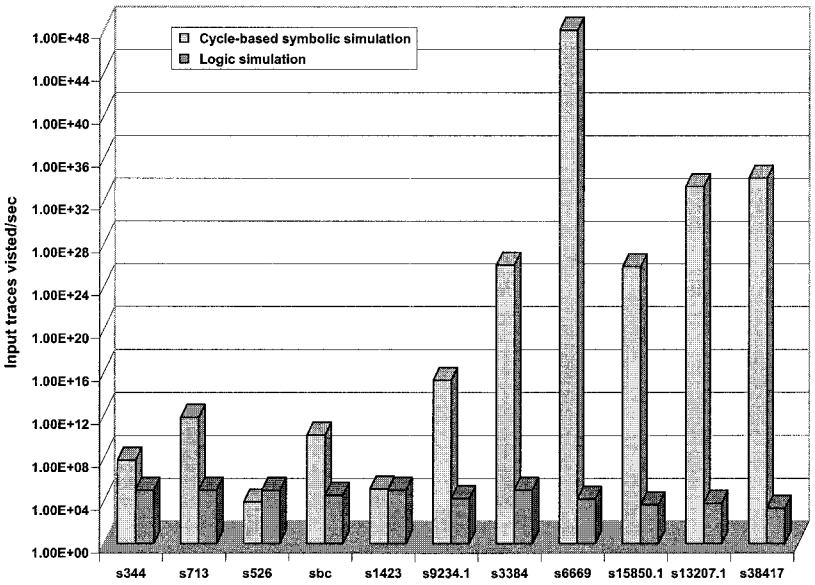


Figure 5.7: Comparison of CBSS vs. logic simulation

Figure 5.7 shows a comparison between our CBSS solution and logic simulation in terms of efficiency, *i.e.*, input traces visited per second of simulation. The testbenches on the horizontal axis are ordered by increasing complexity. As discussed earlier, logic simulation has fast performance, but explores only one input trace per simulation step. On the other hand, CBSS is slower, but can span many traces at once. The diagram shows that for most testbenches CBSS shows better efficiency, and the benefits seems to grow with the complexity of the testbench. 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 parametrizations: 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.

The following section present a solution based on the approximation of complex Boolean functions with X values.

5.5 Quasi-symbolic simulation

Chris Wilson's solution [WD00, WDB00] to the scalability problem suggests to avoid the computation of complex symbolic expressions at internal nodes that are irrelevant for the verification goal at hand. His algorithm starts very conservatively by carrying forward only very simple expressions: whenever the simulator encounters a node which requires constructing a complex expression, instead of generating such expression, it simply assigns the value X to this node (X represents an unspecified value for that node, which can be seen as an approximation of its real symbolic value, hence the name). At the end of a simulation step, some outputs will be simple symbolic expressions, others will have X values. If the verification goal was to check for a specific value at an output with a symbolic expression, the check can be performed as usual. If, instead, the output has an X value, then we need to "refine" it in order to discern if the check passes or not. The refinement is achieved by re-simulating, this time computing the symbolic value for additional internal nodes that are deemed relevant for our desired output node, based on circuit topology and dynamic values at other internal or input nodes. The refinement phase may require that a simulation step be repeated one or more times, after which the check can be performed and simulation can advance. The example below provides a high-level description of the algorithm's flow through a simple scenario.

Example 5.5. *Consider a design whose outputs include a* fail *signal which should not assume the value* 1 *under any correct working condition. The simulation is initialized as usual by assigning a distinct symbolic variable to each of the circuit's combinational inputs. However, if the evaluation of any internal node requires computing a complex expression, then, instead of generating the expression, the value X is simply assigned to the node. This value is propagated based on an extended definition of Boolean operations which considers the case where one of the operands is X (see Figure 5.8). If, at the end of a simulation step k, we obtain a value X at the* fail *output, then the simulation needs to be refined and re-simulated. This is done by selecting one of the input symbolic variables and by re-evaluating the simulation twice, by substituting the values* 0 *and* 1 *in turn for the symbolic variable. The hope is that one of*

the two re-simulations will generate a simple symbolic expression for the output of interest. If this is not the case, an additional input variable is selected for simplification and the process is repeated recursively. Once a well-defined symbolic expression is obtained for fail, *it is straightforward to evaluate the correctness of the design using the technique presented in 3.3.1 (in addition the expressions for* $\mathbf{OUT}_{@k_i}$ *will be very simple). If the design is found correct up to step k, the simulation can proceed forward.*

The approach taken by this technique shares common traits with symbolic trajectory evaluation (STE), outlined in Section 3.4.2, in particular the use of approximate values and the view of simulation as a continuous computation over a virtual unrolled design. For instance, it is possible that the refinement phase requires evaluating symbolic variables from previous simulation steps, and hence re-simulating the circuit, in general, from the initial state. However, this work has some important differences with STE in that the approximate simulation and the refinement process are completely automatic and transparent to the user, while in traditional STE, the nodes with approximate values are hand selected, and the refinement loop has to be managed directly by a user.

The following sections provide insights into the components of quasi-symbolic simulation. For the interested reader, a detailed presentation is available in Wilson's Ph.D thesis [Wil01].

5.5.1 Simulation with X values

We mentioned that the evaluation of complex Boolean expressions is avoided in quasi-symbolic simulation by replacing these expressions with X values at appropriate internal nodes. In order to do so, we need to extend the classic logic operations to the ternary domain $\{0,1,X\}$. The tables in Figure 5.8 indicate that X is propagated only when the other input operand is not controlling the output value. Note that this is in line with the definition of operation over X values in the hardware description languages (HDL). In both contexts, X represents an unknown or undefined value.

NOT(a)

a	
0	1
1	0
X	X

AND(a,b)

a \ b	0	1	X
0	0	0	0
1	0	1	X
X	0	X	X

OR(a,b)

a \ b	0	1	X
0	0	1	X
1	1	1	1
X	X	1	X

Figure 5.8: Definition of logic operations over the ternary set $\{0,1,X\}$

Moreover, the expressions carried forward in quasi-symbolic simulation are specified over this ternary set, hence a representation with plain BDDs is not suitable anymore (see Section 2.4). To this end, the authors of this work used a variant of BDDs called multi-terminal BDDs (MTBDDs) which can handle multiple terminal values (three, for the problem at hand). MTBDDs benefit from similar advantages as BDDs, including canonicity, simple recursive routines to perform logic operations, and approximately linear memory profile for most circuits. On the downside, the node-sharing aspects of BDDs that make them such a compact data structure, are usually less pronounced in MTBDDs, and the potential for sharing nodes within a function decreases with an increasing number of terminal nodes.

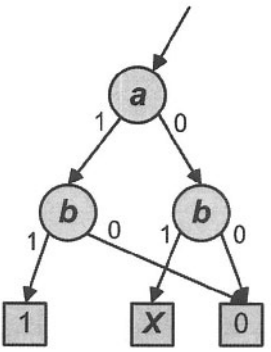


Figure 5.9: MTBDD for the function $(a+X)b$

Example 5.6. *Figure 5.9 shows the MTBDD for the function* $f(a,b) = (a+X)b$. *The function assumes the approximate value* X *only when* $a = 0$ *and* $b = 1$. *For all other input assignments, the output of* f *is a fully specified value. Note that the two nodes labelled by the variable b could not be shared because their terminals differ. In general, for MTBDD, the more distinct the terminal nodes, the fewer the opportunities for sharing.*

5.5.2 Approximating and reclassifying symbolic variables

We stated in the previous section that one of the relevant aspects of quasi-symbolic simulation was the ability to automatically select the variables to be approximated for re-simulation. For the purpose of this discussion, an input variable is the symbolic variable associated with an input signal of the design at a specific time step.

Quasi-symbolic simulation uses two main techniques to refine a simulation that leads to an unspecified X value for the relevant outputs of the design. The first technique is by *case splitting* and was described in Example 5.5. It consists in selecting a symbolic variable and performing two re-simulations,

one evaluating it to a logic 1 and the other with a logic 0. As an optimization, if the first re-simulation is sufficient to fail the checker under consideration, the second one can be avoided. The variables to be used for case splitting are chosen among those present in the most complex logic expression generated.

The second technique classifies variables as care or don't care. In general, this selection is based on an approximate evaluation, and refined at each iteration of re-simulation. Note, however, that during each re-simulation, only the portion of the internal nodes affected by the classification refinement need to be evaluated.

Example 5.7. *In the circuit shown in Figure 5.10, two inputs are associated with two symbolic variables, the third has an X value. However, note that by propagating the symbolic expressions, the symbolic output is well defined, and can be checked for correctness. If the expression obtained at the output had contained X values, a re-simulation would have been required.*

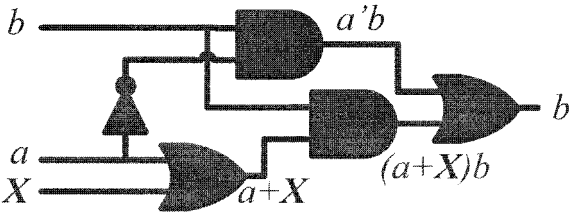


Figure 5.10: Quasi-symbolic simulation for Example 5.7

Variables are classified in control variables, data variables and don't care variables. Control and data variables are associated with control and data signals of the design, respectively. Don't care variables are variables that are irrelevant for the purpose of the test at hand, and they can be specified as such by the user, or reclassified as don't care by the simulator. At the beginning of a simulation only the control variables are symbolic, while data and don't care variables may be assigned the value X. Assigning symbolic variables to data signals can be beneficial because it provides additional coverage. However, often this benefit is offset by the high complexity increase of manipulating many more variables and ternary expressions. On the other hand, the verification of the data-path portions of a design can be more easily addressed through constrained random simulation, since it presents more uniform behavior.

5.5.3 Care and Don't care sets

The initial classification of input variables as care and don't care variables may not be sufficient to obtain a well defined output expression in the simulation. If this is the case, the quasi-symbolic simulator must re-evaluate the

partition between care and don't care variables. To do so, each internal node is tagged with two sets of variables: care and don't care. These set are propagated forward during the simulation and computed for each internal gate. Once an output is reached that has a value X, the care set indicates which variable must be converted from don't care to care in re-simulation. Because at each re-simulation only a finite number i of variables are re-classified, it is sufficient to generate care and don't care set with i elements. As a consequence, the amount of memory required to maintain these lists is proportional to the size of the circuit alone, and not to the number of input signals or the depth of the simulation.

Care and don't care set, called C-set and D-set, respectively, are generated and propagated along with the symbolic expressions across each internal gate. The propagation obeys specific and intuitive rules. For instance, the C-set at the output of an **AND** gate is the union of the inputs' C-sets (the cardinality of the resulting C-set can be pruned, based on the maximum set cardinality allowed) and the D-set is given by the intersection of the D-sets. A **NOT** gate will simply swap C-set and D-set of the input node. The impact of other logic gates on the care and don't care set propagation can be found in [WD00, Wil01].

Experimental results have shown that the approximation and automatic classification techniques of quasi-symbolic simulation have the ability to contain greatly the rate of growth of memory resources required by simulation, and contribute to create a simulation solution that can address industrial size designs and produce verification results that pure frame-based simulation cannot achieve. Moreover, whenever the verification goal is too complex to be achieved, the solution presented here has the ability to degrade gracefully and provide at least some partial results.

5.6 Summary

This chapter presented two approximation solutions for symbolic simulation. Ideally, both cycle-based symbolic simulation and quasi-symbolic simulation improve scalability and performance of simulation by running an approximation of the state space and exploiting the symbolic information within. CBSS solves this problem using reparametrization, while quasi-symbolic simulation has an automatic and adaptive detection technique to select the symbols to approximate.

CBSS has shown to improve the scalability of symbolic simulation by providing a quick and memory-friendly parametrization technique for the state equations. It can find quickly a large subset of the frontier set which can be represented with great efficiency. The experimental results show that, in most cases, we can achieve 10-20 orders of magnitude (or more) better efficiency over a compiled logic simulator. However, in some cases we notice that many variables in the support of the state vector are complex, or shared, and need

to be evaluated to a constant value. In those situations 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 parametrization, we need to explore better techniques to represent the state vector through parameters. To this end, the next chapter introduces alternative exact parametrization techniques.

References

[BBK89] 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.

[BDQ99] 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.

[CUD99] CUDD-2.3.1. *http://vlsi.Colorado.edu/fabio*, 1999.

[WD00] 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.

[WDB00] Chris Wilson, David L. Dill, and Randal E. Bryant. Symbolic simulation with approximate values. In *FMCAD, Proceedings of International Conference on Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 470–485. Springer, November 2000.

[Wil01] James Christofer Wilson. *Symbolic Simulation Using Automatic Abstraction of Internal Node Values*. PhD thesis, Stanford University, December 2001.

[Yan91] Saeyang Yang. Logic synthesis and optimization benchmarks user guide, version 3.0. Technical report, Microelectronics Center of North Carolina, January 1991.

Chapter 6

EXACT PARAMETRIZATIONS

This chapter discusses parametrization techniques that reduce the size of the BDDs of the state vector without compromising accuracy, in contrast with the approximation techniques in Chapter 5. The first technique presented is concerned with the verification of microprocessor designs. The work by Jones, *et al.*, addresses this goal with a range of techniques, including the use of multiple engines in a hybrid verification solution [AJS99]. We are concerned here with the aspects of this work related to parametrization in symbolic simulation, which we describe in Section 6.3.

The second technique is disjoint-support decomposition-based (DSD-based) symbolic simulation. It was developed by us in [BO02], and it exploits the disjoint decomposition properties of Boolean functions when generating a parametric form for a state vector. By restructuring the next-state functions in their disjoint support components into simpler sub-functions which do not share any variable, we gain a better insight in to the role of each input variable and we can then generate a compact parametrization based on the structure of the decomposition. Consequently, we can simplify the next-state functions without sacrificing the accuracy of simulation. In those situations where the simplifications enabled by DSD are not sufficient, we deploy case-splitting partitioning of the state space to maintain a low memory profile during simulation. At the end of the chapter, we provide experimental results on simulations run with a fixed quota of memory resources. Under this constraint, we show that this approach can visit much deeper states and explore a much larger design space, compared to that of plain symbolic simulation.

6.1 Re-encoding the state function using DSDs

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 Section 4.3 allows us to take advantage of such insights very efficiently.

We saw in Chapter 5 that quasi-symbolic simulation and, in particular, cycle-based symbolic simulation are 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 to maintain computational efficiency. In the latter solution, sometimes this trade-off produces simulation performance that is comparable to plain logic simulation in terms of vectors simulated per second. The parametrization presented here, in a new algorithm called *DSD-based symbolic simulation* (DSD-SS), exploits the disjoint decomposition of the state vector functions, to generate an exact parametrization, 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, and thus a simpler BDD representation.

In formal terms, the parametrization of CBSS was building a function $\mathbf{PS}_{@\mathbf{k}}$ such that

$$\mathcal{R}(\mathbf{PS}_{@\mathbf{k}}) \subseteq \mathcal{R}(\mathbf{S}_{@\mathbf{k}}),$$

while the algorithm unveiled in the next few sections builds a parametrized vector function with

$$\mathcal{R}(\mathbf{PS}_{@\mathbf{k}}) = \mathcal{R}(\mathbf{S}_{@\mathbf{k}}).$$

In order to generate the parametric state vector for DSD symbolic simulation, 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 Section 4.2, it is now possible that two or more elements intersect at some intermediate node of their decomposition 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. The structure is the decomposition tree for the function s_1. We call the graph representing the union of all the state vector decompositions a **decomposition graph**.

The decomposed representation is generated dynamically during simulation. At the completion of each cycle, we create the decomposition graph for the state vector and then use this to generate an efficient parametrization to be used in the next step. The parametrization 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, which are: 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

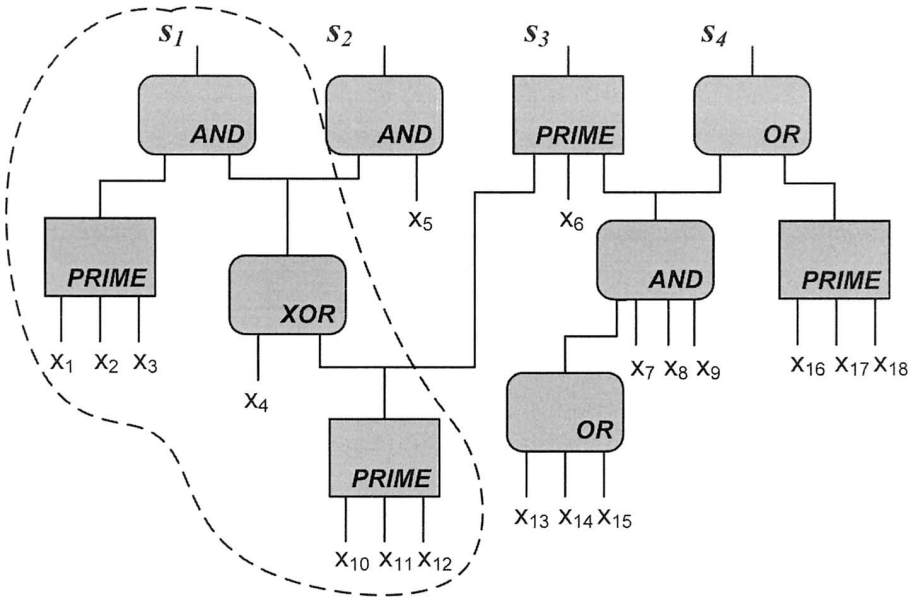


Figure 6.1: The decomposed state vector for a small design

$\mathbf{PS}_{@\mathbf{k}}$ that has also 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. This observation was also made in greater length in Section 4.1.

The relationship between the set of states spanned by the new $\mathbf{PS}_{@\mathbf{k}}$ 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 5.3 of the CBSS parametrization of Section 5.3.

In the following sections we present various transformations that we apply to the decomposition graph to accomplish the objective of producing an exact parametrization 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, such as *AND*, *OR* and *XOR*.

In presenting the techniques, we will refer to the generic vector function $\mathbf{F}$ instead of $\mathbf{S}_{@\mathbf{k}}$ since such transformations can be applied to any Boolean vector

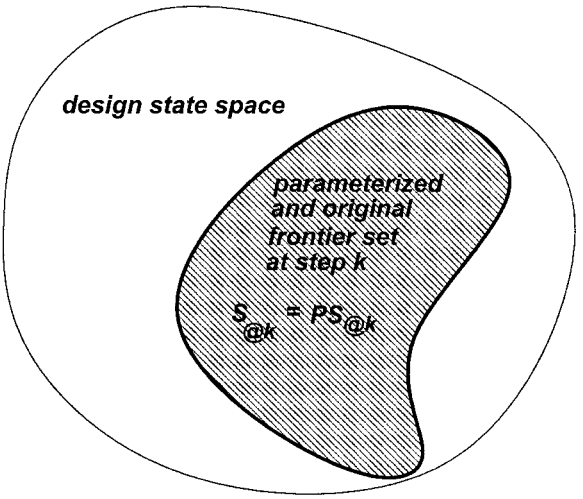


Figure 6.2: The parameterized frontier set $\mathbf{PS}_{@\mathbf{k}}$

function. Moreover, we will use the terms *decomposition graph* **F** and *function* **F** interchangeably to refer to the multiple output function **F**.

6.1.1 Reduction at free points

The first transformation, called *reduction at free points*, aims at simplifying 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 a formal proof that the transformation does not affect the range of the vector function. The following definition is also illustrated in Figure 6.3. The definition of function composition used below was provided in Section mathbackground.

Definition 6.1. *A* **free point** p *in a decomposition graph of* **F** *is a function corresponding to an output of a block node in the graph. It has the property that, if we substitute the function (i.e.the sub-graph) rooted at the point* p *with a new input variable w, such that* $w \notin \mathcal{S}(\mathrm{p})$*, the new function* **G** *has disjoint support with the function rooted at* p*:*

$$\mathbf{F}(x_1, \cdots, x_m) = \mathbf{G}(w, x_{p+1}, \cdots, x_m) \circ \mathrm{p}(x_1, \cdots, x_p) \tag{6.1}$$

and $\mathcal{S}(\mathbf{G}) \cap \mathcal{S}(\mathrm{p}) = \emptyset$.

Figure 6.3 shows three free points with dark circles. Note that the output of p is a free point since none of the variables in the support of p appears in the

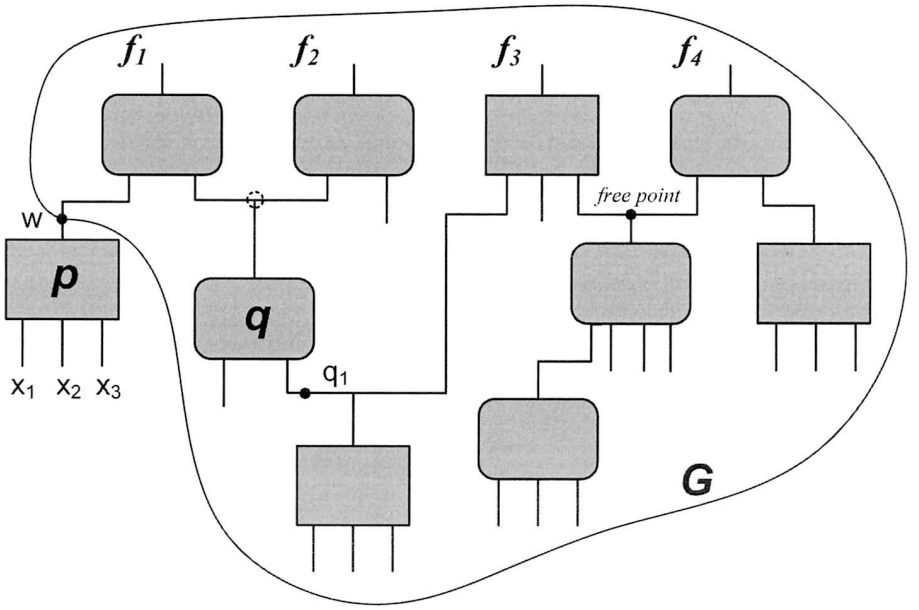


Figure 6.3: A vector function and its free points

support of other parts of the graph. On 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, **H** and q with $\mathbf{F} = \mathbf{H} \circ \mathrm{q}$, would still share the input q_1.

The following theorem shows that we can use free points to simplify the decomposition graph. In fact ,by substituting a fresh symbolic variable at each free point of the decomposition graph, we simplify the state vector functions, without altering their range.

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* **F**, *the range of the two functions* **F** *and* **G** *is the same:*

$$\mathcal{R}(\mathbf{F}) = \mathcal{R}(\mathbf{G}) \tag{6.2}$$

Proof. Consider the function $\mathbf{F}(x_1, \cdots x_m)$ and compute its range by splitting on the input variables(this type of divide and conquer approach was presented in [CBM89] in he context of FSM reachability analysis):

$$\mathcal{R}(\mathbf{F}) = \mathcal{R}(\mathbf{F}_{x_1=0}) \cup \mathcal{R}(\mathbf{F}_{x_1=1}) \tag{6.3}$$

By applying this equation recursively over all the variables $(x_1, \cdots x_p)$ in the support of p, we obtain:

$$\mathcal{R}(\mathbf{F}) = \bigcup_{(i_1,\cdots,i_p)\in\mathcal{B}^p} \mathcal{R}(\mathbf{F}_{x_1=i_1,x_2=i_2,\cdots,x_p=i_p}) \tag{6.4}$$

Then, using Equation 6.1:

$$\mathbf{F}_{x_1=i_1,x_2=i_2,\cdots,x_p=i_p} = \mathbf{G}_{p=i_w} \quad \text{where} \quad i_w = p(i_1,\cdots,i_p) \in \{0,1\} \tag{6.5}$$

since p evaluates to a constant. Substituting in Eq. 6.4 we finally obtain:

$$\mathcal{R}(\mathbf{F}) = \bigcup_{i_w\in\{0,1\}} \mathcal{R}(\mathbf{G}_{p=i_w}) = \mathcal{R}(\mathbf{G}) \tag{6.6}$$

□

Thus, we can substitute all the free points with new variables and generate a new state function **G** with a smaller representation.

To maximize the benefit, we would like to apply the substitution to the largest free points available, that is the free point use support has the largest cardinality. 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 $|\mathcal{S}(\mathbf{G})| \leq |\mathcal{S}(\mathbf{F})|$, which has still the same range of **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. This is relevant from an implementation standpoint, where functions, symbolic variables and parametric variables would be represented by BDDs, and the ability of not allocating new BDD variables at each parametrization, enables the simulation not to be bound by the pool of variables available in the BDD software.

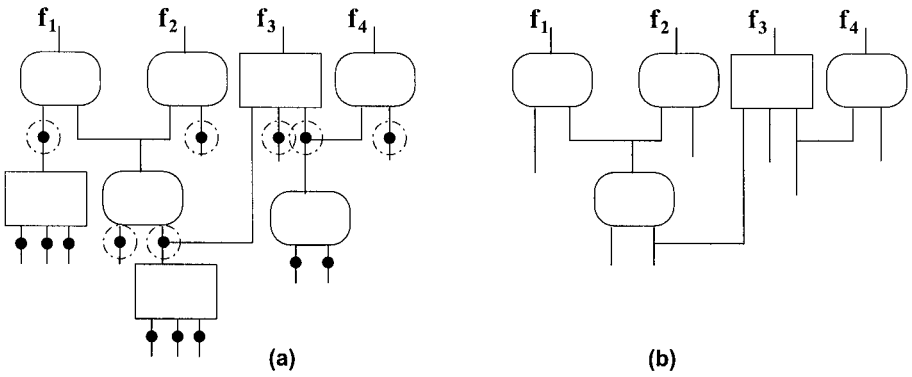


Figure 6.4: Free points elimination for Example 6.1

6.1.2 Elimination of prime functions

As mentioned in Section 4.2, 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 the entire graph rooted at that node, and simply substitute it with a fresh input variable. This simplification can be performed without affecting the range of the state vector.

In order for the substitution to be allowed, 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 fixing the value of 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:

$$\mathbf{F}(a_1, \cdots a_p, r_2, \cdots r_r, \cdots x_m) = \mathbf{G}(r, a_1, \cdots a_p, r_1, \cdots x_m) \circ r(r_1, \cdots r_r) \circ r_1(a_1, \cdots a_p) \quad (6.7)$$

Let's proceed again by computing the $\mathcal{R}(\mathbf{F})$ by recursively splitting on the input variables:

$$\mathcal{R}(\mathbf{F}) = \bigcup_{(i_1, \cdots, i_p) \in \mathcal{B}^p} \mathcal{R}(\mathbf{F}_{a_1=i_1, a_2=i_2, \cdots, a_p=i_p}) \quad (6.8)$$

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 Equation 6.7, we obtain:

$$\mathbf{F}_{a_1=i_1, \cdots, a_p=i_p} = \mathbf{G}_{a_1=i_1, \cdots, a_p=i_p, r_1=i_{r_1}} \circ r_{r_1=i_{r_1}} \quad (6.9)$$

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. 6.9 reduces to:

$$\mathbf{F}_{a_1=i_1, a_2=i_2, \cdots, a_p=i_p} = \mathbf{G}_{a_1=i_1, a_2=i_2, \cdots, a_p=i_p} \quad (6.10)$$

which substituted into Equation 6.8 proves the theorem.

□

A possible structure for the graph $\mathbf{F}$ 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

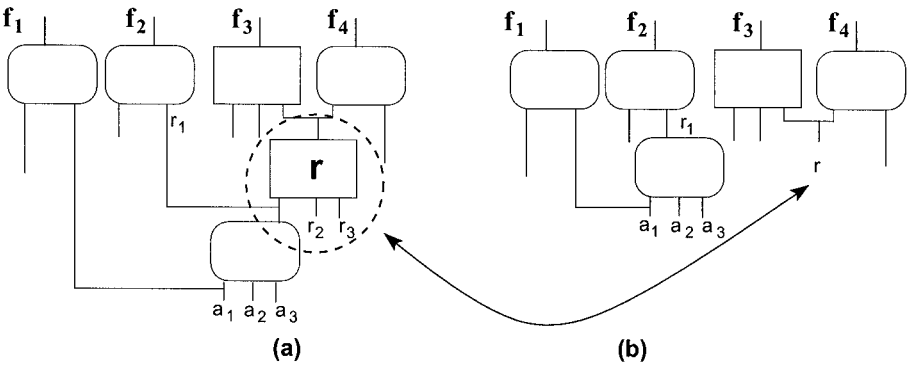


Figure 6.5: General case for prime function elimination: (a) before and (b) after the transformation

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 [Yan91] suite contains the blocks reported in Figure 6.6 in its next-state function at step 10 of symbolic simulation. In the figure, we named the variables from the indices based on the internal association made by the BDD manipulation software, hence that do not correspond to the signal names in the testbench. 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 fan-out, we can completely eliminate this portion of the graph and just substitute it with the input variable r.*

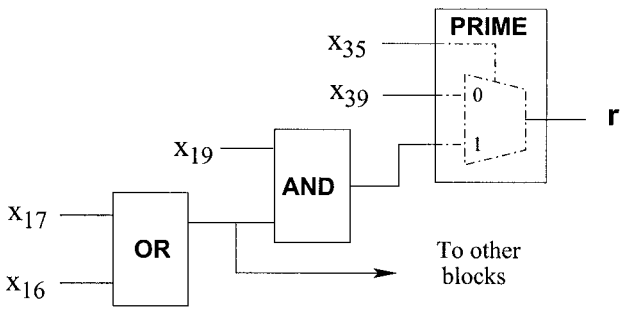


Figure 6.6: Prime elimination in test *s1196* for Example 6.2

6.1.3 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* $\mathbf{F}$. *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 possesses 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 which is a free point, then:*

$$\mathcal{R}(\mathbf{F}) = \mathcal{R}(\mathbf{F}_{v=k}) \tag{6.11}$$

Proof. For a generic function $\mathbf{F}$, we have:

$$\mathcal{R}(\mathbf{F}) = \mathcal{R}(\mathbf{F}_{v=k}) \cup \mathcal{R}(\mathbf{F}_{v=\bar{k}}) \tag{6.12}$$

We now show that under the conditions specified:

$$\mathcal{R}(\mathbf{F}_{v=\bar{k}}) \subseteq \mathcal{R}(\mathbf{F}_{v=k}) \tag{6.13}$$

and Equation 6.12 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:

$$\mathbf{F} = \mathbf{G}(x,y,w,\cdots,x_1\cdots x_x,y_1\cdots y_y,\cdots w_w,\cdots)\circ x(v,p,x_1,\cdots x_x)\circ y(v,q,y_1,\cdots y_y)\cdots \tag{6.14}$$

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 Equation 6.13, 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 $x()$:

$$\mathcal{R}(\mathbf{F}_{v=0}) = \bigcup_{(x_1\cdots x_x)\in\mathcal{B}^x} \mathcal{R}(\mathbf{G}(x,y,\cdots x_1\cdots,y_1\cdots)\circ x(0,p,x_1\cdots)_{x_1=i_{x_1},\ldots}) \tag{6.15}$$

$$\mathcal{R}(\mathbf{F}_{v=1}) = \bigcup_{(x_1\cdots x_x)\in\mathcal{B}^x} \mathcal{R}(\mathbf{G}(x,y,\cdots x_1\cdots,y_1\cdots)\circ x(1,p,x_1\cdots)_{x_1=i_{x_1},\ldots}) \tag{6.16}$$

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:

$$\mathcal{R}(\mathbf{F}_{v=0}) = \mathcal{R}(\mathbf{G}(p,y,\cdots 0\cdots 0,\cdots)) \bigcup_{(x_1,\cdots,x_x)\neq\mathbf{0}} \mathcal{R}(\mathbf{G}(1,y,\cdots x_1\cdots x_x\cdots)_{x_1=i_{x_1},\ldots} \tag{6.17}$$

$$\mathcal{R}(\mathbf{F}_{v=1}) = \mathcal{R}(\mathbf{G}(1,y,\cdots 0\cdots 0,\cdots)) \bigcup_{(x_1,\cdots,x_x)\neq\mathbf{0}} \mathcal{R}(\mathbf{G}(1,y,\cdots x_1\cdots x_x,\cdots)_{x_1=i_{x_1},\ldots} \tag{6.18}$$

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 $XOR(x_1, \cdots, x_x) = 1$:

$$\mathcal{R}(\mathbf{F}_{v=0}) = \bigcup_{XOR(x_1,\cdots x_x)=0} \mathcal{R}(\mathbf{G}(p,y,\cdots x_1 \cdots)_{x_1=i_{x_1},\cdots} \quad (6.19)$$
$$\bigcup_{XOR(x_1,\cdots x_x)=1} \mathcal{R}(\mathbf{G}(\bar{p},y,\cdots x_1 \cdots)_{x_1=i_{x_1},\cdots}$$

$$\mathcal{R}(\mathbf{F}_{v=1}) = \bigcup_{XOR(x_1,\cdots x_x)=0} \mathcal{R}(\mathbf{G}(\bar{p},y,\cdots x_1 \cdots)_{x_1=i_{x_1},\cdots} \quad (6.20)$$
$$\bigcup_{XOR(x_1,\cdots x_x)=1} \mathcal{R}(\mathbf{G}(p,y,\cdots x_1 \cdots)_{x_1=i_{x_1},\cdots}$$

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. □

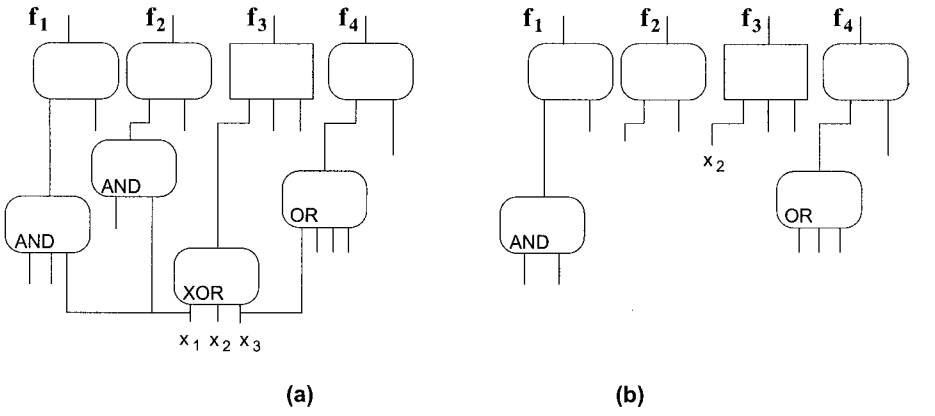


Figure 6.7: Non-dominant variable removal for Example 6.4

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**.

As the previous example pointed out, the three techniques are not exclusive, on the contrary, identifying one situation for a parametric transformation, often generates a parametric decomposition graph which enables additional simplifications. In implementing this solutions, we iterated among the three groups of transformation until convergence.

6.2 The DSD-based simulator

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.

6.2.1 Experimental results

The parametrization techniques presented in the previous section were implemented in a C++ program called DSD-SS. We tested this approach on the largest sequential circuits from the Logic Synthesis Benchmarks suite [Yan91] and the ISCAS'89 Benchmark Circuits [BBK89], including their 1993 additions, as we did for the previous CBSS technique in Section 5.7. 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 DSD-SS to the CUDD package [CUD99] 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 6.1.1-6.1.3. 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, as we presented earlier in Table 5.1: the number of inputs In, outputs Out, memory elements FF, and internal network gates Gates. The subsequent 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 parametrization. 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 parametrization is possible. The values of Table 6.1 indicate that the conditions that allow an exact parametrization 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 parametrization 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 parametrization 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

Table 6.1: DSD-based symbolic simulation (Part 1)

Circuit	In	Out	FF	Gates	Par.techniques			Null	Symbol reductions		Time (s)
					FP	PE	NDV		DSD-SS	PlainSym.	
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

Table 6.2: DSD-based symbolic simulation (Part 2)

Circuit	In	Out	FF	Gates	Par.techniques			Null	Symbol reductions		Time (s)
					FP	PE	NDV		DSD-SS	PlainSym.	
ISCAS '89 - Addition '93 (cont.)											
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

include parametrization. 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 an approximation of the state vector by evaluating symbolic variables to constant values, DSD-SS would attempt first exact parametrization, 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 vari-

able 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 the number of transformations that we were able to complete.

Figure 6.8 compares three simulation techniques: DSD-SS just presented in this chapter, CBSS discussed in Chapter 5 and a plain logic simulator. The comparison is carried in terms of *efficiency*, *i.e.*, input traces visited per second of simulation. In order to evaluate the efficiency of DSD-SS we needed to compute the exact size of the reached state set at each simulation step. This is because DSD-SS does not use a straightforward parametrization as CBSS, where the size of the visited state set is simply a 2-power of the number of Params at each step. DSD-SS uses in general a more involved parametrization where the parametric functions has intra-dependencies, and thus an explicit computation of the reached set is required to evaluate its size. The consequence of this observation is that we could only run the comparison for a few benchmarks, the others not appearing in the graph, could not complete the reached set computation after a few steps. Thus, the reached set computation for DSD-SS is only required to estimate its efficiency for the comparison, it is not part of the algorithm flow, as presented earlier in this section.

As pointed out in the previous chapter and comparison, the fast perfomance of logic simulation is offset by its very limited breadth (only 1 input test vector evaluated per each simulation step). CBSS performs a fast parametrization, which allows it to achieve a good level of parallelism at the cost of extra computation time due to the manipulation of Boolean functions. The time for CBSS is mostly in the simulation of the combinational circuit, where BDD operations can be complex, and not in the parametrization algorithm, which is fairly straightforward. Finally, DSD-SS dwells in a more involved parametrization algorithm, which, however, generates a parametric vector that spans exactly the same range as a pure symbolic simulation approach. The graph shows that DSD-SS has always higher efficiency compared to the other two solutions. In particular, DSD-SS seems to overcome the occasional poor efficiency that CBSS encountered in testbenches not easily parametrizable, where the state vector function where tightly interconnected. Once again, the efficiency benefits seems to grow with the growing complexity of the testbench, unfortunately, as mentioned earlier, we could not compare the most complex designs on this

graph because of the reachability analysis step involved in computing this metrics.

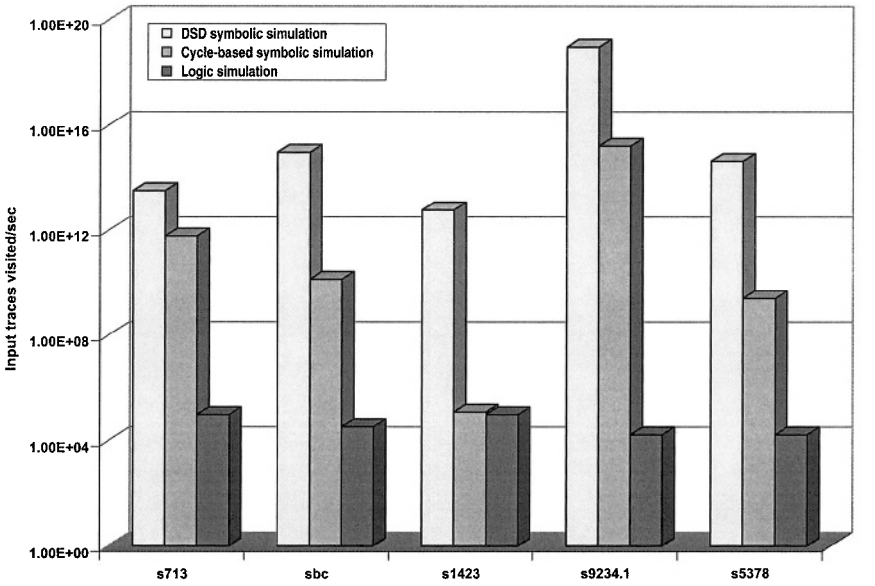


Figure 6.8: Comparison of DSD simulation vs. cycle-based symbolic simulation and vs. logic simulation

The next section presents another parametric solution in symbolic simulation developed by [AJS99] in the context of verification of microprocessors. The solution is an exact parametrization in that it partition the design space in multiple segments and it then proceed to symbolically simulate each of them individually, until all the elements of the partition have been covered.

6.3 Parametrization in the micro-processor domain

A symbolic simulation solution that has been proposed in the context of microprocessor verification is by Robert Jones *et al.* [AJS99, Jon99, Jon02]. In this context the authors would partition the verification search space, and use parametric forms to represent the Boolean constraints to specify each of the subsets.

Microprocessor verification is a classic area of application of verification techniques because it presents many challenges to verification, starting from the sheer complexity of the logic designs and design blocks involved. In addition, because of the large production of general purpose microprocessors, and their deployment in many different domains, from entertainment to life critical activities, guaranteeing the correctness of these designs is a high priority con-

cern in the industry. To address the complexity of the problem, the solution presented by the authors recommends to attack the problem with a divide-and-conquer approach in two fashions:

- **Structural decomposition**. Here, the symbolic simulation problem is partitioned structurally based on the characteristics of the design to be verified. With reference to Example 6.5, if the design is composed of two blocks whose outputs are multiplexed together, a structural partition would first assign symbolic variables to the inputs of one block only, and select the multiplexor to have that block's outputs pass through. Then the multiplexor's selector is switched and the symbolic variables are assigned only to the second block. In general, a structural partitioning would require some architectural understanding of the design in order to be most effective. In the examples presented in [AJS99], the authors discuss how they found an effective partitioning by understanding the overall function implemented by the system and devising clever decompositions of the problem at hand.

- **Data-space decomposition**. This decomposition is performed by case-splitting the problem space, and parametrizing the domain space of each subproblem. In contrast with structural partitioning, case-splitting can be made completely automatic. The partition is performed on individual symbolic variables, where each case-split reduces the exploration space in half. Note, however, that both cases need to be simulated, hence the simulation time increases. The benefit is that case-splitting brings the problem to a manageable size. The parametrization phase consider the domain of each case split and parametrizes the domain so that it is encoded in a compact way. If this phase were not executed, the domain would be represented by a complex characteristic function, usually requiring memory resources beyond those available in the system. A compact parametrization brings the additional advantage that the symbolic simulator manipulates simpler expressions.

It is important to notice that neither of these partitioning techniques actually need fully disjoint partitions. In fact, in the most general case they simply constitute a cover of the entire search space. This aspect could be particularly useful when devising a structural partition, since a cover could lead to more compact problem subsets.

6.3.1 Structural decompositions

Example 6.5. *Consider the schematic design of Figure 6.9. Here two sequential blocks have combinational outputs* $\mathbf{o_A}$ *and* $\mathbf{o_B}$ *connected to a multiplexor controlled by the signal* sel. *A structural partition for the symbolic simulation of this design would consider verifying the two circuits* A *and* B *one at a time.*

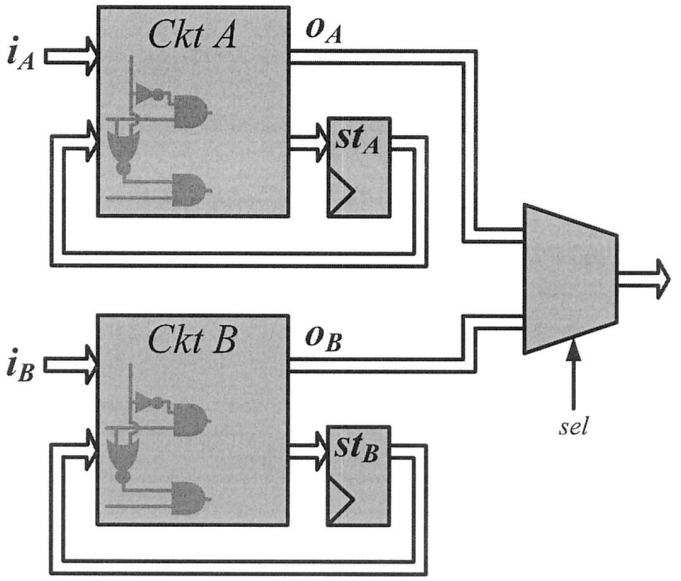


Figure 6.9: Design decomposition for Example 6.5

To do so, the first simulation would generate symbolic variables for each of the combinational inputs $\mathbf{i_A}$*, hold* sel *constant to* 0*, and set the inputs of circuit* A *to constant values. The specific values at* $\mathbf{i_A}$ *are not relevant, given the value at the* sel *input.*

After enough coverage has been obtained for circuit A *with this symbolic simulation run, the setting between the two circuits can be swapped, and the value of* sel *held constant at* 1*. This would enable the second simulator run to provide coverage on circuit* B.

In industrial situations the partitioning can be much more involved than the example. Typical situations includes partitioning on the different modes of operation of a system, which may activate and de-activate entire blocks, or split the range spanned by a multi-bit value into multiple intervals in a way that simplifies the operation of the system (by de-facto partitioning it). Examples of both these types of partitioning are presented in [AJS99].

6.3.2 Parametrization for data-space partitions

Once a data-space partition has generated a tentative number of case-splits, the sub-domain space for a simulation run is encoded in a parametric form. For this phase, the authors target a technique that is efficient, although not optimal. In particular it generates expressions that are usually smaller that the original Boolean vector, but they don't aim at generating the minimum number

of parametric variables. The parametrization algorithm operates on the BDD representation of the function, is recursive and Shannon-decomposition based, similar to the recursive algorithms to perform Boolean manipulation of BDDs. Previous work [JG94] had already suggested a similar technique, however the authors of [AJS99] include additional parametric variables for unconstrained inputs, which have simple representation and increase the span covered by the simulation.

6.4 Summary

This chapter introduced two exact parametrization techniques for symbolic simulation, disjoint-support decomposition-based symbolic simulation and exact parametrizations for industrial applications. We first presented the work on DSD simulation in in [BO02]. Its core contribution is in exploiting the disjoint-support decomposition properties of the state vector in order to generate a compact and exact parametrization during symbolic simulation. The main advantage of this approach is that it is a no-loss transformation. That means that 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 can pure symbolic simulation.

The direction taken by Jones, *et al.*, in [AJS99] takes a step towards aggressively partitioning the search space and traversing it one partition at a time. This technique is also called *case-splitting*. Automatic case-splitting is complemented by a hand-crafted structural partition of the design, which is developed so as to create partitions that de-activate entire portions of the design and focus on verifying only a few. This structural decomposition is key in bringing the complexity to manageable levels, however, it may require a lot of effort from the engineer to devise a good partition.

References

[AJS99] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Formal verification using parametric representations of Boolean constraints. In *DAC, Proceedings of Design Automation Conference*, pages 402–407, June 1999.

[BBK89] 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.

[BO02] Valeria Bertacco and Kunle Olukotun. Efficient state representation for symbolic simulation. In *DAC, Proceedings of Design Automation Conference*, June 2002.

[CBM89] 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.

[CUD99] CUDD-2.3.1. *http://vlsi.Colorado.edu/fabio*, 1999.

[JG94] 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.

[Jon99] Robert Brent Jones. *Applications of Symbolic Simulation to the Formal Verification of Microprocessors*. PhD thesis, Stanford University, August 1999.

[Jon02] Robert B. Jones. *Symbolic Simulation Methods for Industrial Formal Verification*. Kluwer Academic Publishers, 2002.

[Yan91] Saeyang Yang. Logic synthesis and optimization benchmarks user guide, version 3.0. Technical report, Microelectronics Center of North Carolina, January 1991.

Chapter 7

CONCLUSION

This book focuses on recent developments in symbolic simulation, and their subsequent application to digital design verification. The book sets the stage of the discussion by overviewing the process of designing and verifying a digital system and reviewing mainstream techniques for verification, both in functional validation and in formal verification.

Symbolic simulation is presented in detail, along with some of the most closely related symbolic techniques. The central part of this book is devoted to discussing some of the most recent advancements in the area of symbolic simulation. The evolution of this research area is driven by the quest for better performance and scalability in symbolic simulation, where recent and upcoming improvements could, thereby, enable this technique to become a major contributor to the landscape of verification tools deployed in industry.

The main research directions explored by the solutions presented here are heuristic approximations and reparametrization. Key concepts that relate to both topics have been presented in Chapter 4. The book then dives into the exploration of a range of symbolic simulation techniques that move in one, or both, of these directions.

The solutions presented have been evaluated against designs derived from industrial developments,industrial design development with complexity that resembles those of digital design blocks. Among the solutions presented, those developed by us, cycle-based symbolic simulation [BDQ99] and disjoint-support decomposition-based symbolic simulation [BO02], are presented along with experimental results and comparisons to the performance of logic simulation and also to alternative techniques. For the other solutions, that is, quasi-symbolic simulation [WD00] and case-splitting parametrizations [AJS99], relevant experiments have been reported, and the reader is referred to the bibliography to gather the details. Experimental evaluation has indicated that these

solutions are, in fact, sufficiently powerful to enable their use in a broad industrial context.

7.1 Enabling techniques for symbolic simulation

In this work, we presented two core aspects of Boolean functions and their representations used here to develop effective verification techniques based on simulation.

Parametric representations are particularly suited for symbolic simulation, because they condense the information that is carried forward between simulation cycles by removing the redundancy of the original encoding. The key observation here is that the information carried across simulation cycles consists of only the subset of the state space that has been visited thus far. When using reparametrization it is possible to encode the description of said subset in a much more compact fashion than its original form.

Disjoint-support decompositions are an intrinsic property of Boolean functions, which lead to the possibility of representing a function by means of smaller, simpler components that do not share any input signal. The consequence is a structure of small functional blocks loosely connected in a tree graph. These decompositions have been applied to both the synthesis and the verification domains. In this book we exploited the fact that DSDs expose the inherently parallel components in the computation of a function and we then used them to create a compact and efficient parametrization technique for symbolic simulation.

7.2 Scalable symbolic simulation techniques

The solutions presented in the central part of the book use approximations and one, or more, techniques from initial sections to achieve robustness against diverse design sizes and topologies. They all rely on core algorithms that are self-tunable in a feedback loop which evaluates the quality of the simulation run and then adapts the system to respond to a variety of conditions. In addition, they all dedicate a bounded amount of resources and effort to achieve parallelism in the simulation by using symbolic expressions, yet they stand ready to gracefully degrade the symbolic effort if the conditions are not sufficiently favorable.

Along side the specific solutions discussed, all these ingredients are critical to providing solid and scalable verification solutions to an industry that is focused on reliably developing more complex designs in a shorter time frame; an industry that demands push-button solutions adaptable to a wide range of situations. In this direction, the advancements presented in this book are a step towards closing the gap between growing complexity of designs and the ability to verify them.

References

[AJS99] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Formal verification using parametric representations of Boolean constraints. In *DAC, Proceedings of Design Automation Conference*, pages 402–407, June 1999.

[BDQ99] 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.

[BO02] Valeria Bertacco and Kunle Olukotun. Efficient state representation for symbolic simulation. In *DAC, Proceedings of Design Automation Conference*, June 2002.

[WD00] 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.

Appendix A
Disjoint-support decompositions

This appendix complements Chapter 4 in providing a formal introduction to disjoint-support decompositions. A preliminary presentation of this work was discussed in [BD97, Ber03b], while recent improvements include [PB05b, PB05a].

A.1 Function decompositions

We derive here the definition of divisor, and prime function starting from a most general definition of decomposition, not necessarily disjoint support.

Definition A.1. *The operation of* **decomposition** *of* $F : \mathcal{B}^n \rightarrow \mathcal{B}$ *consists of finding other, simpler functions* L *and* $A_1, \cdots, A_k$ *such that:*

$$F(x_1, x_2, .., x_n) = L(A_1(x_1, .., x_n), A_2(x_1, .., x_n), \cdots, A_k(x_1, .., x_n)) \qquad \text{(A.1)}$$

The above definition does not impose that the support of the functions A_i is disjoint, which will be implied by the constraints posed on the characteristics of L. The two following definitions bind the characteristics of L as a divisor of F, and specify the conditions for a function to be prime. If a function F has a proper divisor L, then the component functions that are input of L are disjoint support, as we will prove soon.

Definition A.2. *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:*

$$\begin{aligned} F(x_1, \cdots, x_n) &= L(A_1(x_1, \cdots, x_n), A_2(x_1, \cdots, x_n), \cdots) \qquad \text{(A.2)} \\ \mathcal{S}(A_i) \cap \mathcal{S}(A_j) &= \emptyset; \qquad i \neq j \end{aligned}$$

If $n > k$, *we say that* L **divides** F **properly**. *If* F *cannot be divided properly by any* L, *then it is said to be* **prime**.

In particular, 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 Equation A.2. The list of variables $y_1, \cdots, y_k$ and F/L will be termed **formals list** and **actuals list** of L, respectively. We can now formally define a maximal disjoint-support decomposition:

Definition A.3. *We call a* **disjunctive decomposition** *of F any pair $(L, F/L)$ that satisfies Equation A.2. We distinguish two situations in identifying* **maximal** *decompositions:*

- *If $L = y_1 \otimes y_2 \otimes \ldots y_n$, where $\otimes$ 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. The reader is referred to Section 2.3.1 for a definition of NP-equivalence.

Example A.1. *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.*

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.

A.2 The unique maximal disjoint-support decomposition

This Section shows that, under simple restrictions, every logic function has a unique decomposition tree, and that, each decomposition tree corresponds to a distinct function. In general, it may be expected that any nontrivial function F can be divided by many functions. Moreover, for a given divisor L, one may expect that many different functions could contribute to F/L. Contrary to this, the section proves that the decomposition is unique. The section is organized in two parts: we first show that there is actually a unique prime function L maximally dividing F, then prove that the functions A_i that compose F are also unique. 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 A.3.

In order to show the uniqueness of the maximal DSD, we 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 A.2.3 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 with an associative operator (similar in the sense that 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.

A.2.1 Partitions and representative elements

To prove the results of the section, we need to introduce some auxiliary terminology. The proof of the theorems below require to consider support sets, their partitions imposed by the actuals elements A_i, and the ability to select a "representative" variable from each of the partition subsets. A representative variable for a subset is simply a variable that belong to that subset.

Definition A.4. *Given a set of variables $\mathcal{S}$, a* **partition** *$\mathcal{P}$ of $\mathcal{S}$ is a collection of disjoint subsets of $\mathcal{S}$:*

$$\begin{aligned} &\mathcal{P} = \{\mathcal{S}_1, \mathcal{S}_2, \cdots, \mathcal{S}_k\} && && \\ &\mathcal{S}_i \neq \emptyset; && i = 1, \cdots k && \\ &\mathcal{S}_i \cap \mathcal{S}_j = \emptyset, && i, j = 1, \cdots k; && i \neq j \\ &\bigcup \mathcal{S}_i = \mathcal{S} && && \end{aligned} \tag{A.3}$$

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 $\mathcal{P}$.

A.2.2 Uniqueness of the kernel function

This section proves the first part of the uniqueness results, that is, for any Boolean function F there is a unique prime function, which decompose F. We call this function L, however, notice that now L is not a generic divisor, as in Section A.1, but a prime function. Among all the functions that divide a function F, the prime function L is the one with the smallest number of inputs.

Theorem A.1. *Let L denote a prime function dividing F. Then, L divides any other function M that divides F.*

To prove Theorem A.1, we first need to prove the Lemma below, which shows 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. We will then use the result of the Lemma to prove the uniqueness of the prime function L in Theorem A.1.

Lemma A.2. *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 $|S(M)| > |S(L)|$, M is decomposable.*

Proof. Let $A_1,A_2,\cdots,A_{|S(L)|}$ 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 $P_1,P_2,\cdots,P_{|S(M)|}$.

The starting point of the proof is the presumed equality

$$F = L(A_1,A_2,\cdots) = M(P_1,P_2,\cdots). \tag{A.4}$$

The sets $S(P_1),\cdots,S(P_{|S(M)|})$ form a partition of $S(F)$. 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 $|S(M)| = |\mathcal{X}_M|$.

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,

$$\mathcal{X}_M \cap S(A_1) \neq \emptyset \qquad \text{and} \qquad \mathcal{X}_M \cap S(A_2) \neq \emptyset. \tag{A.5}$$

It is always possible to construct $\mathcal{X}_M$ so that Equation A.5 holds, as follows. If Equation A.5 is not satisfied by a current selection $\mathcal{X}_M$, then it must be (say) $\mathcal{X}_M \cap S(A_2) = \emptyset$. Select a variable x_{A_2} from $S(A_2)$. Notice that x_{A_2} also belongs to the support of some function in F/M, say, to $S(P_1)$. By replacing x_{P_1} with x_{A_2} in $\mathcal{X}_M$ the new set is still a valid selection, and it satisfies Equation A.5.

Consider assigning constant values to the variables not belonging to $\mathcal{X}_M$, in the following way. Since $x_{P_i} \in S(P_i)$, it is always possible to assign values to the remaining variables of $S(P_i)$ 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 Equation A.4, we obtain:

$$L(A_1^*, A_2^*, \cdots) = M(x_{P_1}, x_{P_2}, \cdots) \tag{A.6}$$

Notice that the support of the right-hand side of Equation A.6 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 Equation A.5, 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. Equation A.6 then indicates that L^* divides M and M/L^* is the set of non-constant A_i^*. □

We can now prove Theorem A.1, stating that the prime function L dividing F is unique:

Proof. - Theorem (A.1) - Consider a selection $\mathcal{X}_M$ of $|\mathcal{S}(M)|$ variables and a sensitizing assignment as in the Proof of Lemma A.2. Equation A.4 then reduces to:

$$L(A_1^*, A_2^*, \cdots) = M(x_{P_1}, x_{P_2}, \cdots). \tag{A.7}$$

(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 Equation A.7 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, Equation A.7 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 A.2, L would then be decomposable, against the assumption. None of the A_i^* of Equation A.7 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, Equation A.7 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 A.2. *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 the function* $MAJORITY(a,b,c)$ *is prime by exhaustive enumeration. Hence, from Theorem A.1, F cannot be decomposed with any prime function L other than MAJORITY, while maintaining arguments with disjoint support. It follows that MAJORITY is the unique kernel of F.*

The following Corollary is a direct application of the previous Theorem and it is reported here because, as we will see, it has relevant application in the the process of constructing the disjoint-support decomposition of a function.

Corollary A.3. *If a function F can be decomposed by the two input function* OR_2 *(*AND_2 *,* XOR_2 *) of two disjoint-support functions, then it cannot be decomposed using any of the other two operators.*

We then show an important property of such prime function, namely, that it divides any other function M that divides F.

A.2.3 Uniqueness of the actuals list

The second part of the uniqueness results consists of showing that the elements of the actuals list, F/K_F, are also unique. We provide here a characterization of the functions in F/K_F, by analyzing the two possible cases one at a time: Theorem A.4 below considers the case where the prime L is not an associative operator. The reader may recall that this implies that $|\mathcal{S}(L)| > 2$. As 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 A.5.

Theorem A.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*

$$L(A_1,A_2,\cdots) = L(B_1,B_2,\cdots) \tag{A.8}$$

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

$$(A_1,A_2,\cdots) = \mathbf{NP}(B_1,B_2,\cdots). \tag{A.9}$$

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 Equation A.8 but violate Equation A.9, 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 to 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 $S(A_1) = S(B_1); S(A_2) = S(B_2), \cdots$. Consider any assignment that is complete for the variables in $S(A_2), S(A_3), \cdots$, but that does not assign any values to those variables in $S(A_1)$. In this case, all functions except A_1 and B_1 reduce to constants. Let $a_2, a_3, \cdots$ denote the constant values $A_2^*, A_3^*, \cdots$. Equation A.8 reduces to

$$L(A_1, a_2, a_3, \cdots) = L(B_1, b_2, \cdots) . \tag{A.10}$$

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

$$L(A_1, a_2, \cdots) = A_1 \qquad \text{or} \qquad L(A_1, a_2, \cdots) = \overline{A_1}. \tag{A.11}$$

In this case, Equation A.10 becomes

$$A_1 = L(B_1, b_2, \cdots) \qquad \text{or} \qquad A_1 = \overline{L}(B_1, b_2, \cdots). \tag{A.12}$$

Since A_1 is, by construction, not a constant, Equation A.12 indicates that expression $L(B_1, b_2, \cdots)$ 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, Equation A.12 ultimately implies that

$$A_1 = B_1 \qquad \text{or} \qquad A_1 = \overline{B}_1. \tag{A.13}$$

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 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 X_B from S_B containing at least two variables from $S(A_2)$. We impose one more requirement on X_B, namely, that for at least another function $A_i, i \neq 2$ $X_B \cap S(A_i) \neq \emptyset$.

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

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

Consider applying a partial assignment, such that all functions $B_1, B_2, \cdots$ reduce to variables in X_B or their complements: $B_i^* = x_{B_i}$. Since no variables of A_1 are included in X_B, A_1 reduces to a constant a_1, and Equation A.8 becomes

$$L(a_1, A_2^*(x_{B_1}, x_{B_2}), A_3^*, \cdots) = L(x_{B_1}, x_{B_2}, \cdots) \tag{A.14}$$

Notice that, since the left-hand side of Equation A.14 must have support 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^* :

$$L^*(A_2^*(x_{B_1},x_{B_2}),A_3^*,\cdots) = L(x_{B_1},x_{B_2},\cdots) \tag{A.15}$$

Equation A.15 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 $|S(L)| > 2$ is essential to the proof, for if $|S(L)| = 2$, Equation A.14 reduces to

$$L(a_1,A_2^*(x_{B_1},x_{B_2})) = L(x_{B_1},x_{B_2}) \tag{A.16}$$

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

Example A.3. *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 A.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-equivalent set.*

Example A.4. *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 A.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 A.4 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 [BD96a] :

Theorem A.5. *Suppose a function* F *is decomposable using one binary associative operator* $\otimes$ *(where* $\otimes = AND,OR,XOR$*) as*

$$F = A_1 \otimes A_2 \otimes \cdots \otimes A_n \tag{A.17}$$

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 :

$$A_1 + \cdots + A_n = B_1 + \cdots + B_q \ . \tag{A.18}$$

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 $S(B_1)$ have not been assigned any value. Corresponding to this partial assignment, Equation A.18 becomes:

$$A_1^* + \cdots + A_n^* = B_1 \tag{A.19}$$

In Equation A.19, 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 Equation A.19.

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

$$S(B_1) = S(A_1^*) \subset S(A_1) \qquad \text{strictly.} \tag{A.20}$$

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

Consider a second assignment, zeroing all functions $A_i, i \neq 1$. Equation A.18 now reduces to

$$A_1 = B_1^* + \cdots + B_q^* \tag{A.21}$$

By the same reasonings carried out so far, the r.h.s. of Equation A.21 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 Equation A.20 one would have

$$\mathcal{S}(A_1) = \mathcal{S}(B_j^*) \supset \mathcal{S}(B_1) \tag{A.22}$$

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 Equation A.20, we get

$$\mathcal{S}(A_1) = \mathcal{S}(B_1^*) \subset \mathcal{S}(B_1) \qquad \text{strictly} \tag{A.23}$$

which contradicts Equation A.20. 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 Equation A.20 and A.22 are still valid. From the two equations we can then derive the contradiction. □

The Corollary below indicates how the decomposition of Theorem A.5 is the common denominator of all the other decompositions through an associative operator:

Corollary A.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*

$$F = B_1 \otimes B_2 \otimes \cdots \otimes B_q. \tag{A.24}$$

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

$$B_j = A_{k_j+1} \otimes \cdots A_{k_{j+1}} \qquad \textit{with} \qquad k_1 = 0, k_q = k. \tag{A.25}$$

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

$$F = B_1 + B_2 = A_1 + \cdots + A_k. \tag{A.26}$$

Consider two distinct partial assignments leading to $B_1 = 0$ and to $B_2 = 0$, respectively. Equation A.26 becomes

$$B_2 = A_1^* + \cdots + A_k^*; \tag{A.27}$$

and

$$B_1 = A_1^{**} + \cdots + A_k^{**}. \tag{A.28}$$

By computing the *OR* of the two components,

$$F = B_1 + B_2 = A_1^* + A_1^{**} + A_2^* + A_2^{**} + \cdots + A_k^* + A_k^{**} \tag{A.29}$$

From Theorem A.5, there can be at most k terms in the right-hand side of Equation A.29. For each function A_i, at least one of A_i^*, A_i^{**} must be nonzero (or otherwise $S(A_i) \cap S(F) = \emptyset$). 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 Equation A.27 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, Equations. A.27 and A.28 reduce to Equation A.25. □

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 A.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.

A.3 The canonical decomposition tree

In Sections A.2.2 and A.2.3, 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 (see also Section 2.3.1). 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, or canonical, 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 BDD representations ([BRB90]). 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 A.5. *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 BDD 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 BDD 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 A.1 and A.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.

A.3.1 Extracting all decompositions from the canonical tree

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 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 the decomposition algorithm.

Definition A.6. *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 A.5. *Consider the function* $F = x_1x_2x_3 + x_4x_5$. *A decomposition tree is reported in Figure A.1.a. Figure A.1.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 A.6. 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.*

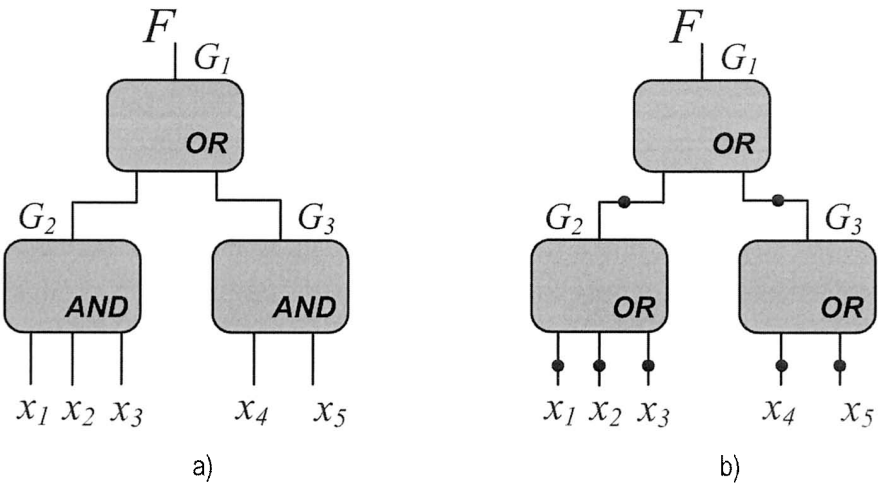


Figure A.1: Decomposition tree for Example A.5

Notice that if a function G appears in $DT(F)$, then every function in $DT(G)$ will also appear in $DT(F)$. The following definition and lemma will enable us

to extract all the disjoint-support decompositions of a function from its normal decomposition tree by considering all the *cuts* of the tree. As a result, the normal DT can be viewed as a compact way to encode all disjoint decompositions.

Definition A.7. *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 $S(A_i) \cap S(A_j) = \emptyset; \qquad i \neq j$

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

Example A.6. *A possible cut for the function of Example A.5 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 A.7. *For every function* M *dividing* F, F/M *is a cut of* $DT(F)$. *Conversely, for any cut* C *of* $DT(F)$, *there is a function* M *such that* $F/M = C$.

Proof. The first part of the theorem is proved by induction on the number of variables in $S(F)$. The base of the induction (when $|S(F)| = 2$) is trivial.

For the generic induction step, let $m = |S(M)|$ 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:

$$F = M(P_1, P_2, \cdots, P_m). \tag{A.30}$$

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 A.1, 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

$$M = L(B_1, \cdots, B_l). \tag{A.31}$$

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

$$S(B_i) = \{y_{b_{i-1}+1}, \cdots, y_{b_i}\} \qquad i = 1, \cdots, l \tag{A.32}$$

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 Equations. A.30 and A.31 one obtains

$$F = M(P_1, \cdots, P_m) = L(B_1(P_1, \cdots, P_{b_1}), \cdots, B_l(P_{b_{l-1}+1}, \cdots, P_m)) \tag{A.33}$$

Since L divides F and it is a prime function, from Theorem A.1 it must be

$$A_i = B_i(P_{b_{i-1}+1}, \cdots, P_{b_i}) \qquad i = 1, \cdots, l \tag{A.34}$$

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, Equation A.34 indicates that A_i is decomposed by B_i. Since $|S(A_i)| < |S(F)|$, 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

$$M = G_1(y_1, \cdots, y_{m_1}) \otimes G_2(y_{m_1+1}, \cdots, y_m) \tag{A.35}$$

for suitable functions G_1, G_2. By substituting the formals y_i with the actuals P_i in Equation A.35 , and taking into account Theorem A.5,

$$F = G_1(P_1, \cdots, P_{m_1}) \otimes G_2(P_{m_1+1}, \cdots, P_m) = A_1 \otimes A_2 \otimes \cdots \otimes A_k \tag{A.36}$$

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 A.36 is the subject of Corollary A.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

$$G_1(P_1, \cdots, P_{m_1}) = A_1 \otimes \cdots \otimes A_{k_1} \quad 2 \leq k_1 < k. \tag{A.37}$$

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 $DT(F)$. Equation A.37 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 C. □

Example A.7. *Consider the function F given previously in Example 4.5. 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 C of* $DT(F)$ *with reference to the Example is* $C = \{a, b, H, I\}$.

A.4 Building the decomposition tree from a BDD

This section presents in detail the various components of the decomposition algorithm that have been introduced in Section 4.3.1, and a proof of the correctness of this construction.

As a reminder, the objective here is to decompose a function F, whose root node is labeled by z, and produce $DT(F)$. The decompositions of the two cofactors of F w.r.t to z are assumed to be known, and have decomposition trees $DT(F_0)$ and $DT(F_1)$. In addition, we assume that, for any decomposition, the actuals list members are ordered so that the top variable of each A_i has

a decreasing index when going from A_1 to A_k. With this assumption, we can easily derive that the decomposition of F will contain an actuals list member A_1 whose support includes the variable z: $z \in S(A_1)$. Section 4.3.1 indicates that the construction can be split into two groups: inherited and new decompositions, as reported by Definition 4.2.

Inherited decompositions can be classified further into smaller subgroups, as the list below illustrates. Let $A_{10} = A_1(z = 0)$ and $A_{11} = A_1(z = 1)$, then an inherited decomposition must fall into one of the following cases:

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 $\bar{z}$ – 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 $\bar{z}$) 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 analyze each case in turn: Sections A.4.1 to A.4.3 cover all the three subtypes of inherited decompositions, Section A.4.4 analyzes new decompositions. Each Section shows how to determine to which scenario a Shannon decomposition belongs to, and how to construct $DT(F)$ from $DT(F_0)$ and $DT(F_1)$, for its specific condition.

A.4.1 Case 1. Neither A_{10} nor A_{11} is constant and $A_{10} \neq \overline{A_{11}}$

This case was implicitly described in Example 4.7. 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:

A.4.1.1 Case 1.a - *PRIME* decomposition

Lemma A.8. *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*

$$F_0(G=0)=F_1(H=0) \quad \textit{and} \quad F_0(G=1)=F_1(H=1) \tag{A.38}$$

or

$$F_0(G=0)=F_1(H=1) \quad \textit{and} \quad F_0(G=1)=F_1(H=0) \tag{A.39}$$

must hold.

Moreover, if Equation A.38 holds, then F/K_F is obtained from F_0/K_{F_0} by replacing G with $A_1=\bar{z}G+zH$, else by replacing G with $A_1=\bar{z}G+z\bar{H}$.

Notice that Lemma A.8 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 Equation 4.7 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:

$$F_0(A_{10}=0) = K_{F_0}(G=0,A_2,\cdots,A_l)=K_{F_1}(H=0,A_2,\cdots,A_l)=F_1(A_{11}=0) \tag{A.40}$$
$$F_0(A_{10}=1) = K_{F_0}(G=1,A_2,\cdots,A_l)=F_1(A_{11}=1). \tag{A.41}$$

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

$$F_0(A_{10}=0) = K_{F_0}(G=0,A_2,\cdots,A_l)=K_{F_1}(H=1,A_2,\cdots,A_l)=F_1(A_{11}=1) \tag{A.42}$$
$$F_0(A_{10}=1) = K_{F_0}(G=1,A_2,\cdots,A_l)=F_1(A_{11}=0). \tag{A.43}$$

Hence, Equations A.40 and A.42 reduce to Equation A.38 and A.39.

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 Equation A.38 holds:

$$F_1 = \overline{H}F_1(H=0) + HF_1(H=1) = \overline{H}F_0(G=0) + HF_0(G=1) \quad \text{(A.44)}$$

Using the decomposition of F_0 in Equation A.44 :

$$F_1 = HK_{F_0}(G=0, A_2, \cdots, A_l) + HK_{F_0}(G=1, A_2, \cdots, A_l) = K_{F_0}(H, A_2, \cdots, A_l). \quad \text{(A.45)}$$

Equation A.45 indicates that K_{F_0} divides F_1 as well, hence F_0 and F_1 have the same decomposition type. From Equation A.38 it also follows that

$$F = zF_0 + zF_1 = zK_{F_0}(G, A_2, \cdots, A_l) + zK_{F_0}(H, A_2, \cdots, A_l) = K_{F_0}(zG + zH, A_2, \cdots, A_l) \quad \text{(A.46)}$$

Equation A.46 indicates precisely that F has *PRIME* decomposition (its kernel being K_{F_0}), and that $F/K_F = \{\bar{z}G + zH, A_2, \cdots, A_l\}$, which is what we needed to prove.

The case where Equation A.39 holds can be handled in the same way, just by replacing H with $\overline{H}$. □

Example A.8. *The function $F = azb + e\bar{z}b + cb \oplus d$ has kernel $K_F(x_1, x_2, x_3, x_4) = x_1x_2 + x_3x_2 \oplus x_4$ and actuals list $(az + e\bar{z}, b, c, d)$. 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 A.8, we can find the decomposition of F from their kernels and actuals lists.*

A.4.1.2 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, it is shown that $K_{F_0} = OR_{k-1+l}$, $F_0/K_{F_0} = \{B_1, \cdots, B_l, A_2, \cdots, A_k\}$ 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 A.9 below:

Lemma A.9. *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,

$$F = z'F_0 + zF_1 = OR_k(zOR_l(B_1, \cdots, B_l) + zOR_m(C_1, \cdots, C_m), A_2, \cdots, A_k) \tag{A.47}$$

We need to show now that $\bar{z}OR_l(B_1, \cdots, B_l) + zOR_m(C_1, \cdots, C_m)$ 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, Equation A.47 indicates that F has a OR_k decomposition. □

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

A.4.2 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.

A.4.2.1 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:

$$F_0 = K_F(0, A_2, \cdots, A_l) \quad \text{and} \quad F_1 = K_F(A_{11}, A_2, \cdots, A_l) \tag{A.48}$$

Equation A.48 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 A.10. *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 Equation A.48 indicates that F_1 has the same kernel as F. The second point also follows immediately from Equation A.48, 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 A.10 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.

A.4.2.2 Case 2.b - Associative decomposition

Since we assumed at the beginning of Section A.4.2 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 :

$$\begin{aligned} F &= OR_k(zA_{11},A_2,\cdots,A_k) \quad k \geq 2 \\ F_0 &= OR_{k-1}(A_2,\cdots,A_k) \qquad \text{(A.49)} \\ F_1 &= OR_k(A_{11},A_2,\cdots,A_k) = OR_{k+l-1}(B_1,\cdots,B_l,A_2,\cdots,A_k) \qquad \text{(A.50)} \end{aligned}$$

Equation A.50 indicates that F_1 will also have an *OR* decomposition. F_0, however, may have a different decomposition: in fact, in the special case $k = 2$,

Equation A.49 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 A.11. *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:

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

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

A.4.3 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.

A.4.3.1 Case 3.a - *PRIME* decomposition

If F has *PRIME* decomposition, then

$$F_0 = K_F(A_{10}, A_2, \cdots, A_l) \quad \text{and} \quad F_1 = K_F(\overline{A_{10}}, A_2, \cdots, A_l) \tag{A.51}$$

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 A.3). Another consequence of Equation A.51 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 A.12. *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:*

$$F_0(H=0) = F_1(H=1) \quad \textit{and} \quad F_0(H=1) = F_1(H=0) \tag{A.52}$$

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 Equation A.52 holds, then:

$$\begin{aligned} F_1 &= HF_0(H=1) + HF_0(H=0) \\ F &= zF_0 + zF_1 = zHF_0(H=0) + zHF_0(H=1) + zHF_0(H=1) + zHF_0(H=0) \\ &= (zH + zH)F_0(H=0) + (zH + zH)F_0(H=1) = F_0(H = z \oplus H). \end{aligned}$$

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 A.12 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 Equation A.52 is verified.

A.4.3.2 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 A.4.1.2 and A.4.2.2 for the other cases. Therefore, we omit it from the present analysis.

A.4.4 New decompositions

We now consider the case where $A_1 = \bar{z}$ 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.

A.4.4.1 Case NEW.a - *AND* or *OR* decomposition

$F = OR(z, G)$, then the two cofactors are $F_z = G$ and $F_{\bar{z}} = 1$. Conversely, if $F_{\bar{z}} = 1$, then $F = F_z\bar{z} + 1z = OR(z, F_z)$. 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 $\bar{z} \in F/OR$ if and only if the cofactor $F_{\bar{z}}$ 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:

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

A.4.4.2 Case NEW.b - *XOR* decomposition

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

A.4.4.3 Case NEW.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 $F = K_F(z, A_2, A_3, \cdots, A_l)$:

$$F_z = K_F(0, A_2, A_3, \cdots) \qquad F_{\bar{z}} = K_F(1, A_2, A_3, \cdots) \tag{A.53}$$

Let $L_{y_1}(y_2, \ldots, y_m)$ and $L_{\bar{y}_1}(y_2, \ldots, y_m)$ denote the functions $K_F(0, y_2, .., y_m)$ and $K_F(1, y_2, .., y_m)$, respectively. Equation A.53 can then be written as

$$F_z = L_{y_1}(A_2, A_3, \cdots) \qquad F_{\bar{z}} = L_{\bar{y}_1}(A_2, A_3, \cdots) \tag{A.54}$$

In general, L_{y_1} and $L_{\bar{y}_1}$ 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 $F_z, F_{\overline{z}}$. However, from Equation A.54, L_{y1} divides F_z. From Lemma A.7, the set of functions $\{A_2, A_3, \cdots\}$ forms a cut of $DT(F_z)$ and thus F/K_F also contains a cut of the same decomposition tree. Similar reasoning applies to $F_{\overline{z}}$.

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 A.8. *Given a function F and a variable $z \in S(F)$, a function A appearing in $DT(F_z)$ or in $DT(F_{\overline{z}})$ is said to have* **uniform-support** *if it has positive polarity and exactly one of the following is true:*

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

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

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

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

For a given pair of decomposition trees $DT(F_z)$, $DT(F_{\overline{z}})$, we denote by $Max(F_z, F_{\overline{z}})$ the set of maximal uniform support functions. It is this set of functions, together with the top variable z, that we will use as the actuals list for the decomposition of F. Theorem A.13 shows that this is the correct set of functions for F/K_F.

Example A.9. *Consider the function F of Figure A.2. 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 $Max(F_z, F_{\overline{z}})$ for this function is $\{x_1 + x_2, x_3, x_4 x_5, x_6\}$. Notice that $x_1 + x_2$ appears implicitly in $DT(F_0)$ by rule (2) of Definition A.6, 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 $Max(F_z, F_{\overline{z}})$ effectively represents the actuals list of F. This is stated by the following Theorem:

Theorem A.13. *For a function F with decomposition $F/K_F = \{z, A_1, \cdots, A_l\}$, the actuals list is given by $\{z\} \cup Max(F_z, F_{\overline{z}})$.*

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

Example A.10. *Based on Theorem A.13, the actuals list for the decomposition of the function in Figure A.2 is given by $\{z, x_1 + x_2, x_3, x_4 x_5, x_6\}$. The*

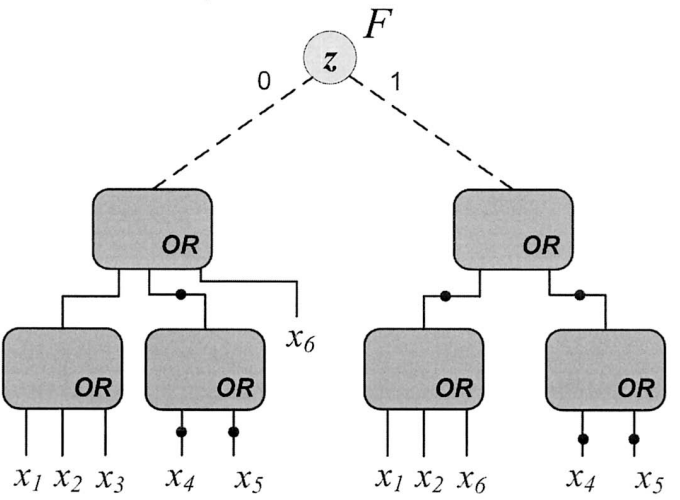


Figure A.2: Example of a PRIME decomposition.

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 A.13 requires the proof of some properties of uniform-support functions.

Lemma A.14. *Any two maximal uniform-support functions of* $DT(F_z)$ *or* $DT(F_{\overline{z}})$ *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 $DT(F_z)$ and the other only appeared in $DT(F_{\overline{z}})$, 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:

$$A_1 = \otimes_{i=1}^{k} B_i \qquad k \geq 2 \tag{A.55}$$

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

$$Q_1 = \otimes_{i=1}^{m} B_i \quad 2 \leq k < m \tag{A.56}$$

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

$$A_2 = \oslash_{i=1}^{l} C_i. \tag{A.57}$$

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

$$Q_2 = \oslash_{i=1}^{n} C_i \quad 2 \leq l < n. \tag{A.58}$$

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 Equation A.57, $\mathcal{S}(A_1) \subseteq \mathcal{S}(C_i) \subseteq \mathcal{S}(A_2)$, 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 Equation A.55 must coincide with $\oslash$, and the functions C_i in Equation A.58 must coincide with the functions B_i in Equation A.56. Hence, A_2 can be written as

$$A_2 = \otimes_{i=j}^{l} B_i \quad \text{for} 1 \leq j \leq k < l \leq m. \tag{A.59}$$

Consider then the function

$$U = \otimes_{i=1}^{l} B_i. \tag{A.60}$$

U contains all the functions in the decomposition of $A_1/\otimes$ and of $A_2/\otimes$. Hence, U has uniform support, and $S(U) \supset S(A_1), S(U) \supset S(A_2)$, 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 A.15. *The set $Max(F_z, F_z)$ contains a cut of $DT(F_z)$ and of $DT(F_z)$.*

Proof. We only prove that $Max(DT(F_z), DT(F_z))$ contains a cut of $DT(F_z)$, the second part being entirely symmetrical.

Consider the collection C of functions $A_i \in Max(DT(F_z), DT(F_z))$ 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 A.14, they are disjoint-support. Therefore,

$$\bigcup_{A_i \in C} S(A_i) \subseteq S(F_z). \tag{A.61}$$

It remains to be shown that the containment relation A.61 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 $Max(DT(F_z), DT(F_z))$ and therefore x_i must belong to the left-hand side of Equation A.61. 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 A.9. *Given a function F and a variable $z \in S(F)$, a collection of uniform support functions (not necessarily maximal) $C_2 = \{A_i\}$ is termed a* **bi-cut** *if the following holds:*

1 $S(A_i) \cap S(A_j) = \emptyset$ *for* $i \neq j$;

2 C_2 *contains a cut of $DT(F_z)$ and of $DT(F_z)$.*

Example A.11. *Consider a function F such that $F_z = (x_1 + x_2)x_4$ and $F_z = (x_1 + x_2 + x_3)x_5$ as in Figure A.3 (we present a non-normal decomposition tree for improved readability). A possible bi-cut for such function is $C_2 = \{x_1 + x_2, x_3, x_4, x_5\}$. Note that the set $C = \{x_1 + x_2 + x_3, x_4, x_5\}$ is not a bi-cut since it does not contain a cut of $DT(F_z)$.*

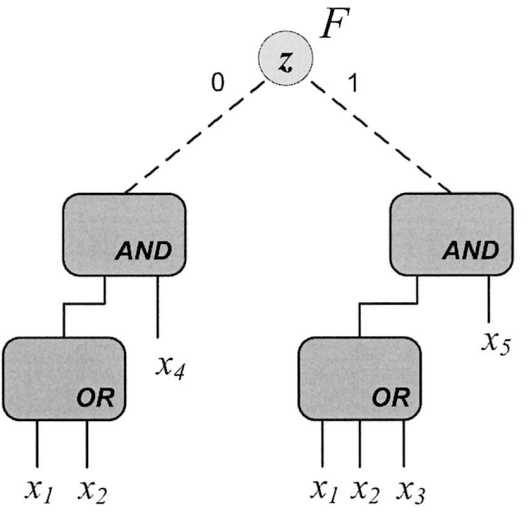


Figure A.3: Function for Example A.11.

From Lemma A.15, $Max(DT(F_z), DT(F_{\bar{z}}))$ is a bi-cut. It is also straightforward to verify that $Max(DT(F_z), DT(F_{\bar{z}}))$ 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 A.13.

Lemma A.16. *Let M denote any function dividing F, such that expression $F/M = \{z, A_2, \cdots, A_m\}$. Then, the subset $C_2 = \{A_2, \cdots, A_m\}$ is a bi-cut of F w.r.t. z. Conversely, for each bi-cut C_2 there exists a function M such that $F/M = \{z\} \cup C_2$.*

Proof. Equation A.54 shows that C_2 contains a cut of $DT(F_z)$ and of $DT(F_{\bar{z}})$. The functions A_i are all disjoint-support, and each of them appears in at least one of $DT(F_z), DT(F_{\bar{z}})$ (or else F would be independent from the variables in $S(A_i)$). 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, $S(A_2)$) is not uniform. It is not restrictive to assume that A_2 appears in $DT(F_z)$. Then $S(A_2) \subseteq S(F_z)$. Since we take A_2 to be not uniform, it must be

$$S(A_2) \cap S(F_z) \neq \emptyset \tag{A.62}$$

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

$$S(A_2) \cap \overline{S(F_z)} \neq \emptyset \tag{A.63}$$

otherwise A_2 would be uniform by condition 2. Let C indicate a subset of $\mathcal{C}_2$ forming a cut of $DT(F_z)$; it follows that

$$S(C) = \bigcup_{A_i \in C} S(A_i) = S(F_z); \tag{A.64}$$

The last equality being valid by definition of cut.

From Equation A.63, if $A_2 \in C$, then $S(C) \cap \overline{S(F_z)} \neq \emptyset$, contradicting Equation A.64. Hence, A_2 cannot belong to C. We now show that A_2 cannot be left out of C either: From Equation A.62, there is a variable x in $S(A_2) \cap S(F_z)$. 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 C, $x \notin S(C)$ and 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_z. From Lemma A.7, there exists a function $L_0(y_2, \cdots, y_{c_0})$ such that $F_z/L_0 = \mathcal{C}_0$. 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

$$L(z, A_2, \cdots, A_m) = F, \tag{A.65}$$

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

Finally, the proof of Theorem A.13 follows:

Proof. - Theorem (A.13) - From Lemma A.16, a function L can be found such that $F/L = \{z\} \cup Max(F_z, F_z)$. Then, from Theorem A.1, F/K_F cannot contain more elements than $\{z\} \cup Max(F_z, F_z)$. Since $Max(F_z, F_z)$ 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 A.1 and A.4, K_F must be NP-equivalent to L and F/K_F must coincide with $\{z\} \cup Max(F_z, F_z)$, modulo NP-equivalence. □

A.5 The DEC procedure

Section 4.3 presents a decomposition procedure which implement the algorithm just presented. The procedure operates bottom-up on the BDD representation of the function to be decomposed and, at each node, constructs the decomposition of the root node from the already generated decompositions of its two cofactors. The decomposition procedure, called DEC, solve the problem by considering a list of cases which together cover all the possible situations

that may arise. These cases match one by one all the situations that we considered in Section A.4. Section 4.3 overviews the high level structure of the procedure, here we provide a presentation of the sub-procedures developed for each of those cases.

We showed in Section 4.3 that the `decompose` functions breaks the problem into discovering an inherited decomposition or a new decomposition. The pseudo-code for those two sub-functions whose given in Figures 4.10 and 4.11.

A.5.1 Inherited decompositions

This section discusses all the internal calls of `decompose_INHERITED` as outlined in Figures 4.10, the next section will present `decompose_NEW`.

A.5.1.1 *OR* decompositions

`decompose_INHERITED_OR_123.b` groups the constructions described in Sections A.4.1.2, A.4.2.2 and A.4.3.2 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 A.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 A.11 shows how to treat this situation in its special case of $k = 2$. Figure A.4 illustrates this presentation with the pseudo-code of the routine.

A.5.1.2 *XOR* decompositions

Inherited *XOR* decompositions can arise only from Cases 1.b and 2.b of Section A.4.1.

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.

```
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;
}
```

Figure A.4: Pseudo-code for `decompose_INHERITED_OR_123.b`

A.5.1.3 *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 A.12. *Consider again the function of Example 4.5 and assume that the top variable in its BDD representation was g. We consider available the*

decompositions of the cofactors w.r.t. g:

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

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 Equations A.38 or A.39 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 pseudo-code in Figure A.5 checks if Equations A.38 or A.39 hold. It returns the decomposition of F if the tests are successful.

Case 2.a has a more complex set of comparisons. As the reader may recall from Section A.4.2.1, Lemma A.10 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 $S(A_i) \cap S(F_0) = \emptyset$.

Thus we can detect such decomposition by considering the generalized cofactors (see Definition 2.6) 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 $2 \cdot n$ cofactors, where n is the number of candidate elements.

```
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;
}
```

Figure A.5: Pseudo-code for `decompose_INHERITED_PRIME_1.a`

Example A.13. *Consider the functions $F_z = ITE(A, CD, B+C), F_z = CD$. The actuals list of F_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, $F_z = A + CD \neq F_z$, and that assigning $B = 0$ results in $F_z = C(A + D) \neq F_z$. The function B is then discarded. Assigning $A = 1$ instead results in $F_z = ITE(1, CD, B + C) = CD = F_z$. A new function $Z = A + z$ is constructed, and F is decomposed as $ITE(Z, CD, B + C)$.

The pseudo-code in Figure A.6 reflects the observations above.

Case 3.a can be carried out similarly 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 A.12 can be any of the actuals list elements.

```
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
}
```

Figure A.6: Pseudo-code for `decompose_INHERITED_PRIME_2.a`

A.5.2 New decompositions

The detection of new decompositions is also subdivided into matching a series of different situations. The presentation, and all the cases, follow the analysis in Section A.4.4. For a high level view of how all these situations are plugged in together, the reader is referred to the presentation of `decompose_NEW` as outlined in Figures 4.11.

A.5.2.1 *OR* and *XOR* decompositions

`decompose_NEW_OR` and `decompose_NEW_XOR` implement the checks of Sections A.4.4.1 and A.4.4.2. 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 decomposi-

tion node will have an actuals list that is the same of its cofactor with the new element z prepended.

A.5.2.2 *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 A.13.

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 A.14. *Figure A.7 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 tree $Max(F_0, F_1)$ and label the root node with type `PRIME`.

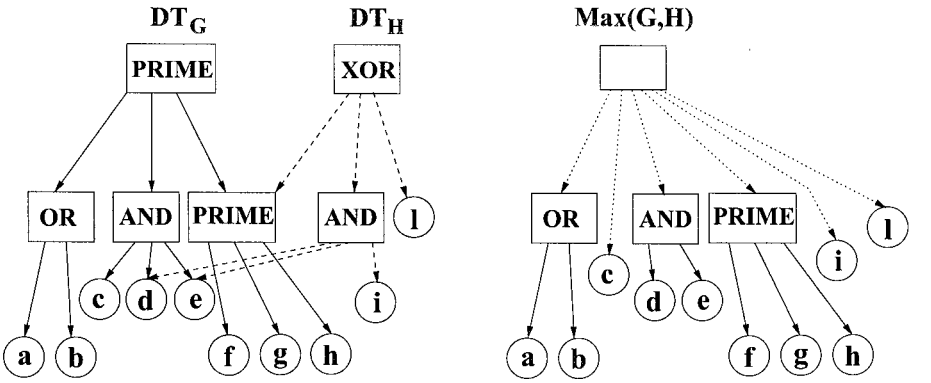


Figure A.7: Two functions and the construction of their $Max(G,H)$ tree.

Example A.15. *Consider the case where* $F_0 = ITE(abc, d+e+f, g \oplus h)$, $F_1 = ITE(ab, e+f+g, h \oplus c)$. *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 $F = \bar{z}F_0 + zF_1$ *is given by* $F = K_F(z,A,E,c,d,g,h)$.

This concludes the presentation of disjoint-support decompositions. The material that was presented in this Appendix complements the discussion in Chapter 4 and provides a formalization of the concepts presented there.

References

[BD96] 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.

[BD97] 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.

[Ber03] Valeria Bertacco. *Achieving Scalable Hardware Verification with Symbolic Simulation*. PhD thesis, Stanford University, 2003.

[BRB90] 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.

[PB05a] Stephen Plaza and Valeria Bertacco. Boolean operations on decomposed functions. In *International Workshop on Logic Synthesis*, pages 310–317, June 2005.

[PB05b] Stephen Plaza and Valeria Bertacco. STACCATO: Disjoint support decompositions from BDDs through symbolic kernels. In *ASPDAC, Proceedings of the Asia South Pacific Design Automation Conference*, pages 276–279, January 2005.

References

[AGL+95] 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.

[AJS99] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Formal verification using parametric representations of Boolean constraints. In *DAC, Proceedings of Design Automation Conference*, pages 402–407, June 1999.

[Ash57] 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.

[BBB+87] 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.

[BBK89] 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.

[BBS90] Derek L. Beatty, Randall E. Bryant, and Carl-Johann H. Seger. Synchronous circuit verification by symbolic simulation: An illustration. In *Proceedings of Sixth MIT Conference on Advanced Research in VLSI*, pages 98–112, 1990.

[BC01] Randal E. Bryant and Yirng-An Chen. Verification of arithmetic circuits using binary moment diagrams. *International Journal on Software Tools for Technology Transfer*, 3(2):137–155, 2001.

[BCL+94] Jerry R. Burch, Edward M. Clarke, David E. Long, Ken L. McMillan, 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.

[BCRR87] 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.

[BD96a] 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.

[BD96b] Valeria Bertacco and Maurizio Damiani. Boolean function representation using parallel-access diagrams. In *Proceedings of the Sixth Great Lakes Symposium on VLSI*, March 1996.

[BD97] 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.

[BDQ99] 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.

[Ber03a] Janick Bergeron. *Writing Testbenches: Functional Verification of HDL Models*. Kluwer Academic Publishers, 2nd edition, 2003.

[Ber03b] Valeria Bertacco. *Achieving Scalable Hardware Verification with Symbolic Simulation*. PhD thesis, Stanford University, 2003.

[BF89] Soumitra Bose and Allan Fisher. Verifying pipelined hardware using symbolic logic simulation. In *ICCD, Proceedings of the International Conference on Computer Design*, pages 217–221, October 1989.

[BL92] 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.

[BLW95] 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.

[BM82] 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.

[BO02] Valeria Bertacco and Kunle Olukotun. Efficient state representation for symbolic simulation. In *DAC, Proceedings of Design Automation Conference*, June 2002.

[BRB90] 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.

[BRSVW87] 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.

[Bry85] Randall E. Bryant. Symbolic verification of MOS circuits. In *Proceedings of 1985 Chapel Hill Conference on VLSI*, pages 419–438, May 1985.

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

[Bry92] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary–decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[CBM89] 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.

[CCLQ97] 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.

[CCQ96] 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.

[CIJ+95] 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.

[CJB79] William C. Carter, William H. Joyner, and Daniel Brand. Symbolic simulation for correct machine design. In *DAC, Proceedings of Design Automation Conference*, pages 280–286, June 1979.

[CM90] Olivier Coudert and Jean Christophe Madre. A unified framework for the formal verification of sequential circuits. In *ICCAD, Proceedings of the International Conference on Computer Aided Design*, pages 126–129, November 1990.

[CM92] 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.

[CMA02] 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.

[CUD99] CUDD-2.3.1. *http://vlsi.Colorado.edu/fabio*, 1999.

[Cur62] Herbert A. Curtis. *A New Approach to the Design of Switching Circuits*. Van Nostrand, Princeton, N.J., 1962.

[DeV97] 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, November 1997.

[FFK88] 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.

[FMY97] Masahiro Fujita, Patrick McGeer, and Jerry Yang. Multi-terminal binary decision diagrams: An efficient datastructure for matrix representation. *Formal Methods in System Design*, 10(2-3):149–169, 1997.

[GB03] Amit Goel and Randal E. Bryant. Set manipulation with Boolean functional vectors for symbolic reachability analysis. In *DATE, Design, Automation and Test in Europe Conference*, pages 10816–10821, March 2003.

[Han88] Craig Hansen. Hardware logic simulation by compilation. In *DAC, Proceedings of Design Automation Conference*, pages 712–716, June 1988.

[Hau95] Scott Hauck. *Multi-FPGA Systems*. PhD thesis, University of Washington, Dept. of Computer Science and Engineering, 1995.

[HD93] 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.

[HKM01] Faisal I. Haque, Khizar A. Khan, and Jonathan Michelson. *The Art of Verification with Vera*. Verification Central, 2001.

[HMN01] 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.

[HSH+00] 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.

[Hue02] 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.

[JG92] Prabhat Jain and Ganesh Gopalakrishnan. Some techniques for efficient symbolic simulation-based verification. In *ICCD, Proceedings of the International Conference on Computer Design*, pages 598–602, October 1992.

[JG93] 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.

[JG94] 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.

[Joh01] 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.

[Jon99] Robert Brent Jones. *Applications of Symbolic Simulation to the Formal Verification of Microprocessors*. PhD thesis, Stanford University, August 1999.

[Jon02] Robert B. Jones. *Symbolic Simulation Methods for Industrial Formal Verification*. Kluwer Academic Publishers, 2002.

[Kar63] Richard M. Karp. Functional decomposition and switching circuit design. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):291–335, 1963.

[Kar88] Kevin Karplus. Representing Boolean functions with if-then-else dags. Technical Report UCSC-CRL-88-28, Baskin Center for Computer Engineering & Information Sciences, 1988.

[Kar89] Kevin Karplus. Using if-then-else dags for multi-level logic minimization. In *Proceedings of Advanced Research in VLSI*, pages 101–118, 1989.

[Kar90] 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.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[KKD01] Alferd Kolbl, James Kukula, and Robert Damiano. Symbolic RTL simulation. In *DAC, Proceedings of Design Automation Conference*, pages 47–52, June 2001.

[KN96] 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.

[KOW+01] 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.

[LMUZ02] 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.

[MD93] 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.

[Min93] S.-I. Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *DAC, Proceedings of Design Automation Conference*, pages 272–277, June 1993.

[MKK+02] In-Ho Moon, Hee Hwan Kwak, James Kukula, Thomas Shiple, and Carl Pixley. Simplifying circuits for formal verification using parametric representation. In *FMCAD, Proceedings of International Conference on Formal Methods in Computer-Aided Design*, pages 52–69. Springer-Verlag, 2002.

[MKRS00] 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.

[MNS+90] 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.

[MSBSV93] 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.

[MWBSV88] 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.

[PB05a] Stephen Plaza and Valeria Bertacco. Boolean operations on decomposed functions. In *International Workshop on Logic Synthesis*, pages 310–317, June 2005.

[PB05b] Stephen Plaza and Valeria Bertacco. STACCATO: Disjoint support decompositions from BDDs through symbolic kernels. In *ASPDAC, Proceedings of the Asia South Pacific Design Automation Conference*, pages 276–279, January 2005.

[Pfi82] Gregory F. Pfister. The yorktown simulation engine: Introduction. In *DAC, Proceedings of Design Automation Conference*, pages 51–54, January 1982.

[REH99] Gerd Ritter, Hans Eveking, and Holger Hinrichsen. Formal verification of designs with complex control by symbolic simulation. In *CHARME, Proceedings of Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 234–249. Springer-Verlag, 1999.

[RS95] 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.

[Rud93] 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.

[Sas99] Tsutomu Sasao. Totally undecomposable functions: Applications to efficient multiple-valued decompositions. In *ISMVL*, pages 59–65, 1999.

[SB95] Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6(2):147–189, 1995.

[Sha49] Claude E. Shannon. The synthesis of two-terminal switching circuits. *Bell Systems Technical Journal*, 28(1):59–98, 1949.

[Sin53] Theodore Singer. The decomposition chart as a theoretical aid. Technical Report BL-4, Sec.III, Harvard Computational Laboratory, 1953.

[SM98] Tsutomu Sasao and Munehiro Matsuura. DECOMPOS: An integrated system for functional decomposition. In *International Workshop on Logic Synthesis*, pages 471–477, 1998.

[SMW71] 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.

[TSL+90] 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.

[VBJ97] Miroslav N. Velev, Randal E. Bryant, and Alok Jain. Efficient modeling of memory arrays in symbolic simulation. In *CAV, Proceedings of International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 388–399. Springer-Verlag, June 1997.

[vEJ96] 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.

[WCZK01] Dong Wang, Edmund Clarke, Yunshan Zhu, and Jim Kukula. Using cutwidth to improve symbolic simulation and Boolean satisfiability. In *HLDVT, IEEE International High Level Design Validation and Test Workshop*, pages 165–170, November 2001.

[WD00] 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.

[WDB00] Chris Wilson, David L. Dill, and Randal E. Bryant. Symbolic simulation with approximate values. In *FMCAD, Proceedings of International Conference on Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 470–485. Springer, November 2000.

[WHPZ87] 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.

[Wil01] James Christofer Wilson. *Symbolic Simulation Using Automatic Abstraction of Internal Node Values*. PhD thesis, Stanford University, December 2001.

[Yan91] Saeyang Yang. Logic synthesis and optimization benchmarks user guide, version 3.0. Technical report, Microelectronics Center of North Carolina, January 1991.

[YS03] Jin Yang and Carl-Johan H. Seger. Introduction to generalized symbolic trajectory evaluation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(3):345–353, 2003.

[YSP+99] 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.

Index

About the Author

Valeria Bertacco is an Assistant Professor of Electrical Engineering and Computer Science (EECS) at the University of Michigan. Her research interests are in the areas of computer-aided design with emphasis on formal verification, logic simulation, system analysis and core CAD algorithms. Valeria joined the faculty at Michigan after being at Synopsys for 4 years as a Staff Research Engineer in the Verification Group and in the Advanced Technology Group, where she was part of the core team that created Magellan, the software solution proposed by Synopsys for the formal verification of digital designs. Prior to Synopsys, she was with Systems Science Inc., a Palo Alto startup which developed Vera, a testbench development language for verification, later acquired by Synopsys.

During her transition from Synopsys to the University of Michigan, she received her Ph.D. degree in Electrical Engineering from Stanford University in 2003. She also holds an M.S. in Electrical Engineering from Stanford and a Laurea degree summa cum laude in Computer Engineering from the University of Padova in Italy.

Currently, her research group is exploring a family of "low-maintenance" verification techniques that improve the scalability of verification without sacrificing quality, and reduce the engineering effort involved in verification with solutions that fit within current methodologies. On the reliability front, she is developing efficient simulation-based solutions to evaluate the reliability of complex digital designs. She is the author of the publicly available Staccato package, an efficient algorithm to expose all the disjoint support decompositions of a given Boolean function, which finds application both in the domain of verification and synthesis. Valeria also led the effort for the development of the verification section in the 2004 and 2005 ITRS reports (International Technology Roadmap for Semiconductors).