The Dissertation Committee for Jun Yuan
Certifies that this is the approved version of the following dissertation:

Symbolic Methods in Simulation-based Verification

Committee:

Adnan Aziz, Supervisor

Jacob Abraham

Vijay Garg

Carl Pixley

Gustavo de Veciana

Martin D. F. Wong

Symbolic Methods in Simulation-based Verification

by

Jun Yuan, B.S., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2002

To Katy

Acknowledgments

It is hardly believable that I have finally come to the conclusion of this seemly endless pursuit of a PhD. But here I am, writing the last few pages of my dissertation; I'd like to thank the people that have made this possible.

It is my extreme pleasure to thank my advisor, Adnan Aziz. Adnan has been a great mentor and friend during my extended stay at UT Austin. Besides his courses on verification and synthesis, which become the topic of this dissertation, Adnan also convinced me to buy my first LaTeX book that greatly improved the quality of my writing. His steadfast devotion to uncompromising standards in demonstrating scientific ideas has always been a source of inspiration in my academic life.

I am very happy indeed to thank Carl Pixley for being a highly interactive and supportive supervisor in my job at Motorola. We managed to find a great match between my research at UT and my work in the formal verification group that Carl developed. Carl's drive to mathematic rigor and insightful conversations have made this dissertation a less difficult task.

I am also deeply honored to have Jacob Abraham, Vijay Garg, Gustavo de Veciana, and Martin Wong serving on my committee. I thank them all for giving me invaluable advice during my qualification process and afterwards.

I would like to take this opportunity to thank my former advisors Daniel

Berleant (U. of Arkansas) and Stephen Szygenda. Dan's precise style in teaching C programming gave me a sound understanding of software and computer architectures. Steve introduced me to the world of Electronic Design Automation. For the past eight years, I have made a living by writing EDA tools in C.

Melanie Gulick, our department's graduate coordinator, deserves a special thank for her resourceful help in my constant, heroic fight to "come back" after being deleted from the registration, for having to skip a semester or two once in a while to keep my day job.

I would also like to thank my current and former colleagues. Jian Shen (UT Austin) collaborated on my first paper, his microwaved potatoes and beef stews tasted so delicious those sleepless nights before paper deadlines. Matt Kaufmann (Motorola) talked me into switching from *vi* to *emacs*: as he put it "Emacs is all you need for the job." Nearly so, considering the quantum leap in productivity as a result. I can't even imagine how many times I would have to "i" and "Esc" to type up this document. I thank Kurt Shultz for the initial implementation of SymGen, and helpful discussions in the course of the tool's development.

I heartily thank all my soccer buddies, for bringing together a brotherhood in the games and in the parties thereafter. I enjoyed the passion and artistry in motion on the field, as well as the free-spirited chats in happy hours at the Double Dave. Soccerizing has kept me up both physically and emotionally.

I thank my parents for always being supportive and encouraging in my pursuit of academic excellence, ever since I was very young. I thank my wife for being

patient with my frequent late returns to home, and for her down-to-earth Sichuan cuisine, particularly the real “doubly cooked pork” that is extremely rare on the menus of local Chinese restaurants.

Finally, I owe a most profound debt and thank you to my best friend, Katy, who has shaped my life more than anyone else. I would not have come to America for graduate studies, and I could have given up on my doctorate long ago, without her intellectual inspiration and unwavering support through both good times and bad.

Symbolic Methods in Simulation-based Verification

Publication No. _____

Jun Yuan, Ph.D.
The University of Texas at Austin, 2002

Supervisor: Adnan Aziz

This dissertation conducts research in automating the design of digital hardware. Specifically we apply symbolic methods in simulation-based functional verification. Simulation, due to its simplicity and close coupling with the electronic design process, has been the prevalent approach to checking the correctness of designs. However, it suffers from several drawbacks. First, simulation verifies only the portion of design behavior that is exercised by input vectors; in addition, input vector generation itself is a time-consuming and error-prone process. Both problems are aggravated by the exponential growth in integrated circuit design complexity.

On the other side, formal verification is "vector-less" in that it certifies correctness either through mathematically rigorous proofs, or by exhaustive enumeration of design behaviors. Needless to mention, this approach requires either enormous computation resources or a great deal of manual intervention to verify large

designs. The problem, however, is greatly alleviated by the advent of symbolic methods, particularly the introduction of Binary Decision Diagrams to represent sets of state and transition dynamics. Symbolic formal verification has since been adopted in practice, but still limited to simple protocols and small designs.

It is natural to explore ways to leverage symbolic methods in simulation verification. To this end, we introduce several such applications. We first describe what we referred to as "saturated simulation" and "retrograde analysis" in checking invariant properties that are common to electronics designs. State and transition coverage are used as the guidance for a partial symbolic simulation. Consequently, a higher level of verification confidence is achieved.

We then present a symbolic input vector generation method, in which state-dependent constraints and input biases are used to confine the generated vectors to "legal" and "interesting" cases. The constraints and biases are both of a dynamic nature, that is, they can depend upon the state of the design. This enables generation of complicated sequences of vectors.

We also discuss methods of optimizing the vector generation process through efficient extraction of a special kind of constraints, in which the inputs are fully specified by the state of the design. In the end, we present an alternative vector generation method based on constraint synthesis. Beyond its obvious role in simulation, the method also provides a constraint based interface model for other verification approaches, such as model checking and emulation.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Integrated circuits have been growing rapidly in scale and functionality. Driving this growth is the relentless advance of technologies in integrated circuit fabrication, which, as sagely predicted by Moore's law, has been doubling the transistor count per chip every eighteen months for the past three decades. Today, microprocessor designs are implemented using hundreds of millions of transistors; they involve design technologies ranging from performance boosters such as deep pipelining and parallelism, to integration endeavors highlighted by the System-on-Chip movement. On the other hand, time-to-market has been steadily shortening thanks to the ferocious competition in the industry. This increase in complexity, joined by the lack of verification time, has greatly magnified the likelihood of "buggy" products being put into customers' hands. Well-known examples of such include Intel's Pentium^{TM} floating point division bug [12], and Toshiba's disk drive problem [31]. Both cost the companies hundreds of millions in dollars, and more in the damage to consumers' confidence in their products.

In short, there is a clear and present crisis of verification, a crisis that has changed the balance between design and verification, making the latter the bottleneck in product development cycle. This has prompted research in various di-

rections, targeting verifications at all implementation levels, including functional verifications from behavior level down to transistor level, timing verifications at the gate and transistor levels, layout versus schematic (LVS) comparison for the correctness of lithographic masks, and the testing of manufacture faults on Silicon. One way or another, all forms of verification play an indispensable role in insuring a functioning product.

The focus of this dissertation will be on functional verification, referring to the checking of whether the design conforms to a set of specifications, regardless of power, timing, area, testability, manufacturability, and so forth. All of the latter are very important considerations and deserve the attention they get. But functional verification, in its own right, might easily be the most challenging problem in design practices [93]. This challenge stems from the drive for completeness in an ideal functional verification. The exploration of the complete design functionality, under the circumstance of a case-by-case testing, translates indirectly into having to examine the design on application of all possible execution sequences, with all combinations of the design inputs. This is a formidable task even for designs of modest size, say a 256-bit RAM (Random Access Memory), which nevertheless yields 10^{80} possible combinations of its contents. Since case-based testing alone is inadequate, functional verification sometimes resorts to complementary methods of nonenumerative nature. Consequently, functional verification approaches can be broadly categorized as being in the domain of case-based testing, which is commonly referred to as *simulation*, or the domain of analytical, i.e. *formal*, methods. The latter is often assisted by nonenumerative techniques, such as the *symbolic*

methods.

The rest of this chapter provides background information on simulation-based and formal approaches to functional verification. It motivates the synergy of the two approaches, and the application of symbolic methods to simulation in particular, in an effort to improve the efficiency and applicability of simulation-based functional verification. This chapter also outlines the contents, and discusses in some detail the main contributions of this dissertation.

1.1 Simulation

Simulation is the most common approach to functional verification. The verification is done by checking that the design has the proper behavior as elicited by a series of functional vectors [100, 105]. More specifically, the verification consists of three major tasks: generating the functional test vectors, executing the vectors on the design, and monitoring if the design satisfies its specification during the execution. Simulation-based verification is straightforward, and scales well in the sense that the amount of CPU time and memory taken to simulate is proportional to the size of the design. However, the drawbacks of this approach are also apparent: (1) An exhaustive simulation is impossible for most designs of interests; therefore, a degree of confidence can only be obtained by simulating with a large number of vectors. (2) The generation of simulation vectors for large designs can be tedious and time-consuming.

The situation is aggravated by the lack of good metrics that gauge the quality of simulation. Roughly speaking, simulation quality metrics fall into two cate-

gories: code coverage and function coverage. Code coverage, such as the coverage on lines and branches in a design description, does allow a sense of completeness since the description must be finite. The completeness, however, does not extend to the coverage of functionalities. In fact, code coverage is purely empirical and quite inadequate in regard to measuring how much of the design behavior is exercised. A better measurement can be achieved by the function coverage, for example, the coverage on state or state transitions. Theoretically, any coverage defined over the structure or functionality of a digital design is finite, including coverage on state and transitions. In practice, however, full state or transition coverage is usually unattainable because of the vast state space, or undetectable, due to the possibility that some states are never reachable and we are not aware of this *a priori*.

Therefore, simulation-based verification is left with no feasible stopping criterion and deemed incomplete. A typical industry practice is to simulate the design with a relatively small set of vectors created manually by the engineers, then with random vectors for as long as is feasible.

1.2 Formal Verification

In contrast to simulation, formal verification methods, such as theorem proving [19, 55, 67, 87] model checking [20, 25, 43, 82, 91], and language containment [8, 71–73, 107], offer complete verification, but require either extensive human expertise or enormous computational resource.

In theorem proving, the design and the properties are expressed as formulas in some mathematical logic. A property is proved if it can be derived from the

design in a logic system of axioms and a complete set of inference rules. The proof, resembling a combinatorial search, may appeal to intermediate definitions and lemmas in addition to the axioms and rules. Theorem proving typically requires a great deal of skilled human guidance and is therefore limited to applications of higher abstraction levels, such as the communication protocols.

Model checking is a technique that relies on building a finite state model and checking that the desired properties hold in that model. A property, represented by a formula in temporal and modal logics [6, 30, 41, 43], corresponds to a set of states that can be algorithmically obtained by fixpoint analysis of the formula on the finite state model [44]. The set of states reachable from a designated set of initial states can also be obtained via a fixpoint analysis, commonly known as the reachability analysis. The validity of the formula is then established by the inclusion of the reachable states of the design in the set corresponding to the formula. Model checking is essentially an exhaustive state space search that is guaranteed to terminate since the state space is finite.

Language containment in spirit is similar to model checking. In language containment based verification, the property is represented by a deterministic finite state automata, and the design by a nondeterministic finite state automata. Verification is cast in terms of whether the formal language of the former automata contains that of the latter.

Unlike theorem proving, both model checking and language containment are fully automatic. However, the two share a major disadvantage — the *state explosion* problem. In the worst case, exhaustive state exploration requires a memory

usage that is exponential to the number storage elements in the design.

1.3 Symbolic Methods in Formal Verification

As we have seen, the applicability of formal verification is hindered by the problems of usability and state explosion. Recently, the advent of symbolic methods, particularly the Binary Decision Diagram (BDD) [21], has great alleviated the problems. BDD is a data structure that can be used to store and manipulate Boolean functions implicitly. In late 1980s and early 1990s, several researchers independently realized that BDD can be use to extend the scope of formal verification [18, 32, 82, 91]. A BDD-based model checker executes the model checking algorithm symbolically, by representing sets of state as characteristic functions. This avoids the explicit enumeration of the potentially exponential state space. An initial effort in applying BDD to model checking achieved the verification of designs with up to 10^{120} states [25]. As a comparison, a model checker using explicit state enumeration [62] usually handles state graphs with a capacity in the order of 10^5 states. Symbolic simulation, a scaled-down model checking that handles a limited set of temporal properties, has shown to be effective in memory verification [23, 54] and validity check of microprocessor's instruction set architectures [64, 108]. Theorem proving can also use symbolic evaluation as a decision procedure so that lower level proof goals can be checked automatically [1, 94]. A symbolic language containment prover has also shown improvement over its nonsymbolic counterpart [107].

Despite its improvement in usability and capacity, formal verification remains inadequate in the presence of practical designs. For example, symbolic

model checking is usually expected to handle designs with up to a few hundred storage elements; on more complex designs, the graph representation of states can grow extremely large, resulting in space-outs or severe performance degradations due to paging. As of today, formal verification is still limited to protocols, simple algorithms, and sub-blocks of designs.

1.4 Scope and Overview of the Dissertation

This dissertation addresses the major issues in simulation verification by applying formal and symbolic methods. On the one hand, simulation scales well for large designs but suffers from low coverage and the vector generation problem, due to its enumerative nature; on the other hand, formal verification is inherently complete but expensive, while symbolic methods help improve the capacity but not enough for practical designs. This contrast leads to our approach of combining the advantages of formal and symbolic methods and applying them towards the improvement of simulation.

In the first part of the dissertation, we attack the coverage problem by enhancing bug-finding ability in symbolic reachability analysis. Since neither simulation nor formal methods can verify a large design completely, that is, certificate the absence of design bugs, we may just evaluate a verification tool by its ability of catching bugs. This concept is also known as “falsification”, in contrast to verification. We propose two cases, the *saturated simulation* based on reachability analysis from model checking, and *retrograde analysis* based on a strategy employed in two-player games. Both are applied in checking invariant properties.

Our synergistic approach culminated, as covered in the second part of this dissertation, in symbolically modeling design interfaces, thus automating the simulation vector generation process. This problem is systematically studied beginning with the construction of efficient algorithms that generate vectors according to environment constraints and input biases. We then discuss related issues including constraint debugging and simplification of constraint solving. This constrained vector generation method is implemented in a tool called *SymGen* developed at Motorola, which has been used in the verification of numerous commercial designs. For completeness, we also present an alternative vector generation method based on constraint synthesis, which finds applications in model checking and hardware-accelerated simulation.

For reference, the applications on improving invariant checking are published in [7, 116], and the others in [112, 115, 117–120]. The rest of this section outlines each of these applications and their contributions.

1.4.1 Saturated Simulation

Many hardware designs can be partitioned into datapath and control portions. Saturated simulation exploits this distinction in a partial reachability analysis that focuses on the control behavior of the design. The designer designates as control variables a subset of the storage elements, for example, the program counter and status bits in a micro-processor. At each step of the analysis, symbolic abstraction techniques are used in the traversal of the full set of control states, as well as all the control transitions, from the current set. Heuristically, the control portion of the

design, while being much smaller than the datapath, is the main source of design errors. Roughly speaking, saturated simulation attempts to explore as much of the control behavior as possible, thus increasing the likelihood of finding bugs.

The efficiency of this approach comes from the observation that it is feasible to compute the symbolic image of a single state even for very large designs, coupled with the fact that the set of control states is typically much smaller than the entire state space. Additionally, fast BDD routines exist for generating and manipulating representative elements of equivalence classes [75].

1.4.2 Retrograde Analysis

Retrograde analysis is a technique used to create endgame databases for several two-player games. The main idea is to pre-compute a set of positions that eventually lead to a winning position. The set can then be used as the target of a winning strategy search, with two advantages: (1) The target is enlarged, and (2) The search distance is shortened. Following this line of thinking, we enhance retrograde analysis by enlarging the set of initial states, and heuristically select the starting states so that the overall search distance is further reduced. This idea is applied to checking invariant properties.

1.4.3 Constrained Simulation Vector Generation

In saturated simulation and retrograde analysis, symbolic methods are applied to the execution of simulations. Here we discuss their application to another aspect of simulation — vector generation.

Vector generation is tedious and time-consuming. In the meantime, simulation with a large amount of vectors is the only way to obtain a high degree of verification confidence. In dealing with the dilemma, we use BDDs to symbolically generate simulation vectors. The first requirement for any vector generation tool is to define, and generate within, the legal input space, which often is a function of the current state of the design. In addition, a robust tool should also provide the ability to influence the distribution of the generation, thus directing the simulation towards the "corner" cases, or other test scenarios deemed interesting. We strive to meet these important requirements, in a unified fashion, using Boolean constraints and input biases in an efficient algorithm. The algorithm is immune to the *backtracking* problem that is typical to many vector generation tools, as well as independent of variable orderings, despite of the use of BDDs.

The constraints used in vector generation also provide a formal description of the design interface. They can be treated as the assumption for the environment in a formal verification of the design, or the proof obligation in the verification of the parent module of the design. Thus, constrained simulation generation fits seamlessly in an assume/guarantee style of hierarchical verification framework [66, 92].

1.4.4 Constraint Diagnosis

Any constraint based programming will have to deal with the diagnosis of conflicting constraints, which has been well studied in many research disciplines, for example, in artificial intelligence [36]. Our focus of constraint diagnosis is on a

problem unique to the simulation vector generation method, that is, constraint conflicts conditioned upon the state of the design. We present methods of identifying and eliminating these conflicts.

1.4.5 Simplification of Constraint Solving

Complexity of constraints can grow arbitrarily due to their combinatorial nature. Therefore, even implicit representations, in our case, BDDs, will have to face the efficiency problem. An immediate step toward dealing with the problem is the disjoint-support partitioning of the constraints [119]. This partition can be further refined due to the fact that the variables in hardware constraints are not homogeneous: all state variables are bounded by the design while some of them may fully specify some input variables which can be exploited in simplifying the constraints. To this end, we present efficient methods of extracting and utilizing a special kind of constraints, called the *hold-constraints*. The extraction not only simplifies the involved constraints, but also decomposes partitions which is otherwise impossible.

1.4.6 Constraint Synthesis

It is often of interest to implement constraint based vector generation in hardware, especially when the verification environment allows no proprietary constraint solving capability, for example, in hardware accelerated simulation, or emulation. We present a synthesis algorithm which computes a set of Boolean functions whose range is exactly the input space defined by a constraint. Although resembling an equation solving technique, called Boolean Unification, our method takes into

the consideration of the heterogeneous nature, again, of the variables in hardware design constraints. This leads to a fundamental observation that synthesis of such constraints enjoys an optimization space represented by states for which there are no satisfying inputs.

1.5 Chapter Structures

The dissertation proper begins, in Chapter 2, with definition of terminology and preliminaries that will be used throughout this document. Each of the ensuing chapters covers one of the applications outlined in the preceding section. For better encapsulation, introductory information including preliminaries and related works local to a specific application is explained in the corresponding chapter. The final chapter summarizes the dissertation and gives directions for future work. The appendices contain results that are relevant to our topics but do not fit in the main flow of our presentation.

Chapter 2

Preliminaries

In this chapter, we define various notations and terminologies. We first give a quick review of Boolean algebra and functions, adopted from [83], and show how digital hardware designs are modeled in them. Then we describe how reachability analysis, the basis for many formal verification methods, is done in the Boolean representation of designs. We also touch upon the topic of model checking, the formal verification method that is most relevant to our work on invariant checking. Finally, we give a detailed exposition of the symbolic data structure Binary Decision Diagrams, that is used throughout this dissertation.

2.1 Boolean Algebra and Notations

A *Boolean Algebra* is defined by the set $B = \{0, 1\}$ and two Boolean operations, denoted by $+$, the *disjunction*, and $\cdot$, the *conjunction*, which satisfy the *commutative* and *distributive* laws, and whose identity elements are 0 and 1, respectively. In addition, any element $a \in B$ has a complement, denoted by $\overline{a}$, such that $a + \overline{a} = 1$ and $a \cdot \overline{a} = 0$.

A Boolean variable is one whose range is the set B. A *literal* is an instance of a Boolean variable or of its complement. The multi-dimensional space spanned

by n Boolean variables is the n-nary Cartesian product $B \times \ldots \times B$, denoted by B^n. A point in this Boolean space is represented by a *vector* of dimension n. When Boolean variables are associated with the dimensions, a point can be identified by the values of the corresponding variables, i.e., by a *minterm*, the product of n literals. Often, products of literals are called *cubes* since they can be graphically represented as hypercubes in the Boolean space.

A Boolean formula is a composition of Boolean variables, the constants 0 and 1, and Boolean operations, that obeys the following rules:

- 0, 1, and Boolean variables are Boolean formulas
- if f is a Boolean formula then so is $\overline{f}$
- if f and g are Boolean formulas then so are $f + g$ and $f \cdot g$

Many other Boolean operations can be introduced to abbreviate formulas generated above; for example, $f \oplus g$ where $\oplus$ denotes the *exclusive-OR*, denoted by $\oplus$, is a shorthand for $f \cdot \overline{g} + \overline{f} \cdot g$.

Since all Boolean formulas evaluate to either 0 or 1, they define mappings from B^n to B, where n is the number of variables in the formula. Such mappings are called the Boolean functions. In this dissertation, we shall use the terms Boolean formula, and Boolean function or simply *function* interchangeably, as deemed appropriate by the context.

Given a function $f : B^n \mapsto B$, the set of minterms $\{\alpha \in B^n \mid f(\alpha) = 1\}$ is called the *onset* of f, denoted by f^{on}, and the set $\{\alpha \in B^n \mid f(\alpha) = 0\}$ is called

the *offset* of f, denoted by f^{off}. Conventionally, a function f, when treated as a set, represents f^{on}. For this reason, f is called the *characteristic function* of the set f^{on}. Conversely, a set S, when treated as a function, represents the characteristic function of S.

The *cofactor* of $f(x_1, \ldots, x_i, \ldots, x_n)$ with respect to x_i is $f(x_1, \ldots, 1, \ldots, x_n)$, denoted by $f\mid_{x_i=1}$ or simply f_{x_i}; similarly, the *cofactor* of $f(x_1, \ldots, x_i, \ldots, x_n)$ with respect to $\overline{x_i}$ is $f(x_1, \ldots, 0, \ldots, x_n)$, denoted by $f\mid_{x_i=0}$ or simply $f_{\overline{x_i}}$. The notation f_c where c is a cube represents the successive application of cofactoring of f with respect to the literals in c.

The *existential quantification* of f with respect to a variable x is $f_x + f_{\overline{x}}$, denoted by $\exists_x f$; the *universal quantification* of f with respect to x is $f_x \cdot f_{\overline{x}}$, denoted by $\forall_x f$; the *Boolean differential* of f with respect to x is $f_x \cdot \overline{f_{\overline{x}}} + \overline{f_x} \cdot f_{\overline{x}}$, abbreviated as $f_x \oplus f_{\overline{x}}$ or simply $\partial f/\partial x$.

2.2 Hardware Modeling

Digital hardware circuits operate on binary numbers 0 and 1, and can be analytically reasoned in their mathematical models based on Boolean algebra. The modeling can be done at the *structural* level using *netlists*, or at the *behavioral* level using *state transition graphs*; Singhal [103] gives a detailed exposition for hardware modeling.

A netlist consists of an interconnected set of primary inputs, outputs, gates, and latches. Each gate performs a Boolean operation on signal values of its inputs,

and produces the result at its output. A special primary input, called a *clock*, governs the latches in a manner such that a latch either maintains the value of its output, or updates the value to that of its input if a designated *clock event* occurs, e.g., the clock changes from 0 to 1. Consequently, a latch is often referred to as a *storage* or *state holding* element, and identified by a state variable representing its output.

Let $X = \{x_1, \ldots, x_n\}$ be the set of variables representing the primary inputs, and $Y = \{y_1, \ldots, y_m\}$ the set of state variables, where y_i is for the i-th latch. A state is a valuation of Y, given by the minterm

$$\bigwedge_{i=1}^{m} (y_i == c_i)$$

where c_i is the valuation of y_i, $y_i == 1$ and $y_i == 0$ correspond to the literals y_i and $\overline{y_i}$, respectively.

The valuation of y_i is determined by the *transition function* $F_i(X, Y)$, which is the composition of gate operations transitively affecting the input of the i-th latch. Define $Y' = \{y'_1, \ldots, y'_m\}$ to be the set of *next state* variables, where y'_i represents the input of the i-th latch. The *transition relation* of y'_i is defined as

$$T_i(y'_i, X, Y) = (y'_i == F_i(X, Y)).$$

For designs where all latches are synchronized by the same clock thus transition at the same time, a global transition relation can be defined as

$$T(Y', X, Y) = \bigwedge_{i=1}^{m} T_i(y'_i, X, Y).$$

Separately, the outputs of the design are determined by a vectorial function $U(X, Y)$,

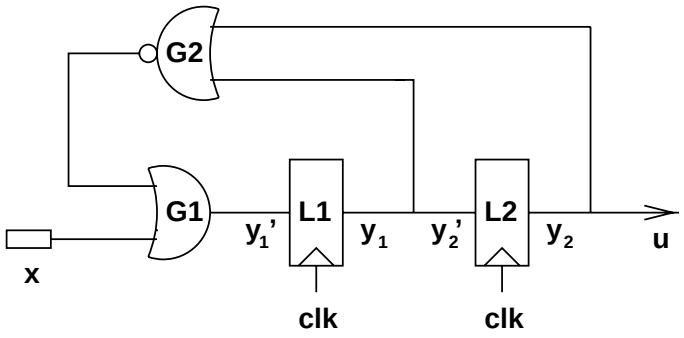


Figure 2.1: Example: Netlist

the composition of gate operations that drive the outputs.

The transition relation and output function can be visualized in the state transition graph (STG), which provides a behavioral description of the design. An STG is an edge-labeled directed graph $G = \{V, E, L\}$, where the vertices V correspond to the *states*, the edges E correspond to the state transitions, and the labels L represent the *input-output* pairs. More precisely, we have

$$V = B^{|Y|} = B^m$$

$$E = \{(s,t) \mid \exists x.(t,x,s) \in T$$

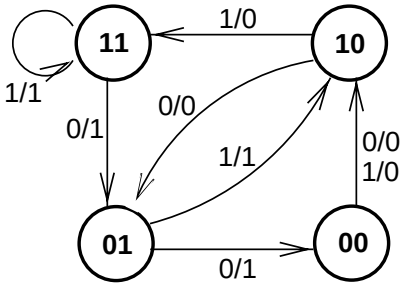
$$L = \{(s,t,x,u) \mid (t,x,s) \in T, u = U(x,s)$$

That is, if an edge $s \stackrel{(x,u)}{\rightarrow} t$ exists in the graph, then applying the input x at the state s results in an output of u, and a transition to the state t.

As an example, consider a design whose netlist is given in Figure 2.1. The design has two latches $L1$ and $L2$, an OR gate $G1$ and a NOR gate $G2$. The transition functions are $y_1' = x + \overline{y_1 + y_2}$ and $y_2' = y_1$, and the output function is $u = y_2$.

x	y_1	y_2	y_1'	y_2'
0	0	0	1	0
0	0	1	0	0
0	1	-	0	1
1	0	-	1	0
1	1	-	1	1

"-" means "0 or 1" .



"0/1" means the input is 1, the output is 0.

Figure 2.2: Example: Transition relation and state transition diagram

The corresponding transition relation and STG are shown in Figure 2.2.

2.3 Reachability Analysis

A common functional verification problem, known as *invariant checking*, is to decide whether some designated "bad" states are *reachable* from a set of initial states of the design. This kind of problems can often be solved by the *reachability analysis*, a method for state space exploration that is the basis of model checking.

Given the STG of a hardware design, a state t is said to be reachable from s if there is a sequence of transitions that starts from s and ends at t. The set of states reachable from s can be obtained by traversing the STG either depth-first (DF) or breadth-first (BF). In comparison, a DF traversal explores one successor state per step, thus requiring less memory, but longer run time in general, than a BF traversal, which explores all the successor states at each step. A natural compromise is to always start with a BF traversal if the computation resources allow, otherwise explore as many states as possible, per step. We therefore give more details only on

the BF-based analysis.

In the previous section, we showed that one can build the STG of a design from its transition relation. The construction is actually a one step BF state traversal, starting from all the states. Similarly, reachability analysis on an STG can be implicitly done using the transition relation, as in the following.

Let $T(Y', X, Y)$ be the transition relation, where X, Y and Y' are the input, state and next state variables, respectively. The set of states reachable in one step from the states in R is the *image* of R under T, or formally

$$Img(R(Y)) = \exists_{XY}[R(Y) \cdot T(Y', X, Y)]. \tag{2.1}$$

The series $R^0, R^1, \ldots$ where $R^{i+1} = Img(R^i) \vee R^i$ increases monotonically. Since the state space is finite, the series has a *least fixed point* (*lfp*) R^K where K is finite and for all $i \geq K$ $R^{i+1} = R^i$. Using the *lfp* operator (μ) from Mu-Calculus [42, 90], the computation of R^K can be characterized as

$$\mu Z.\,[Img(R^i) \vee Z]. \tag{2.2}$$

Figure 2.3 gives the corresponding algorithm with the optimization that *Img* is performed only on states that are "new" in R^i.

For a simply illustration of the algorithm, in the STG shown in Figure 2.2(b), the reachable states of the state 11 is computed in the series $\{\{11\}, \{11, 01\}, \{11, 01, 00\}, \{11, 01, 00, 10\}\}$.

Invariant checking can also be approached by the *backward reachability*

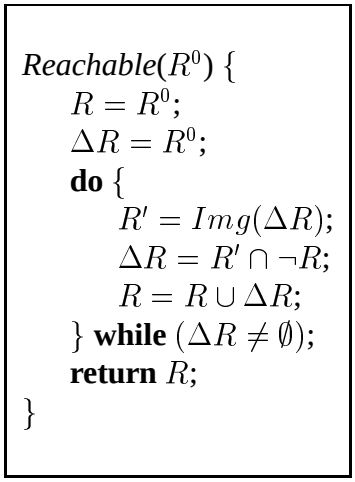


Figure 2.3: Reachability Analysis

analysis, which computes the set of states that reaches, instead of being reachable from, the bad states. The recursion is based on the following *pre-image* operator

$$PreImg(R(Y')) = \exists_{XY'}[R(Y') \cdot T(Y', X, Y)], \tag{2.3}$$

which also leads to a *lfp* computation

$$\mu Z.\,[PreImg(R^i) \vee Z]. \tag{2.4}$$

The algorithm implementing the above would be identical to that of the (forward) reachability analysis shown in Figure 2.3, except that Img would be replaced by $PreImg$.

2.4 Model Checking

Model checking (MC) is a problem of deciding whether a design satisfies, or *models*, a property expressed in some temporal logic [30, 43]. In this section, we give an overview of MC based on the Computational Tree Logic (CTL). Most of the definitions are taken from Emerson [41]. For other types of MCs, for example, the ones using CTL* or Linear Time Logic (LTL), we also refer the reader to [41].

Let P be the set of atomic propositions, which correspond to the state variables in hardware designs. Let M be the structure (S, s_0, R, L) representing the design where S correspond to the set of states, s_0 the initial state, R the state transition relation, and L the mapping from S to 2^P. Then the CTL-based MC is the problem of checking the validity of

$$M, s_0 \models p \tag{2.5}$$

where p is a CTL formula. Intuitively, the problem is to decide if p holds in the "computation tree" (the unfolding of R) of M rooted at s_0. CTL formulas can be thought of as being built up from Boolean combinations and nestings of atomic propositions, and temporal operators that include:

1. the path quantifiers A ("for all paths") and E ("for some path"), and
2. the state quantifiers G ("always"), F ("sometimes"), X ("nexttime"), and U ("until").

The syntax of CTL is given by two sets of rules, below, that inductively define the state (rules S1-3) and path (rule P1) formulas.

S1 Each atomic proposition in P is a state formula

S2 If p and q are state formulas then so are $p \wedge q$ and $\neg p$

S3 If p is a path formula then Ep is a state formula

P1 If p and q are state formulas then Xp, Gp, and $p\,U\,q$ are path formulas

The set of state formulas generated from the above rules forms the language of CTL. Note in this context, we use the logic connectives $\wedge$ (conjunction) and $\neg$ (negation), other connectives can be derived in the usual way. Also, the formulas are in the forms of only EX, EG, and EG since all other forms are abbreviations thereof: EFp stands for $E(true\,U\,p)$, AXp for $\neg EX\neg p$, AFp for $\neg EG\neg p$, AGp for $\neg EF\neg p$, and $A(p\,U\,q)$ for $\neg E[\neg q\,U(\neg p \wedge \neg q)] \wedge (\neg EG\neg q)$.

Let $x = (s_0, s_1, \ldots)$ denote a path, i.e., a sequence of states in the computation tree, and x^i denote the suffix $(s_i, s_{i+1}, \ldots)$. The semantics of $\models$, or the validity of Formula 2.5, is established inductively as follows:

S1 $M, s_0 \models a, a \in$ iff $a \in L(s_0)$

S2 $M, s_0 \models p \wedge q$ iff $M, s_0 \models p$ and $M, s_0 \models q$

$M, s_0 \models \neg p$ iff it is not the case that $M, s_0 \models p$

S3 $M, s_0 \models Ep$ iff $\exists x,\ M, x \models p$

P1 $M, x \models p$ iff $M, x \models p$

P2 $M, x \models p \wedge q$ iff $M, x \models p$ and $M, x \models q$

$M, x \models \neg p$ iff it is not the case that $M, x \models p$

P3 $M, x \models p\,U\,q$ iff $\exists i\ [M, x^i \models q$ and $\forall j(j < i \rightarrow M, x^j \models p)]$

$M, x \models Gp$ iff $\forall i\ [M, x^i \models p]$

$$M, x \models Xp \text{ iff } M, x^1 \models p$$

For illustration, we give two simple examples of CTL formulas and their meanings:

- AFp : p will be true eventually;
- $E(p\ U\ q)$: there is a path on which p holds until q is true.

We now describe the MC algorithms. First, we observe that MC can be converted to a membership testing:

$$M, s_0 \models p \text{ iff } s_0 \in Sat(p)$$

where $Sat(p)$ is the set of states that satisfies the CTL formula p. The remaining task is then to compute this set. Obviously, for $p \in P$, $Sat(p)$ is the set of states predicated by p, that is, $L^{-1}(p)$; for other CTL formulas, we consider only EXp, EGp, and $E(p\ U\ q)$, for the reason given previously. By observing the semantic rules, we have

$$\begin{aligned} Sat(EXp) &= PreImg(Sat(p)) \\ Sat(E(p\,U\,q) &= \mu Z.\,[Sat(q) \vee (Sat(p) \wedge EX(Z)] \\ Sat(EGp) &= \nu Z.\,[p \wedge EX(Z)] \end{aligned}$$

where $\mu Z.[f]$ and $\nu Z.[f]$ are notations in Mu-Calculus for the least and greatest fixpoints, respectively, of function f, and $PreImg(f)$, as we recall, computes the set of states which can reach in one step some state in f. As an example, the

implementation of $Sat(E(p\ U\ q))$ computation is given in Figure 2.4. Computing $Sat(EGp)$ is similar, and actually simpler.

We would like to stress, if it is not obvious enough, that the invariant checking problem discussed in Section 2.3 is an instance of MC. Specifically, checking the invariant q can be formulated as model checking the CTL formula $AG(q)$, thus, $\neg E(trueU\neg q)$. In computing $E(true\,U\,\neg q)$, the MC algorithm in Figure 2.4, letting $p = true$, matches the backward reachability analysis for $\neg q$ (Figure 2.3, replace Img by $PreImg$). Further, as we observe, the complement of the set of states that can reach $\neg q$ is equal to the set that can reach q. Therefore, MC of $AG(q)$ and invariant checking of q are identical problems.

As a side-node, the above equivalence will be more evident if MC is implemented with the Img operator, as it can, instead of with $PreImg$. The performance tradeoff between these two approaches has been an interesting topic in MC [63].

2.5 Reduced Ordered Binary Decision Diagrams

The Binary Decision Diagram (BDD) was first introduced by Lee [74] in 1959 and Akers [4] in 1978 as a compact representation for Boolean functions. In the mid-1980s, Bryant [21] proposed Reduced Ordered Binary Decision Diagrams (ROBDD) by imposing restrictions on BDDs such that the resulting representation is canonical. He also presented efficient ROBDD algorithms for Boolean operations. Since Bryant's work, there has been a blossoming of related research, mainly in the field of formal verification, but also in logic synthesis. In this chapter, we give an overview of BDDs and ROBDDs.

```
Sat_EU(p, q) {
    do {
        Z = q;
        ΔZ = q;
        Y = p ∧ PreImg(ΔZ);
        ΔZ = Y ∧ (¬Z);
        Z = Z ∨ ΔZ;
    } while (ΔZ ≠ ∅);
    return Z;
}
```

Figure 2.4: Computing $Sat(E(p\ U\ q))$

2.5.1 BDD Representation of Boolean Functions

A BDD is a rooted, directed acyclic graph wherein each internal node has two sub-BDDs and the terminal nodes are *ONE* and *ZERO*, representing the Boolean values 1 and 0, respectively. A BDD γ indentifies a Boolean function f of variables X, for a given mapping l from its nodes to X, as in the following.

1. Let r be the root of γ
2. If r is *ONE*, then $f = 1$
3. If r is *ZERO*, then $f = 0$
4. If r is an internal node, let $v = l(r)$ be the associated variable, g and h be the functions identified by r's *right* and *left* sub-BDDs, respectively, then $f = v \cdot g + \overline{v} \cdot h$

Because of the composition in Item 4 above, we sometimes use the adjectives *then* and *else*, in place of *right* and *left*, in distinguishing the two subgraphs of BDD node.

An ROBDD is an ordered and reduced BDD, as defined in the following:

1. An ordered BDD is a BDD whose variables follow a total ordering $\prec$ such that $l(s) \prec l(t)$ if t is a descendant of s.

2. An ROBDD is the maximally reduced version of an ordered BDD obtained by repeatedly applying to the latter the following rules until none is applicable.

   (a) If two subgraphs are identical, remove one of them, and let the dangling edge point to the remaining subgraph

   (b) If a node points to the same subgraph, remove the node, and let the dangling edge point to the subgraph

Figure 2.5 illustrates the derivation of an ROBDD from the "complete" BDD of function $a \cdot b + c$. The index $b2$ means the node is the second node labeled with variable b.

ROBDD is the symbolic data structure used throughout this dissertation. For brevity, we will refer to ROBDDs simply as BDDs hereafter.

(a) Merge identical subgraphs c1, c2 and c3

(b) Remove redundant nodes b1 and c4

(c) The reduced graph

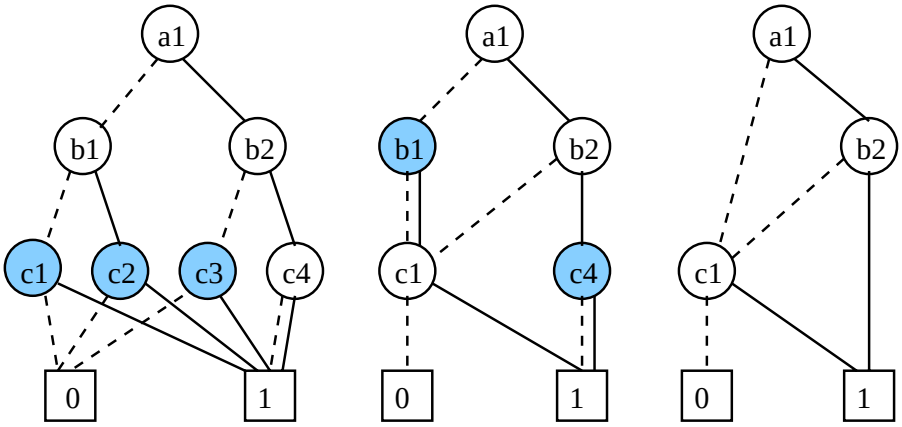


Figure 2.5: Reducing BDD for function $a \cdot b + c$

2.5.2 BDD Manipulations

All Boolean operations on BDDs can be implemented using one procedure called *Apply* [21], which is based on the Shannon expansion

$$g \circ h = v \cdot (g_v \circ h_v) + \overline{v} \cdot (g_{\overline{v}} \circ h_{\overline{v}}) \tag{2.6}$$

where $\circ$ is a Boolean operation and v a variable in the support of g or h. Let $Compose(v, l, r)$ be the BDD that is a composition of variable v and two BDDs l and r, such that the root is labeled with v and the left and right sub-BDDs are l and r, respectively. Let $Reduce$ be the reduction procedure given in the previous section. *Apply* recursively constructs a BDD from two operant BDDs and a binary operator,

```
Apply(g · h) {
    if (g == 0 || h == 0) return ZERO;
    if (g == 1) return h;
    if (h == 1) return g;
    Let v be the higher-ranked variable of variables
        labeling the roots of g and h;
    return Reduce( Compose(v, Apply(g_v · h_v), Apply(g_v̄ · h_v̄)));
}
```

Figure 2.6: The BDD Apply operation

based on the following recursion that is a direct translation of Equation 2.6.

$$Apply(g \circ h) = Reduce(Compose(v, Apply(g_v \circ h_v), Apply(g_{\overline{v}} \circ h_{\overline{v}})))$$

As an optimization, $Apply$ designates v as the higher-ranked variable of the variables labeling the roots of g and h. The terminal cases of the recursion depend upon the selection of $\circ$. For instance, Figure 2.6 gives the $Apply$ function for the BDD conjunction operation. Note the complement of f is computed as $Apply(f \oplus 1)$.

2.5.3 The BDD Size Consideration

For a given function, the size of its BDD representation, that is, the number of nodes in the BDD, varies with the variable ordering. To demonstrate this dependency, we use an example taken from [21]. Assume we are to construction a BDD for the function

$$f = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

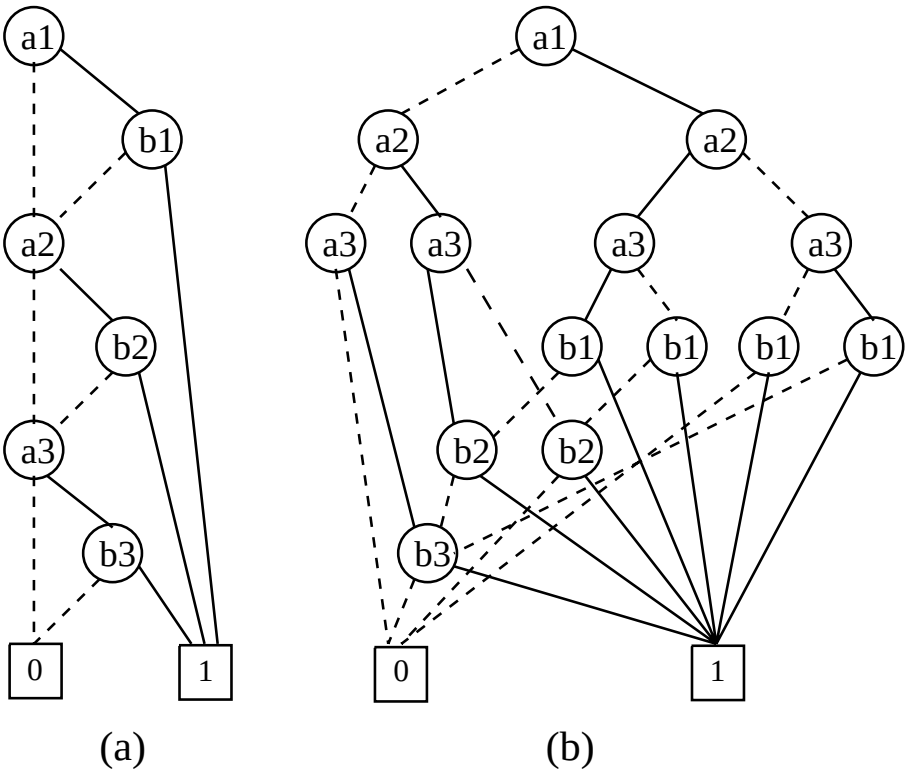


Figure 2.7: Dependency of BDD size on variable ordering

With the ordering $a_1 \prec b_1 \prec a_2 \prec b_2 \prec a_3 \prec b_3$, i.e., variables appear in the order $a_1 b_1 a_2 b_2 a_3 b_3$ on any path from the root to a terminal node, the BDD representation of f is shown in Figure 2.7(a). It has 8 nodes. In contrast, with $a_1 \prec a_2 \prec a_3 \prec b_1 \prec b_2 \prec b_3$, the BDD representation as shown in Figure 2.7(b) has 16 nodes. The first ordering is sometimes referred to as the *interleaved* ordering of the a and b variables. More generally, for the function $a_1 \cdot b_1 + \cdots + a_n \cdot b_n$, the interleaved ordering yields a BDD with $2(n+1)$ nodes, whereas the second ordering yields a BDD with 2^{n+1} nodes. Evidently, a poor choice of variable orderings can have very undesirable effects.

Thanks to pioneering works such as the static ([51]) and dynamic ([98]) variable orderings, a variety of techniques and heuristics [9, 17, 26, 39, 47, 49, 52, 88] have since been proposed and shown to be effective in finding good variable orderings for BDDs. In practice, good variable orderings exist for a large class of Boolean functions, and their BDDs are much more compact than traditional representations such as truth tables, conjunctive and disjunctive normal forms. For this reason, BDDs are widely used in design automation tools where compact and canonical representation of functions is in order.

Before we conclude this section, we need to mention another common problem with BDD sizing, that is, during a BDD operation, the intermediate BDDs can have a much larger size than the final BDD. This is especially noticeable when there are many BDD operants, for example, in the conjunction of a large group of BDDs. Techniques such as *conjunctive partitioning* [95] and *early quantification* [59] have been developed to handle this problem.

2.5.4 BDD-based Reachability Analysis and Model Checking

Reachability analysis, as discussed in Section 2.3, is the fixpoint computation of a series of monotonically increasing sets of states. The analysis, specifically, the *Img* and *PreImg* operations, can be performed symbolically using BDDs, since BDDs can represent and manipulate sets by treating the sets as their corresponding characteristic functions.

Chapter 3

Saturated Simulation

3.1 Introduction

Many hardware designs can be partitioned into *datapath* and *control* portions: the datapath portion manipulates and transports data among major components of the design, while the control portion configures how the data is steered and handled. For most such designs, the number of latches (memory elements) in the control portion is usually a small fraction of the total number of latches in the design. As an example, consider the `viper` microprocessor as shown in Figure 3.1. The data is temporarily stored in the register file, waiting to be loaded into the ALU (Arithmetic and Logic Unit) for computation. The control logic takes an instruction, and accordingly sets up the ALU and loads data. The ALU later informs the control about the status of the computation; the latter then updates the program counter to conclude the operation. In this picture, the data-control partition follows naturally from the flow of data. This partition of `viper` designates 9 latches to the control logic out of a total of 219 latches. Hence, there are no more than 512 different possible values for the control state, while the full state space can contain as many as 2^{219} states.

Small as the control portion is, it usually exhibits complicated behavior and

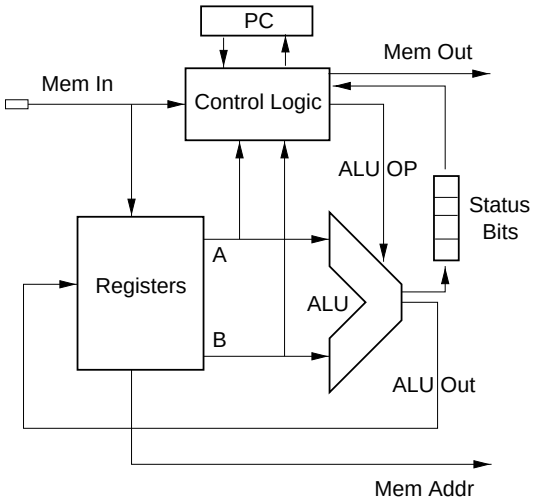


Figure 3.1: Partitioning a design into Control and Datapath

is regarded as the main source of elusive logic bugs. To grasp the extensity and complexity of control logics, consider a microprocessor: imagine that a controller, or a collection of controllers, will have to implement mechanisms such as pipelining, speculative branching, out-of-order execution, and bus arbitration. Because the control portion of a design is error-prone, and the verification of datapath can be best handled by dedicated methods such as BMD (Binary Momentum Diagrams) [22] for arithmetic logics and STE (Symbolic Trajectory Evaluation) [11, 24] for memory arrays [89], it is a reasonable and effective abstraction strategy to focus only on the control behavior in simulation-based functional verification.

"Saturated simulation" attempts to heuristically explore as much of the control portion of the design as possible, by performing a "partial" reachability analysis as follows. It explores, in the iterative computation of the reachable states, all the control *states* or *transitions* available in the next step, but just one representing data

state from a class in which all the data states incur the same control state or transition. As a result, the exploration preserves all distinct control states or transitions at each step while maintaining a minimal data representation. The heuristic is inherently greedy and does not guarantee to cover the whole control space. Nevertheless, saturated simulation can go deeper in the control space than a full reachability analysis, thereby obtaining a higher coverage with regard to control behavior.

The efficiency of this approach comes from the observation it is feasible even for very large designs to compute the image of a small (in the sense of cardinality) set of states. In part, this follows from the fact that the construction of the BDD for the next-state logic can be restricted to the current set of states. This suggests that it may be possible to perform a "partial" reachability analysis, in which all distinct control states are preserved at each step. Additionally, fast BDD routines exist for generating and manipulating representative elements of equivalence classes [75].

Before we go on, we need to point out that we do not study, in this dissertation, the problem of how to separate datapath from the control. This is a subject that has been studied by others [57, 61, 65]. In general, the separation of datapath and control is not always as clear as illustrated in the preceding example; user input is usually required. For example, in [57], the user needs to designate some design signals as the *seed control signals* to start the partitioning.

3.2 Previous Work

We were influenced by the dramatic improvements made to cycle simulation by the use of BDDs by Ashar and Malik [5], and McGeer *et al.* [80], who made clear the importance of making maximum use of the physical memory available on the machine. Ravi *et al.* [86, 96] attempt to pick subsets of state sets encountered during reachability analysis which have small BDDs but contain a large number of states. This is distinct from our approach, wherein a subset is chosen which attempts to maximize the number of distinct controller states. Cho *et al.* [29] pick nets to abstract into primary inputs, consequently obtaining supersets of the set of reachable states. The work of Ho *et al.* [58] and Hoskote *et al.* [60] on creating simulation vectors which excite a large number of transitions on the controller states of a design suggested the usefulness of using transitions rather than states to obtain good coverage of controller behavior. However, they used designer supplied "translation functions", or test-based techniques to generate simulation input sequences which excited as much of the control as possible; our approach is rooted in symbolic methods.

3.3 Preliminaries

Let $X = \{x_1, \ldots, x_n\}$ be a set of Boolean variables, and X' a subset of X. Two minterms $\alpha, \beta \in B^n$ of X are said to be *indiscernible* [78] with respect to X' if they agree on the valuation of the variables in X'. Such pairs of minterms form an *indiscernibility relation,* which we denote by $\xi(X')$. Let $v : B^n \times X \mapsto B$ be the

function that returns the value of a variable in a minterm in B^n, $\xi(X')$ is defined as:

$$\xi(X') = \{(\alpha, \beta) \in B^n \times B^n \mid \forall x_i \in X'.\, v(\alpha, x_i) = v(\beta, x_i)\}. \qquad (3.1)$$

The indiscernibility relation is reflexive, symmetric and transitive, thus is an equivalent relation; for any subset f of B^n, $\xi(X')$ defines a disjunctive partition of f into equivalent classes of minterms, $P(X', f)$, which has as many classes as f has distinct valuations for X'.

An abstraction of $P(X', f)$ can be obtained by keeping exactly one representative pair from each equivalent class. Given a BDD representation of f, this can be done efficiently via the use of the so called `cproject` operator, introduced by Lin *et al.* in [75]. The `cproject` operator takes the BDD for f and a subset X' of the variables in f, and returns a BDD for a function f^*, which is an abstraction of $P(X', f)$ because of the following properties.

1. for any minterm α such that $f(\alpha) = 1$, there is exactly one minterm β such that $f^*(\alpha) = 1$ and $(\alpha, \beta) \in \xi(X')$, and furthermore
2. for all β, $f^*(\beta) = 1 \Rightarrow f(\beta) = 1$.

A BDD implementation of the `cproject` operator is given in Figure 3.2.

3.4 Control State Saturated Simulation

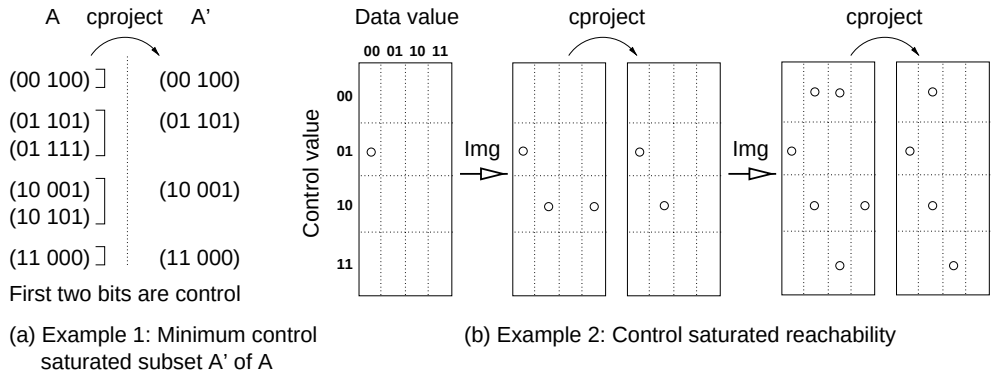
Let the variables associated with the control portion of the design be X_c and the variables associated with the datapath be X_d. Thus the state of the design

```
 /* A --- BDD for set over variables V.  */
 /* V' ⊂ V --- variables being cprojects.    */
BDD_t function BDD_cproject(A, V') {

    v = topVar(A);
    if (v ∉ V') {
        return v · BDD_cproject(A_v, V') + v̄ · BDD_cproject(A_v̄, V');
    }

    T = ∃(V' ⇔ v)A_v;
    if (BDD_Equal(T, BDD_ONE) {
        return v · BDD_cproject(A_v, V');
    }
    else if (BDD_Equal(T, BDD_ZERO) {
        return v̄ · BDD_cproject(A_v̄, V');
    }
    else {
        return v · BDD_cproject(A_v, V') + v̄ · T̄ · BDD_cproject(A_v̄, V');
    }
}
```

Figure 3.2: The `cproject` operator



(a) Example 1: Minimum control saturated subset A' of A

(b) Example 2: Control saturated reachability

Figure 3.3: Minimal control saturated subsets and reachability

is given by an evaluation to, or a minterm of, $X_c \cup X_d$, and a control state is an evaluation to X_c.

Definition 3.1 Let A be a set of states. A subset A' of A is *control-saturated* with respect to A if

$$\forall\alpha[\ \alpha \in A\ \rightarrow\ \exists\beta\,[\,\beta \in A'\ \wedge\ (\alpha,\beta) \in \xi(X_c)\,]\ ]$$

Intuitively, A' is a control-saturated subset of A if every control state occurring in A occurs in A'. Thus a control-saturated subset of A preserves all the controller states present in A. An example of a control-saturated subset is given in Figure 3.3(a).

Since sets can be thought of in terms of characteristic functions, we will freely apply the `cproject` operator to sets to compute the control-saturated subset of A. Observe that this subset is *minimal* in the sense that it has exactly one representative state for each of the equivalent classes of control states.

```
 /* A --- initialized to the initial states.  */
 /* G --- is the BDD for the invariant.    */
BDD_t function Cntrl_Sat_Sim(A, Cntrl_Vars, G) {

    if (BDD_Intersects(A, ¬G))  /* Invariant fails!!  */
        assert FAIL;

    ImgA = BDD_Img(A);
    R = BDD_Or( A, ImgA );
    R* = BDD_Cproject( R, Cntrl_Vars );

    if (BDD_Equal( R*, A))
        return R*;

    return Cntrl_Sat_Sim(R*, Cntrl_Vars, G);
}
```

Figure 3.4: Control-saturated simulation

In Figure 3.4 we sketch a simple symbolic procedure for invariant verification. Reachable states are iteratively computed using the *Img* operator; at each step, a control-saturated subset of the current reached state is computed using the `cproject` operator. This in turn is used as the current reached state set. The first few steps are illustrated in Figure 3.3. The procedure is incomplete, since it is greedy: minimal control-saturated subsets of the sets computed by the `cproject` operator will not necessarily be sufficient to cover all possible controller states.

One simple way of further enhancing the coverage achieved by control-saturated simulation is to generate several "representative" control states. There are simple modifications to the `cproject` operator which can achieve this effect. Another approach is to apply `cproject` only to the frontier of the reached states at each iteration.

3.5 BDD Minimization

We specifically point out two modifications to the `cproject` operator that not only increase the coverage, but also result in reduction of the BDD size. The first one is the randomization of the biasing in `cproject`. The implementation of the `cproject` operator in Figure 3.2 is "biased towards 1", i.e., when presented with a choice for a projection variable, it sets it to 1. This biasing can result in dropping portions of the state space that may be significant (e.g., branch on zero). We overcome this by computing the union of two subsets; one biased towards 0, and one biased towards 1. While no longer minimal, the resulting set has cardinality no more than twice of the original set computed by `cproject`. More generally, we

can define a randomized cproject operator, wherein at each level of the recursion the bias for each variable is selected at random. The results of this new operator can be added to the 0/1 biased subsets to obtain a rich, yet sparse subset.

The second heuristic is the "supersetting" of intermediate results in `cproject`. Consider the expression for the final case of the *BDD_cproject* function listed in Figure 3.2:

$$v \cdot BDD\_cproject(A_v, V') + \overline{v} \cdot \overline{T} \cdot BDD\_cproject(A_{\overline{v}}, V')$$

Replacing $\overline{T}$ by 1 results in a function which computes a superset of the result of $BDD\_cproject$. Intuitively, since the expression is simplified, we reason that the BDD should also be simpler.

3.6 Control Edge Saturated Simulation

A fundamental extension to obtain enhanced coverage is to perform a partial reachability analysis and at each step pick a subset of the image which preserves all "controller transitions" to the image from the current set. Ho *et al.* [58] and Abraham *et al.* [60] created simulation vectors which excite a large number of control transitions in designs; the high quality of their results in terms of finding bugs with these vectors underlines the usefulness of using transitions rather than states to obtain good coverage.

As an example, consider a microprocessor where the control state is the value of the program counter. Two states which correspond to different lines in the

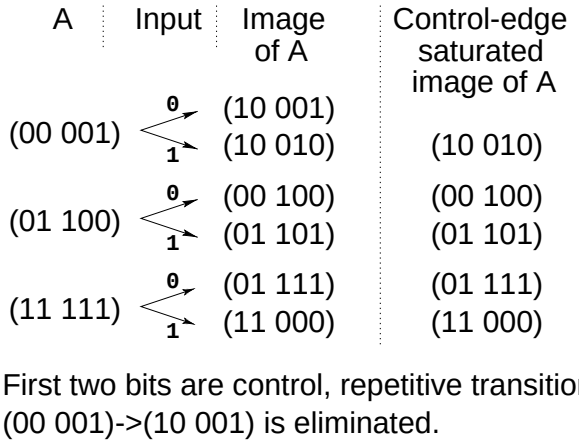


Figure 3.5: Example: A minimal control-edge saturated subset

program may both transition the same program line with different data values; in this case, it is natural to keep the resulting states different.

We now describe how to explore edges in the control state space.

Definition 3.2 Let A be a set of states. A subset B of $Img(A)$ is said to be *control-edge saturated* with respect to A if

$$(\forall\alpha.\forall\alpha')\,[\,[\,\alpha \in A \ \wedge\ \alpha' \in Img(\alpha)\,] \ \rightarrow$$

$$(\exists\beta.\exists\beta')\,[\,\beta \in A \ \wedge\ \beta' \in B \ \wedge\ \beta' \in Img(\beta) \ \wedge\ (\alpha,\beta) \in \xi(X') \wedge\ (\alpha',\beta') \in \xi(X')\,]\,]$$

In English, the above definition says that B is control-edge saturated when for every transition $\alpha \rightsquigarrow \beta$ from A to *Img*(A), there is a state β' in B and a state α' in A so that $\alpha' \rightsquigarrow \beta'$.

Thus in some sense, a control-edge saturated subset of *Img*(A) preserve all the distinct controller transitions originating at A and is as small as possible. Heuris-

tically, a minimal control-edge saturated subset of *Img*(A) is a good representative set — it includes all the distinct controller configurations resulting in *Img*(A) from transitions from A, and is as small as possible. An example of a minimal control-edge saturated subset is given in Figure 3.5.

Minimal control-edge saturated sets can be computed augmenting the design: for every control latch x_C, add a new latch x_S which "shadows" x_C, that is, the next state of x_S is the present state of x_C. Denote the set of shadow state variables thus introduced by X_s. Clearly the next-state of the latches indexed by $X_c \cup X_d$ is independent of that of the shadow latches.

The following lemma demonstrates that minimal control-edge saturated sets can be computed from the augmented design.

Lemma 3.1 Let A^* be A lifted from $X_c \cup X_d$ to $X_c \cup X_d \cup X_s$. Define B to be the existential quantification of `cproject`$(Img(A^*), X_c \cup X_s)$ by X_s. Then B is minimal control-edge saturated with respect to A.

Proof: Observing that `cproject`(Γ, θ) is always a subset of Γ, it follows that `cproject`$(Img(A^*), X_c \cup X_s)$ is a subset of $Img(A^*)$. Since the next state of nonshadow latches does not depend on the shadow latches, it follows that the existential quantification of $Img(A^*)$ by X_s is equal to $Img(A)$, and so B is a subset of $Img(A)$.

We now show B is control-edge saturated with respect to A. Let (α_C, α_D) and (β_C, β_D) satisfy the "if" portion of the implication in Definition 3.2. Then there is a transition from $(\alpha_C, \alpha_D) \in A$ to (β_C, β_D), i.e., $(\beta_C, \beta_D) \in Img(\{(\alpha_C, \alpha_D)\})$.

From the construction of the augmented design, $((\beta_C, \alpha_C), \beta_D)$ is in $Img((\alpha_C, \alpha_S), \alpha_D)$ for an arbitrary assignment α_S to the shadow latches. Hence $\texttt{cproject}(Img(A^*), X_c \cup X_s)$ contains a state of the form $((\beta_C, \alpha_C), \beta_D')$. Note $((\beta_C, \alpha_C), \beta_D{}')$ lies in $Img(A^*)$; let it lie in the image of $((\alpha_C, \alpha_S{}'), \alpha_D{}')$. Hence, on existentially quantifying the X_s variables from $\texttt{cproject}(Img(A^*), X_c \cup X_s)$, the resulting set (namely B) will contain (β_C, β_D'). Since $(\beta_C, \beta_D{}')$ lies in the image of $(\alpha_C, \alpha_D{}')$, $\alpha_D{}'$ and $\beta_D{}'$ are existential witnesses for the "then" portion of the implication in Definition 3.2.

Minimality of B follows from the properties of `cproject` described in the previous section. ■

A minimal control-edge saturated simulation algorithm can be obtained by substituting B as in Lemma 3.1 for $BDD\_cproject$ in the control saturated simulation algorithm shown in Figure 3.4.

3.7 Experimental Results

The saturated simulation described above is implemented as part of the VIS program [20]. Results are provided on two benchmarks – the *8085*, and *viper* microprocessors. The *8085* is approximately 4000 gate equivalents, and contains 242 latches, of which 33 were identified as being control. The *viper* is also 4000 gate equivalents, and contains 218 latches of which 9 were from the control. All experiments were conducted on an UltraSPARC 1, with a 170 Mhz processor, and 128 MBytes of main memory. A timeout of 2000 seconds was used for all *viper* experiments, and 1000 seconds for *8085* experiments. Sifting-based dynamic reordering was enabled throughout the experiments.

Example	|Rchd. States|	Peak BDD	|Ctrl States|	|Ctrl Edges|	Depth
viper	1.36×10^{19}	2033	23	31	4
8085	1.43×10^{7}	275641	1233	3723	10

Table 3.1: Complete BDD based reachability analysis

Example	Peak BDD	|Ctrl States|	|Ctrl Edges|	Depth
viper	160180	246	688	64
8085	81089	1846	4765	43

Table 3.2: Partial reachability analysis using control-state saturated subsets

Table 3.1 presents results on the use of a complete BDD-based reachability analysis on the two benchmarks. Peak BDD is the number of nodes in the largest BDD encountered during reachability analysis. (The abnormally low peak BDD for *viper* in Table 3.1 stems from the fact that the program timed out after the first four reachability steps, which were easily performed.) Table 3.2 presents results on the use of a control-state saturated simulation (as given in Figure 3.4). For *8085*, we compute almost twice as many reachable control states and transitions; for *viper*, an order of magnitude more. Table 3.3 presents results on the use of control-edge saturated simulation. In the same time, more edges are visited; this comes at the expense of higher memory consumption with respect to control-state saturated simulation. Interestingly, fewer control states are visited; we ascribe this to the fact that the control-state saturated simulation is faster, and so manages to go deeper into the state space in the same amount of time; this is seen in the depth column.

Example	Peak BDD	|Ctrl States|	|Ctrl Edges|	Depth
viper	71213	236	705	60
8085	81089	1696	6324	30

Table 3.3: Partial reachability analysis using control-edge saturated subsets

Example	Saturated Simulation			Cycle Simulation			
	T (sec)	|States|	|Edges|	T (sec)	Size	|States|	|Edges|
viper	2000	236	705	86616	1000×200	121	288
8085	1000	1696	6324	99143	4000×200	705	2674

Table 3.4: Comparing saturated simulation with cycle simulation

We compare saturated simulation with fast lookup based cycle simulation [5, 80] in Table 3.4. For *viper*, we performed 1000 sets of simulations, each comprising of 200 vectors; for *8085* we performed 4000 sets of length 200. Even though we gave cycle simulation two orders of magnitude more time, it still performed far worse than saturated simulation.

3.8 Summary

We investigated a method of applying symbolic simulation in invariant checking. Symbolic simulation, due to its use of BDDs, has a larger capacity in terms of design behavior coverage, comparing to a traditional simulation. We also provided an abstraction mechanism for selectively exploring control states and transitions, which further improved the effectiveness of the verification.

Chapter 4

Retrograde Analysis

4.1 Introduction

Retrograde Analysis (RA) is an important search technique developed within the field of Artificial Intelligence. The name characterizes the main ingredient of the technique — a backward search starting from the goal. In [106], Ken Thomson employed this technique in the course of analyzing certain chess endgames. He approached the problem by first marking all positions W_0 where white wins, then computing the positions W_1 from which there is a move for white following every move by black that leads to a position in W_0; clearly these are also winning positions for white. Iteratively, he determined the sets $W_0, W_1, W_2, \ldots$ and used them as the new goals in the search of a winning strategy.

The primary benefit of RA is that the set $\cup_i W_i$ typically contains many more positions than W_0; hence, in a heuristic sense, the former offers a much larger "target" for the search. Furthermore, since the positions in W_i is i moves closer to the starting position than the ones in W_0, the search targeting them will span a shorter distance in the game space. Follow this line of thinking, we can improve RA by also "enlarging" the set of initial positions. The new set is the union of the original initial positions and the ones they can reach within certain number of

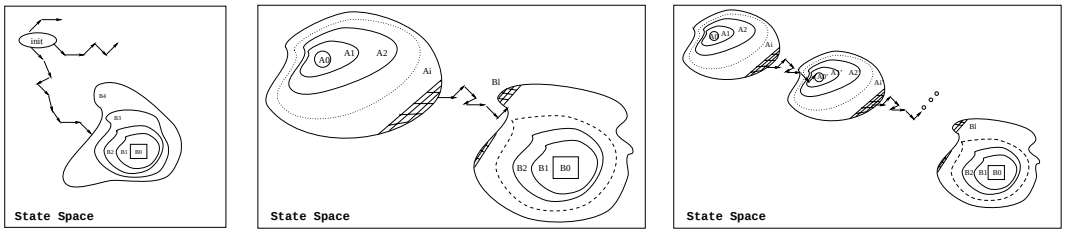
moves. The intention is to further reduce the search distance.

We formulate invariant checking as a RA problem, with the above improvement, by treating the complement of the invariant as the end positions. We also provide a heuristic for selecting candidate starting states from the enlarged set of initial states. As we have seen in Section 2.3, this approach is essentially a backward image computation, thus is orthogonal to saturated simulation based on forward image computation.

4.2 The Implementation

Our RA-based invariant checking consists of two phases, the preparation phase and the search phase. In the first phase, we iteratively construct the series $B_0, B_1, \ldots$ where B_0 is the complement of the invariant and $B_{i+1} = PreImg(B_i)$. Analogous to the end positions, the B_i's are effectively bad states. Since B_i's can grow very large in terms of cardinality, it is natural to resort to BDDs for compact representations. The BDDs, however, may still exhaust the main memory eventually. Therefore, we conclude the first phase either right before this happens, for maximum resource utilization, or simply after a certain number of steps.

In the search phase, we look for an input sequence that takes a starting state to a state in the target $\cup_i B_i$. Several search strategies with ascending levels of sophistication are considered. The simplest strategy is the simulation of random input vectors starting from a random initial state; the search halts if some state reached in this fashion lies in the target. This approach is illustrated in Figure 4.1(a). Note that checking if a state lies in the target defined by a BDD can be performed



(a) Vanilla RA (b) RA with Hamming; closest states are hatched (c) Enhanced RA with Hamming

Figure 4.1: Retrograde search for Invariant checking

in time proportional to the number of variables in the BDD, which is independent of the BDD size.

A more involved strategy is to pick an initial state which is "close" to the target states. We propose the use of *Hamming distance* as the measure of closeness.

The Hamming distance between two vectors $\alpha, \beta \in \{0,1\}^n$ (denoted by $\Delta(\alpha, \beta)$) is the number of positions in which the α and β vectors differ. Consider the relations $H_0, H_1, H_2, \ldots, H_n \subset \{0,1\}^{2n}$ where $(\alpha, \beta) \in H_k$ iff $\Delta(\alpha, \beta) \leq k$. The relation H_1 can be constructed directly using BDDs. The relation H_{i+1} satisfies the following identity:

$$H_{i+1} \;=\; H_i \;\cup\; (\exists\gamma)[(\alpha,\gamma) \in H_i \wedge (\beta,\gamma) \in H_1]$$

Hence, the BDDs for $H_0, H_1, H_2, \ldots, H_n \subset \{0,1\}^{2n}$ can be easily constructed; furthermore a simple argument based on counting cofactors shows that they are small for the interleaved variable ordering.

The search for states in the target can be further enhanced by first computing a series of images of the set of initial states. From the outermost image, pick a

state (say α) which is closest to the target, and then perform random cycle simulation from α. This is illustrated in Figure 4.1(b). Instead of cycle simulation from α, a combination of symbolic forward reachability analysis coupled with the the Hamming heuristic can be recursively applied. This is illustrated in Figure 4.1(c).

4.3 Experimental Results

Retrograde analysis is coded as part of the VIS program, and experimented with a number of examples. Representative results are provided on two benchmarks – *Mesh4* is a routing algorithm on a 4 by 4 mesh of nodes, and *Cube4* is hypercube based routing protocol. For both examples, we chose an invariant which fails.

Results on *Mesh4* are reported in Figure 4.2. We plot BDD size and cardinality after successive pre-images in Figure 4.2(a); both grow quickly. In Figure 4.2(b) we plot the number of simulation trials needed to reach a pre-image, starting from the initial state against the number of pre-image steps taken; each trial consists of applying 100 random vectors. It is clear from the picture that this number decreases rapidly.

The effect of Hamming distance is given in Figure 4.3 for the *Cube4* example. Figures 4.3(a) and 4.3(b) are as before. In Figure 4.3(c), we show the effect of taking one forward step, and then picking a state in the image which is close to the target as opposed to a random state in the image; in Figure 4.3(d) we take two forward steps, and then pick a state which is close to the target. In both cases, there is an appreciable decrease in the number of simulation trials needed when Hamming distance is used. Interestingly, when a state in the image is picked at random, the

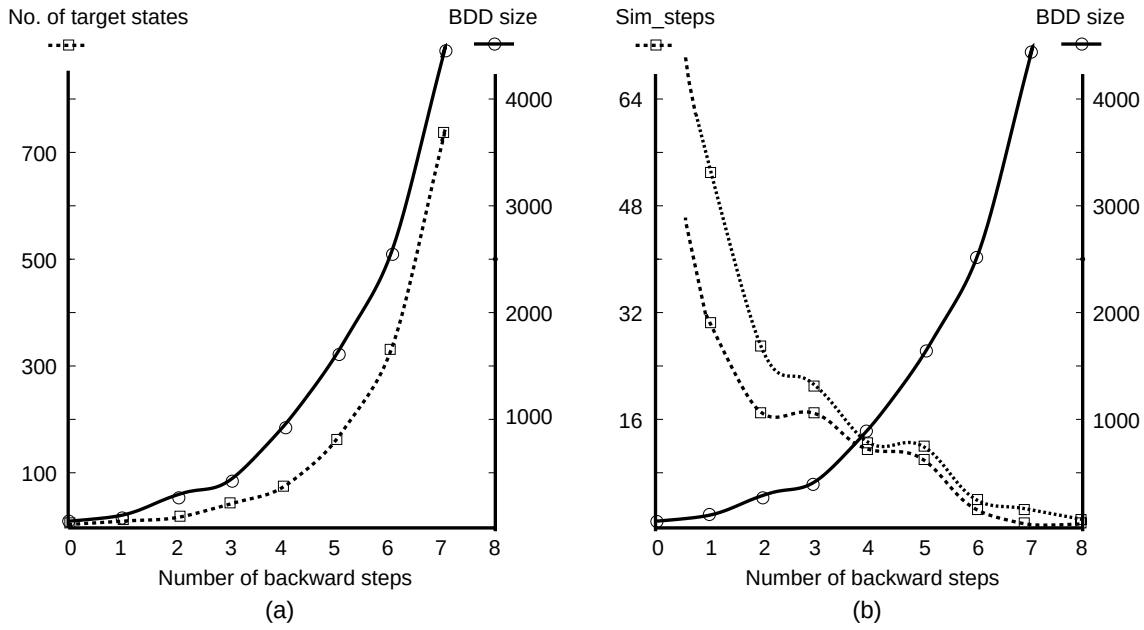


Figure 4.2: Retrograde Analysis applied to *Mesh4*

performance is actually worse than simply starting at the initial state.

4.4 Summary

We presented a symbolic method based on retrograde analysis, and a heuristic, in the application of invariant checking. Although aimed at the same problem, this method, and saturated simulation reported in the last chapter, are orthogonal in their search strategies. Experimental evidence corroborates that both approaches yield enhanced coverage and robustness. Thus the combination of formal and informal verification offers benefits not available in each independently.

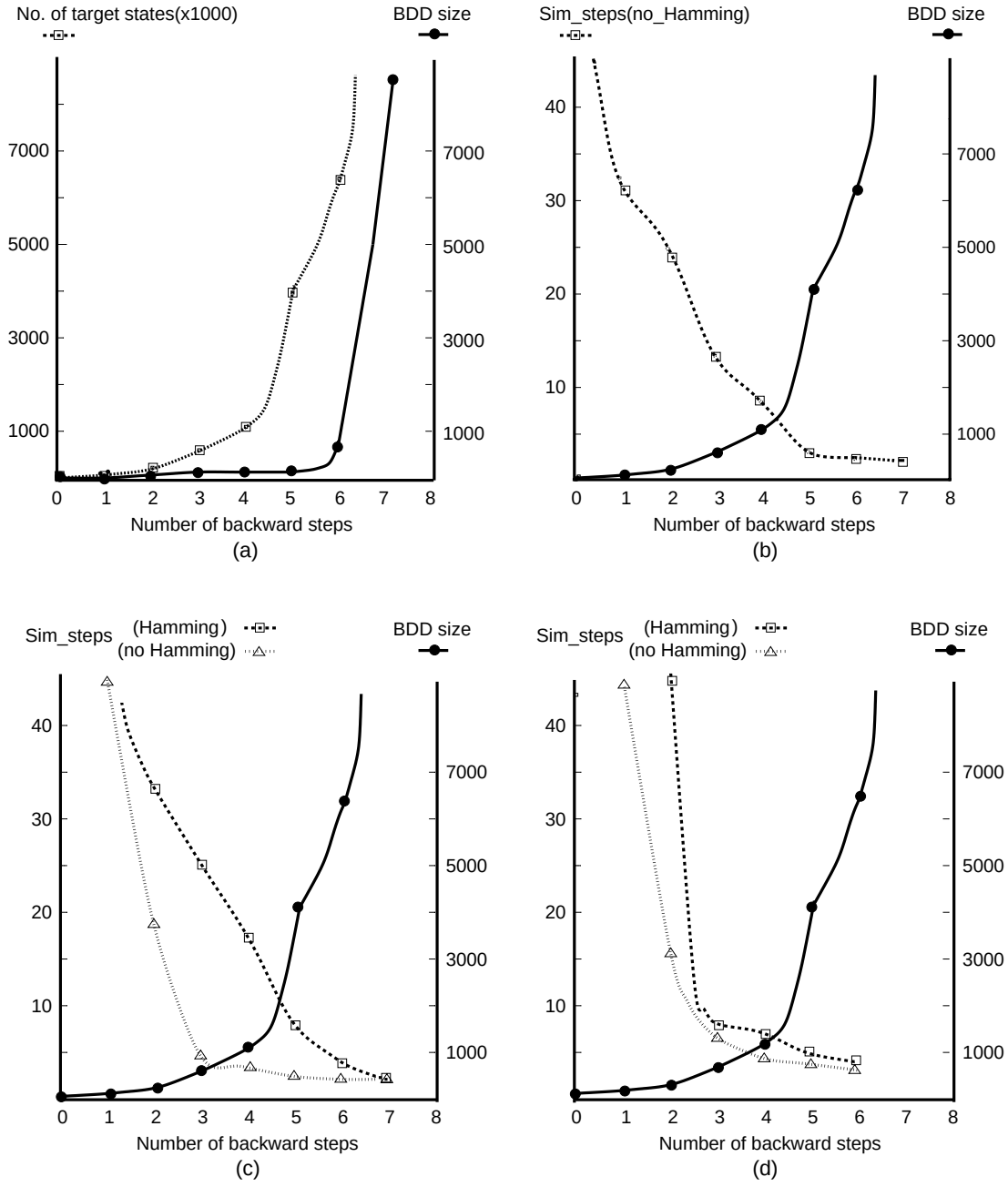


Figure 4.3: Effect of Hamming Distance on *Cube4*

Chapter 5

Constrained Vector Generation

5.1 Introduction

While there is a wide range of verification methodologies [93], simulation is still the prevalent form of functional verification of commercial designs [37, 38, 46, 48, 85, 104]. An important task in simulation is the generation of simulation vectors, which is time-consuming and error-prone, especially in the presence of complex interactions between the design and its environment. Vector generation methods fall into two major categories, namely the deterministic and the random-based. The former refers to the generation of vectors either manually by engineers with detailed understanding of the design, or automatically by programs using ATPG (Automatic Test Pattern Generation) techniques. This leaves the method highly sensitive to the complexity of the design. In practice, a design is usually simulated first with a relatively small set of vectors created deterministically, then with random vectors for as long as is feasible. Therefore, random vector generation is of great importance to simulation-based verification.

In this chapter, we describe a symbolic generator of random vectors, *SymGen*, using constraints represented by BDDs. The first requirement for any vector generation tool is to define, and generate within, the legal input space, which of-

ten is a function of the state of the design. In addition, a robust tool should also provide the ability to influence the distribution of the generation, thus directing the simulation towards the "corner" cases, or other test scenarios deemed interesting. SymGen strives to meet these important requirements, in a unified fashion, using Boolean constraints and input biases in an efficient algorithm. The algorithm consists of two procedures: the first computes the weights of vectors, composed from constraints and biases; the second generates vectors according to probabilities derived from the weights. Both procedures operates symbolically using BDDs. The algorithm is immune to the *backtracking* problem that is typical to many vector generation tools, as well as independent of variable orderings, despite of the use of BDDs.

The rest of this chapter is organized as follows. Section 5.2 reviews the related work. Section 5.3 describes how environment constraints and input biases are represented, and provides a mechanism for combining the effects of the two. The vector generation algorithm is presented in Section 5.4, while the implementation issues are discussed in Section 5.5. Experimental results and a case study are given in Section 5.6. Section 5.7 summarizes this chapter.

5.2 Previous Work

In [50], a tool called RIS implemented a static-biased random generation technique that allows the user to bias the simulation generation within a restricted set of choices — all of which satisfy the constraints. However, the biasing is static in that it is independent of the state of design. To provide more biasing flexibility,

Aharon implemented a dynamically-biased test generator for a tool called RTPG [2, 3], which decides the next input based on the current state of the design. The primary drawback for the simulation generator is the effort required to produce the functional model. A tool introduced in [28] used various constraint solving techniques tailored for specific instructions. A problem is that one may need to *backtrack* and perform a heuristic search to resolve the "dead end" cases.

BDDs have found many applications in design and verification problems. The Algebraic Decision Diagram (ADD) [10], which is an extension to the BDD, was used in [56] to represent the matrix of a state transition graph and compute the steady-state probabilities for Markov chains. Although simulation generation was not the concern of that paper, the experiments showed that symbolic methods can handle very large systems in which direct equation solving methods cannot.

In [15], binary graphs were used to represent Boolean functions, so that the probability distribution of the output can be computed recursively from the input probabilities of the function. This technique was used to probabilistically decide the equivalence of Boolean functions. A similar approach was adopted in [69] to compute exact fault detection probabilities for some given nets in a design.

Both [15, 69] are related to our work from the point of view that probabilities are computed recursively using a decision diagram. But this is where the similarity ends. Specifically, we do not deal with the probability of the output of a function or a design net. The problem we are facing is to generate input vectors which satisfy a set of constraints, and are probabilistically influenced by the input distribution, or biasing. The constraints and biasing we consider can both be state-dependent.

5.3 Constraints and Biasing

5.3.1 Constraints for Environment Modeling

Simulation by random vectors is meaningful only if the vectors meet certain requirements modeling the environment of the design. For example, a design may prohibit some input combinations, or expect the inputs to follow some patterns under specific states. These requirements are *relational* by nature, thus suggesting the use of constraints. SymGen adopts this approach by generating vectors in the state-depend *legal input space*, defined by the conjunction of environment constraints expressed in Boolean functions of input and state variables. As an example, consider a typical assumption about bus interfaces: the "transaction start" input (ts) is asserted only if the design is in the "address idle" state. Syntactically, this is captured in the following SymGen constraint:

$$\text{\$constraint}(ts \rightarrow (\text{addr\_state} == \text{ADDR\_IDLE}));$$

SymGen handles a rich class of constraints because of the constraints' dependency on state. For example, by adding auxiliary variables to remember past states, SymGen can constrain the sequential behavior of the inputs; this is discussed in Section 5.6.2.

Comparing to the *testbenches*, the traditional driver-program approach to environment modeling, the advantage of constraints is manifold. First, constraints are declarative, as opposed to the constructive approach of testbenches, and thus need less effort on the part of the user. This is especially helpful in a prototyping stage when all that is known about the environment are some abstract specifications

in the architecture book. Constraints also form a modular and more formal interface documentation about design blocks; by contrast, a testbench constitutes an unmaintainable and unverifiable documentation of the environment. Finally, constraints automatically convert to properties to be monitored at a higher level of hierarchy; thus, in a sense, use of constraints can be viewed as an assume/guarantee methodology.

5.3.2 BDD Representation of Constraints

The Boolean functions of constraints in SymGen are implicitly represented by BDDs. In the sequel, we use *constraint BDD* to refer to the conjunction of the BDDs of all constraints, unless otherwise stated. Recall we showed in Section 2.5 that the BDD as a representation of a Boolean function is automatically a representation of the onset of that function. Bearing this in mind, we say that the legal input space defined by a constraint is captured by the set of paths in the corresponding BDD that lead to the terminal node *ONE*, in the following sense: each such path can be viewed as an assignment to the variables on that path; the state variable assignment (a *cube*) represent a set of states, whereas the input assignments represent a set of input vectors that are legal under each of these states.

The above derivation of the legal input space for a given state is effectively a computation of the constraint BDD's *cofactor* relative to that state. Depending on the constraint, the legal input space can be empty under certain states, which are referred to as the *illegal states*, or more intuitively, the dead-end states, since the simulation cannot proceed upon entering them. For instance, consider the constraint

$(s_1 + s_2 + x_1 + x_2 \leq 1)$, where s_1 and s_2 are state variables, and x_1 and x_2 are inputs: the state $(s_1 = 1, s_2 = 1)$ is an illegal state, since no assignments to x_1 and x_2 can satisfy the constraint; states $(s_1 = 1, s_2 = 0)$ and $(s_1 = 0, s_2 = 1)$ has one legal input vector, $(x_1 = 0, x_2 = 0)$; and the legal input vectors for state $(s_1 = 0, s_2 = 0)$ are the ones satisfying $(x_1 + x_2 \leq 1)$.

Before we move on, we introduce some notations about BDDs that will facilitate our exposition. We will not distinguish a constraint and its BDD representation; we will use $r(f)$ to denote the root node of a BDD f, $v(\sigma)$ to denote the variable corresponding to the node σ, and $t(\sigma)$, $e(\sigma)$ to denote the $then$ and $else$ nodes of σ, respectively; we will also use the convention that X stands for the input variables, and Y for the state variables, and will use them without further declaration.

5.3.3 Input Biasing

It is often the case that in order to exercise the design in "interesting" scenarios, one needs to "bias" the inputs. In SymGen, we express the biases with *input probabilities*.

Definition 5.1 The *input probability* of $x = 1$ is a function of the state with a range in (0,1), denoted by $p^x(Y)$; the *input probability* of $x = 0$ is the function $1 \Leftrightarrow p^x(Y)$, denoted by $p^{\overline{x}}(Y)$.

Note we exclude 0 and 1 from possible values of input probabilities since they impose "hard" constraints and should be expressed as what they are, i.e., by

the constraints $x == 0$ and $x == 1$, respectively. An input probability can be given either as a constant, or as a function in a Verilog [27] expression, which can be evaluated natively in many commercial simulators. The following statement shows an example of setting the input probability p^x, which evaluates to 0.2 when st is UP, to 0.8 when st is $DOWN$, and to 0.5 otherwise.

$setprob1($x$, st==UP ? 0.2 : st==DOWN ? 0.8 : 0.5);

Input probabilities are "soft" restrictions since they can be "reshaped" by the constraints, which assume higher priority. In extreme cases, the constraints can prohibit an input from taking a specific value at all times albeit the input may be assigned a high probability of doing so.

5.3.4 Constrained Probabilities

To handle the combination of constraints and input probabilities in a unified way, we introduce the notion *constrained probability* of an input vector. First, we define an auxiliary term *weight of an input vector*.

Definition 5.2 Let $\alpha = \alpha_1\alpha_2 \cdots \alpha_n$ be a vector of input variables $x_1, \ldots, x_n$. The *weight* of α, denoted by $\pi(\alpha, Y)$, is given by

$$\prod_{i=1}^{n} [\alpha_i \cdot p^{x_i}(Y) + (1 \Leftrightarrow \alpha_i) \cdot p^{\overline{x_i}}(Y)] \tag{5.1}$$

The constrained probability is then defined in the following:

Definition 5.3 Let f be a constraint, s a state, and f_s the legal input space which is the cofactor of f with respect to s. The *constrained probability* of an input vector α, under s, is 0 if $\alpha \notin f_s$; otherwise, it is given by

$$\frac{\pi(\alpha, s)}{\sum_{\beta \in f_s} \pi(\beta, s)}$$

Conceptually, the constrained probability of an input vector is the weight of that vector divided by the sum of the weights of all vectors that satisfy the constraint; the sum is zero if the given state is an illegal state.

5.3.5 An Example of Constrained Probability

Consider a design with four inputs, cmd[3], cmd[2], cmd[1] and cmd[0], and with the corresponding input probabilities 1/2, 1/3, 1/4, and 1/5. When there are no constraints, all vectors are possible and each has the probability that is the product of input probabilities, as shown in Table 5.1. (Middle vectors are removed for brevity.) Note that the sum of all the vector probabilities is 1.

Now we add a constraint

```
$constraint( cmd[3:0]==4'b1000 || cmd[3:0]==4'b0100 ||
             cmd[3:0]==4'b0010 || cmd[3:0]==4'b0001 );
```

which restricts our choices to the four vectors shown, enforcing a "one-hot" property among the inputs. These vectors and their "probabilities" (actually, *weights*, since the sum is less than 1 now) are given in Table 5.2.

1/2 cmd[3]	1/3 cmd[2]	1/4 cmd[1]	1/5 cmd[0]	probability of vector
0	0	0	0	1/2 · 2/3 · 3/4 · 4/5 = 24/120
0	0	0	1	1/2 · 2/3 · 3/4 · 1/5 = 6/120
0	0	1	0	1/2 · 2/3 · 1/4 · 4/5 = 8/120
		⋮		⋮
1	1	0	1	1/2 · 1/3 · 3/4 · 1/5 = 3/120
1	1	1	0	1/2 · 1/3 · 1/4 · 4/5 = 4/120
1	1	1	1	1/2 · 1/3 · 1/4 · 1/5 = 1/120
				$\sum$ = 120/120 = 1

Table 5.1: Example: explicit computation of vector probabilities

1/2 cmd[3]	1/3 cmd[2]	1/4 cmd[1]	1/5 cmd[0]	unnormalized weight of vector
0	0	0	1	1/2 · 2/3 · 3/4 · 1/5 = 6/120
0	0	1	0	1/2 · 2/3 · 1/4 · 4/5 = 8/120
0	1	0	0	1/2 · 1/3 · 3/4 · 4/5 = 12/120
1	0	0	0	1/2 · 2/3 · 3/4 · 4/5 = 24/120
				$\sum$ = 50/120 $\neq$ 1

Table 5.2: Example: explicit computation of vector weights under constraints

Finally, the constrained probabilities are obtained by normalizing the weights with regard to the total weights of the legal vectors. The results are 3/25, 4/25, 6/25, and 12/25, respectively for the vectors in Row 1 through Row 4.

The drawback of the above tableau approach is obvious: the cost is in the order of 2^n for n inputs. In the next section we present a method that computes the constrained probabilities implicitly in BDDs. Its efficiency, hinging on the compactness of BDD representation of Boolean functions, is evident from the experi-

ments conducted on commercial designs, as will be reported in Section 5.6.

5.4 Simulation Vector Generation

We now develop a constrained vector generation algorithm based on implicit computation of constrained probabilities. The algorithm, consisting of two procedures *Weight* and *Walk*, proceeds as follows: *Weight* computes the *weights* of the nodes in the constraint BDD for a given state; Depending on the result, the algorithm either terminates because of the state being illegal, or starts the *Walk* procedure to generate an input vector; *Walk* traverses the constraint BDD according to *branching probabilities* derived from weights of nodes. The traversed path, together with random assignments to input variables not on that path, identifies an input vector that holds the following properties:

Property 1: The vector is a legal vector.

Property 2: The vector can be any legal vectors.

Property 3: The vector is generated with its constrained probability as given in Definition 5.3.

The first two properties are necessary for an ideal simulation vector generation process that produces *only* and *all* the vectors that satisfy the constraint. The third property provides a utility for controlling the distribution of the generated vectors. Because the algorithm operates on a BDD annotated with probabilities, we call it the *p-tree* algorithm.

5.4.1 The *Weight* Procedure

First, we define what we mean by the *weight* of a BDD node under a particular state.

Definition 5.4 Given a constraint and the set of state variables Y, the weight of node σ, denoted by $\omega(\sigma, Y)$, is inductively given by the rules:

1. $\omega(ONE, Y) = 1$, $\omega(ZERO, Y) = 0$

2. Let Y be the set of state variables, and v the variable corresponding to ω; let t and e be the $then$ and $else$ nodes of ω, respectively. Then

$$\omega(\sigma, Y) = \begin{cases} p^v(Y) \cdot \omega(t(\sigma), Y) + p^{\overline{v}}(Y) \cdot \omega(e(\sigma), Y) & \text{if } v \text{ is an input variable} \\ \omega(t(\sigma), Y) & \text{else if } v = 1 \\ \omega(e(\sigma), Y) & \text{else} \end{cases} \tag{5.2}$$

The *Weight* procedure, as shown in Figure 5.1, applies the above computation of node weights recursively to the constraint BDD in a depth-first order. The following notations are used: *node*.var represents the variable associated with a BDD *node*; *node*.then and *node*.else represent the child nodes of *node*, for the assignments *node*.var=1 and *node*.var=0, respectively.

Weight performs a one-pass computation of node weights through the constraint BDD. A straight-forward upper bound on the time complexity of *Weight* is $O(n)$, for a constraint BDD with n nodes. Note, however, that the procedure traverses only a subgraph in the BDD because it explores only one branch of each state node encountered. Further, we point out that in this subgraph, the nodes with

```
 /* s is the current state.  */
 /* The recursion start with */
 /* the root node.  */
ω(σ, s) {

    if (σ == ONE) return 1;
    if (σ == ZERO) return 0;
    if (σ is visited) return σ.weight;
    set_visited(σ);

    let u be the variable v(σ);
    if (u is an input variable) {
        w_t = ω(t(σ),s);
        w_e = ω(e(σ),s);
    }
    else if (u is a state variable) {
        if (u == 1)
            σ.weight = ω(t(σ),s);
        else
            σ.weight = ω(e(σ),s);
        return σ.weight;
    }

    σ.weight = p^u(s) · w_t + p^ū(s) · w_e;

    return σ.weight;
}
```

Figure 5.1: Procedure *Weight*

positive (nonzero) weights form yet another subgraph which identifies the current legal input space; as we showed in Section 5.3.1, the latter subgraph is effectively the cofactor of the BDD with respect to the current state. This cofactoring is "in-place" in the sense that it never creates new BDD nodes, thus avoiding potential BDD size explosions of a normal BDD cofactoring. For these reasons, *Weight* is fairly efficient in practice even when the constraint BDD is quite large.

The second function of *Weight* is to determine whether the current state is legal, i.e., allows some satisfying input assignment. If it is, then we continue with the vector generation process; otherwise, we have to abort the simulation and start debugging the constraints. The following theorem provides such a test based on the result of *Weight*. Recall $r(f)$ returns the root node of the BDD of f.

Theorem 5.1 Given a constraint f, a state s is a legal state iff

$$\omega(r(f), s) > 0.$$

Proof: Input probabilities are always greater than 0, so are the weights of any input vectors. Therefore, the existence of satisfying input vectors which indicates the state is legal, is equivalent to the sum of weights of satisfying vectors being positive.

The theorem is then the immediate result of Lemma 5.1, below, which says that the weight of the root node of f is the sum of weights of satisfying input vectors under s in f. ■

Lemma 5.1 Given a constraint BDD f and a state s,

$$\omega(r(f), s) = \sum_{\alpha \in f_s} \pi(\alpha, s)$$

where f_s is the set of legal vectors in f under the s.

Proof: We proof by induction on the number of variables in f. First, let $sum(f, s)$ abbreviate $\sum_{\alpha \in f_s} \pi(\alpha, s)$.

If f is a function with no variables, thus a constant, then for any s: (1) If $f = 1$, then $\omega(ONE, s) = 1$, also, $sum(1, s) = 1$ since all input vectors are legal; (2) Similarly, if $f = 0$, both $\omega(ZERO, s)$ and $sum(0, s)$ are 0.

If f has one variable, there are also two cases: (1) If the variable is an input, without loss of generality, let $f = x$. Then $\omega(x, s) = p^x(s)$, which is equal to $sum(x, s)$ since $x = 1$ is the only legal vector; (2) If the variable is a state variable, without loss of generality, let $f = y$. Then $\omega(y, y = 1)$ returns the weight of ONE, 1, and $sum(y, y = 1)$ is also 1 since all vectors are legal; similarly, $w(y, y = 0)$ is the weight of $ZERO$, 0, and $sum(y, y = 0)$ is 0 since no vectors are legal.

Now, we prove the induction hypothesis. Let the lemma hold for the two child BDDs of f, g and h, of variables $u_1, \ldots, u_{n-1}$; let u_n be the new variable in f, and without loss of generality, let $f = u_n \cdot g + \overline{u_n} \cdot h$. Again we have two cases.

(1) u_n is an input variable: From Equation 5.2, we have

$$\omega(r(f), s) = p^{u_n}(s) \cdot \omega(r(g), s) + p^{\overline{u_n}}(s) \cdot \omega(r(h), s)$$

by induction hypothesis, we get

$$\omega(r(f), s) = p^{u_n}(s) \cdot sum(g, s) + p^{\overline{u_n}}(s) \cdot sum(h, s)$$

The right-hand-side, due to Definition 5.2, becomes

$$\begin{aligned}
& p^{u_n}(s) \cdot \sum_{\alpha \in g_s} \prod_{i=1}^{n-1} \left[\alpha_i \cdot p^{u_i}(s) + (1 \Leftrightarrow \alpha_i) \cdot p^{\overline{u_i}}(s)\right] + \\
& p^{\overline{u_n}}(s) \cdot \sum_{\alpha \in h_s} \prod_{i=1}^{n-1} \left[\alpha_i \cdot p^{u_i}(s) + (1 \Leftrightarrow \alpha_i) \cdot p^{\overline{u_i}}(s)\right] \\
= & \sum_{\beta \in u_n \cdot g_s} \prod_{i=1}^{n} \left[\beta_i \cdot p^{u_i}(s) + (1 \Leftrightarrow \beta_i) \cdot p^{\overline{u_i}}(s)\right] + \quad ;; \beta_n = 1 \\
& \sum_{\alpha \in \overline{u_n} \cdot h_s} \prod_{i=1}^{n} \left[\alpha_i \cdot p^{u_i}(s) + (1 \Leftrightarrow \alpha_i) \cdot p^{\overline{u_i}}(s)\right] \qquad ;; \beta_n = 0 \\
= & \sum_{\alpha \in f_s} \prod_{i=1}^{n} \left[\alpha_i \cdot p^{u_i}(s) + (1 \Leftrightarrow \alpha_i) \cdot p^{\overline{u_i}}(s)\right] \\
= & \sum_{\alpha \in f_s} \pi(\alpha, s) \\
= & \; sum(f, s)
\end{aligned}$$

(2) u_n is a state variable: If $s' = s \cdot u_n$, then $f_{s'} = g_s$, therefore $sum(f, s') = sum(g, s)$; further, from Equation 5.2, we have $\omega(f, s') = \omega(g, s)$. Hence, by induction hypothesis, $\omega(f, s') = sum(f, s')$. The case $s' = s \cdot \overline{u_n}$ can be proved similarly. ■

5.4.2 The *Walk* Procedure

If the current state is a legal state, we proceed to actually generate a vector. For a quick intuition, we say that our generation procedure resembles the reverse

of evaluating a BDD for a given vector, in the sense that we take branches and assign values accordingly, whereas the latter uses existing assignments to guide the branching. So the key to our method is how the branches are taken. Our solution is to follow the *branching probabilities*, built up from the weights just computed, in the following way.

Definition 5.5 Let σ be an input node with a positive weight, and u the associated variable. Let s be the state. The *branching probabilities* of σ are given in the following equations:

$$\sigma.\text{then\_prob} = [p^u(s) \cdot \omega(t(\sigma), s)]/\omega(\sigma, s) \tag{5.3}$$

$$\sigma.\text{else\_prob} = [p^{\overline{u}}(s) \cdot \omega(e(\sigma), s)]/\omega(\sigma, s) \tag{5.4}$$

Note that

$$\sigma.\text{else\_prob} + \sigma.\text{else\_prob} = 1. \tag{5.5}$$

We intend to use the branching probabilities to guide a random traversal in the constraint BDD, and generate a vector as follows: the traversal starts from the root nodes; at a state node, it takes the *then* (resp. *else*) branch if the corresponding state variable evaluates to 1 (resp. 0) in the current state; at an input node, it takes a branch according to the *branching probabilities* of the node, and sets the value of the corresponding input variable accordingly, i.e., to 1 if a $then$ branch was taken, to 0 otherwise. This procedure is implemented in Figure 5.2, and is named *Walk*.

Two properties, however, need to be proved in order for *Walk* to be valid. First, we show that all the nodes that can be visited must have positive weights,

```
Walk(node,st) {
    if (node == ONE)   return;
    if (node == ZERO)   error;

    let v be the variable node.var;
    if (v is a state variable) {
        if (v==1)
            Walk(node.then);
        else Walk(node.else);
    } else {
        node.then_prob = p(v) · t / node.weight;
        let r = random(0,1);
        if (r < node.then_prob) {
            assign node.var to 1;
            Walk(node.then);
        } else {
            assign node.var to 0;
            Walk(node.else);
        }
    }
}
```

Figure 5.2: Procedure *Walk*

since these are the only nodes where the branching probabilities are defined. The following theorem states exactly this.

Theorem 5.2 Under a legal state, Procedure *Walk* only visits nodes with positive weights.

Proof: We use an inductive argument: First, from Lemma 5.1, under any legal state the weight of the root is greater than 0; further, by Definition 5.4, any node with a positive weight must have at least one child node with a positive weight, therefore due to Definition 5.5, at least one of its branching probabilities is greater than 0; so the traversal in *Walk* from that node must take a branch with a positive (not a 0) possibility thus reaching a node with a positive weight. Hence by induction, every visited node must have a positive weight. ■

As a result of the above theorem, the last node of the traversal, too, must have a positive weight, and therefore must be node *ONE*.

Corollary 5.1 Under a legal state, Procedure *Walk* terminates only at node *ONE*.

Since the constraint BDD is satisfied by assignments corresponding to any path from the root to node *ONE*, this corollary asserts the second property we wanted to show about *Walk*: it generates only the legal vectors. The corollary, from a different viewpoint, also proves our claim that *p-tree* does not backtrack: if there exist any legal input vectors, it guarantees to find one in one pass.

One point worth mentioning is that, since the input assignments made during *Walk* have already satisfied the constraint, we are totally free in assigning the

variables not visited in the traversal: in our case, we choose to do it according to their input probabilities. The default 0.5 is assumed if an input probability is not specified.

Finally, regarding the complexity, *Walk* always terminates in less than k steps on BDDs with k variables, since that is the length of the longest paths. Overall, since k is a much smaller number than the number of nodes, the time and space complexities of *Weight* and *Walk* combined are linear to the size of the constraint BDD.

5.4.3 Correctness and Properties

Recall that our goal was to generate vectors with the three properties given at the beginning of this section. We now show that the *Weight* and *Walk* procedures have achieved this goal.

The first property, which says *p-tree* generates only legal vectors, is guaranteed by Corollary 5.1, since the constraint BDD is satisfied by assignments corresponding to any path from the root to node *ONE*.

The second property, that the algorithm can generate all legal vectors, is satisfied by the third, which states that the generation follows the constrained probabilities, because of the following: due to Definition 5.3, the set of legal vectors is exactly the set of vectors with positive constrained probabilities, therefore if the third property holds, each legal vector will have a greater than 0 chance to be generated. We restate the third property in the following theorem.

Theorem 5.3 Procedure *Walk* generates input vectors according to their constrained probabilities.

Proof: First, from Corollary 5.1, *Walk* always ends at node *ONE*, thus it never generates illegal input vectors, whose constrained probabilities are 0 by Definition 5.3.

Now consider legal vectors. Let f be the constraint and s the state. Let $\sigma_1, \ldots, \sigma_m, \sigma_{m+1}$ be the sequence of nodes visited in a *Walk* traversal, where σ_1 is the root and σ_{m+1} the node *ONE*. Without loss of generality, assume that the input variables corresponding to this sequence are $x_1, \ldots, x_m$, and that $x_{m+1}, \ldots, x_n$ were not visited. Also, let $\alpha_1 \cdots \alpha_n$ be the generated vector, wherein the first m values correspond to the branches taken in the traversal, and the last $n \Leftrightarrow m$ values correspond to the choices based upon input probabilities. For brevity, let $p^{u,b,s}$ denote $b \cdot p^u(s) + (1 \Leftrightarrow b) \cdot p^{\overline{u}}(s)$. Then, the probability of this vector is given in the following product:

$$\frac{p^{x_1,\alpha_1,s} \cdot \omega(\sigma_2, s)}{\omega(\sigma_1, s)} \cdot \frac{p^{x_2,\alpha_2,s} \cdot \omega(\sigma_3, s)}{\omega(\sigma_2, s)} \cdots \frac{p^{x_m,\alpha_m,s} \cdot \omega(\sigma_m, s)}{\omega(ONE, s)} \cdot \prod_{j=m+1}^{n} p^{x_j,\alpha_j,s}$$

which simplifies to

$$\left(\prod_{i=i}^{n} p^{x_i,\alpha_i,s} \right) / \omega(\sigma_1, s),$$

and further, due to Definition 5.2 and Lemma 5.1, rewrites to

$$\pi(\alpha, s) / \sum_{\beta \in f_s} \pi(\beta, s),$$

which, by Definition 5.3, is exactly the branching probabilities of the legal input vector α. ■

Since constrained probabilities are completely determined by the constraint, the current state, and the input probabilities, a direct result of the above theorem is the *p-tree* algorithm's independence from the BDD variable ordering, which is a nice property for techniques based on BDDs.

Corollary 5.2 The probability of generating an input vector using the p-tree algorithm is independent of the variable ordering of the constraint BDD.

Finally, we show that *p-tree* holds another property that correlates the input and the branching probabilities.

Lemma 5.2 Using the p-tree algorithm, the probability of generating an input vector in which an input variable x_k equals 1 (resp. 0) monotonically increases as p^{x_k} (resp. $p^{\overline{x_k}}$) increases.

Proof: There are two cases: (1) If x_k is not visited in *Walk*, then it is assigned according to its input probabilities, to which the probability of the vector is proportional, hence the lemma is true; (2) If x_k is visited in *Walk*, let f be the constraint and s the state, then the probability of the vector, say $\alpha = \alpha_1 \cdots \alpha_n$ where α_k is the value of x_k, is

$$\frac{\prod_{i=0}^{n}(\alpha_i \cdot p^{x_i}(s) + (1 \Leftrightarrow \alpha_i) \cdot p^{\overline{x_i}}(s))}{\omega(r(f), s)}.$$

Because this probability does not depend on variable ordering (Corollary 5.2), we choose x_k to be the variable associated with the root node. Then the probability becomes

$$\frac{\prod_{i=1}^{n}(\alpha_i \cdot p^{x_i}(s) + (1 \Leftrightarrow \alpha_i) \cdot p^{\overline{x_i}}(s))}{(p^{x_k}(s) \cdot \omega(t(r(f)), s) + p^{\overline{x_i}}(s) \cdot \omega(e(r(f)), s))}.$$

Let q denote the product of $(\alpha_i \cdot p^{x_i}(s) + (1 \Leftrightarrow \alpha_i) \cdot p^{\overline{x_i}}(s))$ for all $i \neq k$, w_t denote $\omega(t(r(f), s)$, and w_e denote $\omega(t(r(f), s)$. Note q, w_t and w_e are independent of the input probabilities of x_k. Let $\alpha_k = 1$, the above formula rewrites to

$$(q \cdot p^{x_k}(s)) \ / \ (p^{x_k}(s) \cdot w_t + p^{\overline{x_k}}(s) \cdot w_e)$$

$$q \ / \ (w_t + (p^{\overline{x_k}}(s)/p^{x_k}(s)) \cdot w_e)$$

$$q \ / \ (w_t + (1/p^{x_k}(s) \ \Leftrightarrow 1) \cdot w_e)$$

therefore the probability of α monotonically increases in p^{x_k}. The case $\alpha_k = 0$ is analogous. ■

5.4.4 An Example of the *p-tree* Algorithm

We reuse the example from Section 5.3.5 by making a slight modification to demonstrate the effect of state in constraints: we qualify the constraint with the state $!reset$ (read *not reset*) as in the following:

```
$constraint( !reset→
        (cmd[3:0] == 4'b1000) ||
        (cmd[3:0] == 4'b0100) ||
        (cmd[3:0] == 4'b0010) ||
        (cmd[3:0] == 4'b0001) );
```

The resulting constraint BDD is shown in Figure5.3: solid arcs represent *then* branches, dashed arcs represent *else* branches, and the variable ordering is shown to

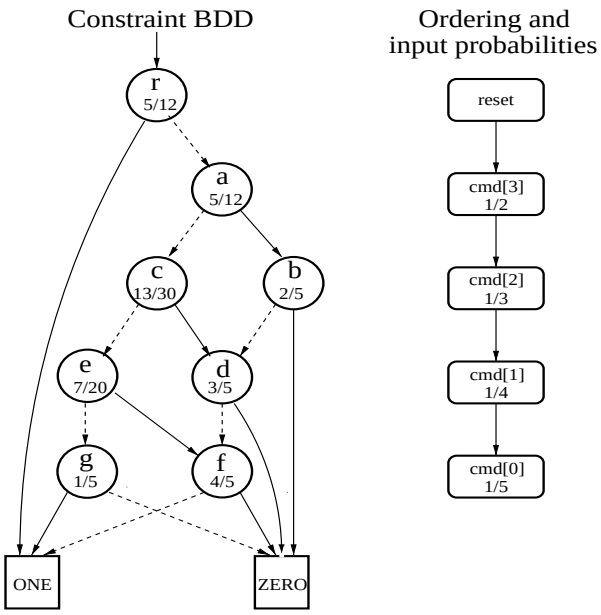


Figure 5.3: A constraint BDD labeled with node weight

the right of the BDD. To aid the discussion, we use $x_0, \ldots, x_3$ to denote the inputs cmd[0], . . . ,cmd[3], respectively.

As described in Section 5.4.1, for a given state, only a subgraph in the BDD involves in the computation: for the state "reset=1", the subgraph contains only the *ONE* node, meaning that all inputs are possible, therefore the constrained probability of a vector $\alpha = \alpha_0 \cdots \alpha_3$ is simply the product $\prod_{i=0}^{n}(\alpha_i \cdot p^{x_i} + (1 \Leftrightarrow \alpha_i) \cdot p^{\overline{x_i}})$; for the state "reset=0", the subgraph contains nodes a through g, which is the BDD corresponding to the original constraint used in Section 5.3.5. We will work on the latter case, and show how *p-tree* implicitly computes the probabilities of legal inputs, which should match the constrained probabilities obtained in Section 5.3.5,

1/2 cmd[3]	1/3 cmd[2]	1/4 cmd[1]	1/5 cmd[0]	constrained probabilities
0	0	0	1	3/25
0	0	1	0	4/25
0	1	0	0	6/25
1	0	0	0	12/25

Table 5.3: Example: constrained probabilities (*reset=0*)

given again in Table 5.3.

First, *Weight* is applied to the root node r, which recurs in a depth-first order in the subgraph rooted at a. We illustrate the calculation by considering nodes f and e. For brevity, we denote the state "reset=0" by "!r".

$$\begin{aligned}\omega(f,!r) &= (1 \Leftrightarrow p^{x_0}(!r)) \cdot \omega(ONE,!r) + p^{x_0}(!r) \cdot \omega(ZERO,!r) \\ &= 4/5 \cdot 1 + 1/5 \cdot 0 \\ &= 4/5\end{aligned}$$

Similarly, we have $\omega(g,!r) = 1/5$. Then,

$$\begin{aligned}\omega(e,!r) &= p^{x_1}(!r) \cdot \omega(f,!r) + (1 \Leftrightarrow p^{x_1}(!r)) \cdot \omega(g,!r) \\ &= 1/4 \cdot 4/5 + 3/4 \cdot 1/5 \\ &= 7/20\end{aligned}$$

The weights of r and a through g are shown in Figure 5.3. The probabilities of legal input vectors are then obtained according to procedure *Walk*. For example,

vector 1000, which corresponds to the path r, a, b, d, f, ONE, has the probability:

$$\frac{\frac{1}{2}\cdot\frac{2}{5}}{\frac{5}{12}}\cdot\frac{(1\Leftrightarrow\frac{1}{3})\cdot\frac{3}{5}}{\frac{2}{5}}\cdot\frac{(1\Leftrightarrow\frac{1}{4})\cdot\frac{4}{5}}{\frac{3}{5}}\cdot\frac{(1\Leftrightarrow\frac{1}{5})\cdot 1}{\frac{4}{5}}=\frac{12}{25}.$$

Note it matches the constrained probability of the same vector, shown in Table 5.3.

5.5 Implementation Issues

In this section, we discuss issues regarding the efficiency of the algorithm, in particular, the size of the constraint BDDs. We also describe an implementation of the algorithm in a commercial simulator.

5.5.1 Variable Ordering

As we have seen in the previous sections, the *p-tree* algorithm takes time and space linear to the size of the constraint BDD, therefore, minimizing the BDD size is of a great interest to us.

We can not do without the mentioning of BDD variable ordering, which we have shown in the introduction chapter can have a dramatic effect on BDD size, sometimes making an exponential-vs-linear complexity difference. Similar cases are abundant in hardware constraints, for example, in

$constraint((st != 2'b11) ? (attr == PREV_attr) : 1);

if all the variables in "attr" occur before the ones in "PREV_attr", the equation would give a BDD with a size exponential to the width of "attr", whereas an interleaved ordering of the two would result in a linear size.

We developed heuristics which identify constructs such as in the above to obtain a “good” initial order. In many cases we observed that sticking to this initial order, rather than dynamically reordering the variables, renders faster and leaner execution of our algorithm.

5.5.2 Constraint Partitioning

When there is a large number of constraints, forming the conjunction BDD can be very expensive, for two reasons: (1) the computation blows up because of large intermediate BDDs; (2) the large conjunction BDD can slow down vector generation. SymGen partitions the constraint BDDs into groups with disjoint input variable support using the following procedure:

1. for each input variable, create a group
2. for each constraint depending on a variable, add the constraint to the variable’s group
3. merge all groups that share a common constraint until each constraint appears in at most one group
4. for each constraint that is not in any group yet, add it to a new group. Observe that these constraints should depend only on state variables

The *p-tree* algorithm can then be applied to each group separately. The soundness of *p-tree* under constraint partitioning is guaranteed by the following theorem:

Theorem 5.4 Let C be a set of constraints, $C_1, \ldots, C_n$ the disjoint-input-support partition of C, and $\alpha^1, \ldots, \alpha^n$ be the corresponding partial vectors generated by applying *p-tree* to the $C_1, \ldots, C_n$. Let α denote the concatenation $\alpha^1.\alpha^2 \ldots \alpha^n$. Then under any state, the probability of generating α^1, α^2,. . ., and α^n is equal to the constrained probability of generating α from C.

Proof: Let s be the state. Let f be the conjunction of all constraints, and $f^1, \ldots, f^n$ the conjunctions of constraints in the groups. Hence, $f = \bigwedge_{i=1}^{n} f^i$. First, we prove

$$\prod_{i=1}^{n} \sum_{\beta \in f_s^i} \pi(\beta, s) \quad = \quad \sum_{\beta \in f_s} \pi(\beta, s). \tag{5.6}$$

We know

$$\begin{aligned} \sum_{\beta \in f_s^1} \pi(\beta, s) \cdot \sum_{\gamma \in f_s^2} \pi(\gamma, s) &= \sum_{\beta \in f_s^1} [\pi(\beta, s) \cdot \sum_{\gamma \in f_s^2} \pi(\gamma, s)] \\ &= \sum_{\beta \in f_s^1} \sum_{\gamma \in f_s^2} [\pi(\beta, s) \cdot \pi(\gamma, s)] \\ &= \sum_{\beta \in f_s^1} \sum_{\gamma \in f_s^2} \pi(\beta.\gamma, s) \end{aligned}$$

since f^i and f^j where $i \neq j$ have disjoint input supports, thus $\beta \in f_s^i \wedge \gamma \in f_s^j$ iff $\beta.\gamma \in f_s^i \wedge f_s^j$, the above formulas can be further reduced to

$$\sum_{\beta.\gamma \in (f_s^i \wedge f_s^j)} \pi(\beta.\gamma, s) \quad = \quad \sum_{\delta \in (f^i \wedge f^j)s} \pi(\delta, s).$$

Therefore, we have proved Equation 5.6 for the case $n = 2$. Since $f^1 \wedge f^2$ and f^i for $i > 2$ again have disjoint input supports, by induction, Equation 5.6 holds for all n.

Equation 5.6 implies that $\sum_{\beta \in f_s} \pi(\beta, s) = 0$ iff there exists i, $\sum_{\beta \in f_s^i} \pi(\beta, s) = 0$, therefore, an illegal state in f is also an illegal state in the partition and vice versa. So in both cases, for the same set of illegal states, the probabilities for all (illegal) vectors are 0.

Now consider legal vectors. From Definition 5.2, we have

$$\prod_{i=1}^{n} \pi(\alpha^i, s) = \pi(\alpha, s).$$

So when s is a legal state, dividing each side of the above equation by the corresponding side of Equation 5.6 gives:

$$\frac{\prod_{i=1}^{n} \pi(\alpha^i, s)}{\prod_{i=1}^{n} \sum_{\beta \in f_s^i} \pi(\beta, s)} = \frac{\pi(\alpha, s)}{\sum_{\beta \in f_s} \pi(\beta, s)}$$

which is

$$\prod_{i=1}^{n} \frac{\pi(\alpha^i, s)}{\sum_{\beta \in f_s^i} \pi(\beta, s)} = \frac{\pi(\alpha, s)}{\sum_{\beta \in f_s} \pi(\beta, s)}$$

where the left-hand side is the product of probabilities of generating $\alpha^1, \ldots, \alpha_n$ from $C_1, \ldots, C_n$, respectively, and the right-hand side is the probability of generating α from C. Hence the theorem holds also for legal vectors. ■

5.5.3 The Overall Flow

The *p-tree* algorithm is implemented as a library function interfaced to the Verilog-XL simulator [27]. After an initialization sequence, the simulator calls *p-tree* at every clock cycle when it needs a new input vector, for example, right before the rising edge of the clock, and *p-tree* performs a sequence of tasks, as illustrated in Figure 5.4.

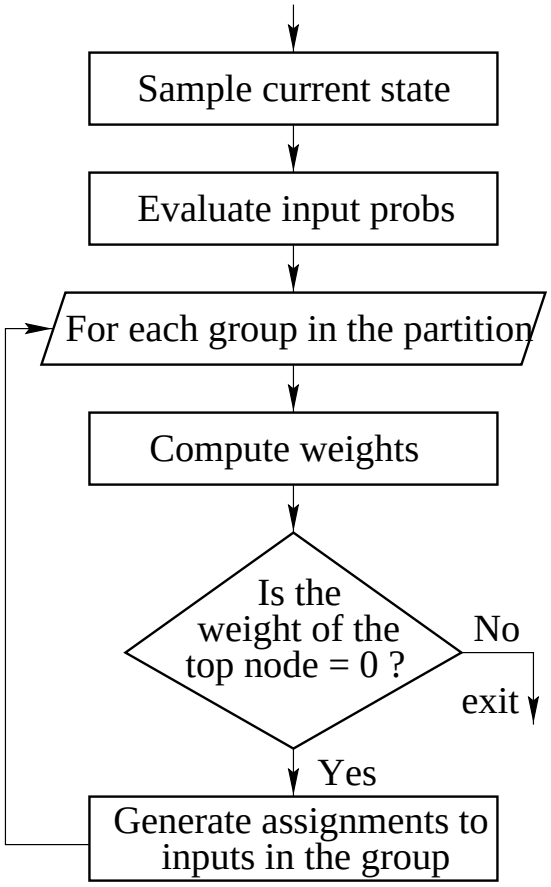


Figure 5.4: The *p-tree* algorithm

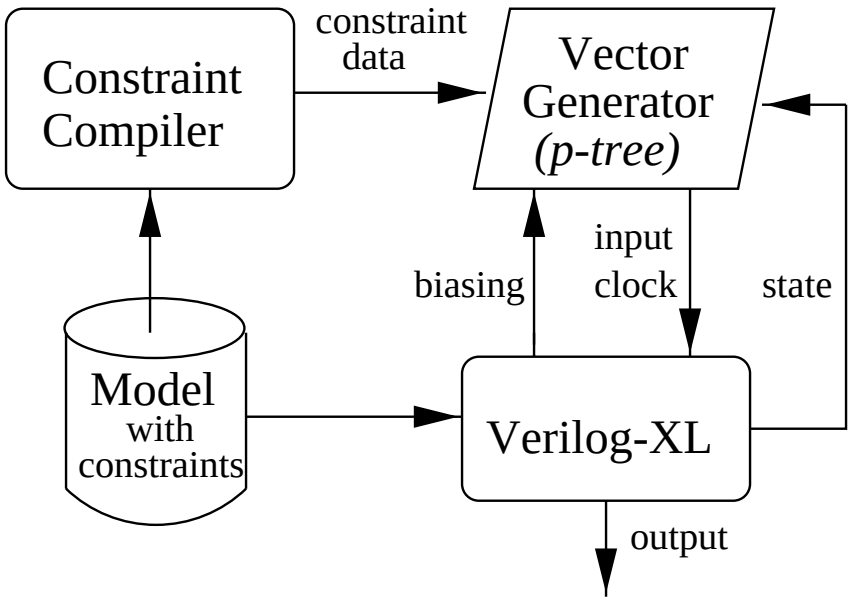


Figure 5.5: SymGen flow chart

Note that procedure *Weight* is not always necessary in every simulation cycle. In particular, if all state variables occur before the input variables in the constraint BDD and the input probabilities are constants, then it suffices to compute the weights only once at the beginning of the simulation.

Figure 5.5 provides a high level view of SymGen. The *Constraint Compiler* reads a Verilog [27] model annotated with constraints, and then extracts and compiles the constraints into BDDs which will be used by the *Vector Generator*. During simulation, the Verilog-XL simulator evaluates input probabilities under current state at each clock cycle. The *Vector Generator* then samples the current state and generates an input vector based on the constraint BDDs.

name	*total vars*	*vars in cons*	*num of cons*
block1	76	26	13
block2	178	59	10
block3	1437	153	11
block4	446	175	33
block5	407	297	34

Table 5.4: Statistics of designs

5.6 Results

We present experimental results on industrial designs which SymGen has already been applied to. These designs came with environment constraints developed by the engineers. We report results of experiments on building constraint BDDs for five designs in Section 5.6.1. In Section 5.6.2, we give a case study on one of the designs.

The underlying BDD package is CUDD [45], developed in the University of Colorado. All experiments were conducted on a 233 MHz UltraSPARC-60 machine with 512 MB main memory.

5.6.1 Constraint BDDs

Table 5.4 reports the complexity of our benchmark designs. The BDD variables reported include inputs and latches. Columns 2 through 4 give the total number of BDD variables, the number of variables used in the constraints, and the number of constraints, respectively. Note that *block5* has about 300 variables in its constraints.

example	*time (sec)*	*peak nodes*	*mem (MB)*	*result nodes*
block1	0.0	6312	0.5	54
block2	5.0	5110	3.6	119
block3	26.0	6132	9.3	774
block4	885.5	303534	35.5	110858
block5	727.7	181243	25.0	82405

Table 5.5: Building the constraint BDD without partitioning

The results of building constraint BDDs are shown in Tables 5.5 and 5.6, respectively for cases without and with constraint partitioning. A sifting-based variable reordering [45] was enabled in all experiments with the same setting.

Without partitioning, *block4* and *block5* each have close to 100k nodes in the result BDD, as shown in Table 5.5. The number of peak intermediate BDD nodes grows over 300k for *block4*.

Table 5.6 shows the effectiveness of using partitioning. Although the numbers of constraints vary from 10 to 34 among designs, the average constraints per partition is about 3, which is fairly small. As a result, many BDD conjunction operations are avoided, and the total BDD size is reduced. Partitioning gives *block1*, *2* and *3* some modest improvement, but reduces both time and space complexity for *block5* and *6* dramatically. The complexity of the designs (over 1000 variables) and constraints (close to 300 variables), together with the size of the constraint BDDs (less than 2000 nodes) demonstrate that our technique is feasible for medium or even large designs.

example	*time (sec)*	*#pnodes*	*mem(MB)*	*#rnodes*	*#cons*	*#parts*
block1	0.0	1022	0.5	43	13	5
block2	4.0	5110	3.4	103	10	7
block3	20.3	6132	8.4	609	11	9
block4	38.0	13286	6.3	1595	33	10
block5	33.4	22484	7.4	1962	34	9

Table 5.6: Building the constraint BDD with partitioning

5.6.2 A Case Study

SymGen was used to construct verification environments for several designs blocks. In the following we present its application to a $PowerPC^{TM}$ slave block - namely the *block5* in the previous subsection.

The first task is to develop the environment constraints according to a specification for the block (including its interface) written in English. Because of the sequential nature of the specification, we needed to introduce auxiliary variables to remember previous input or state value, and construct abstract state machines of the design. The expressiveness of SymGen constraints is further strengthened.

Conflicts between constraints occur often because of their complexity and the number of constraints involved. In such cases, methods such as prioritization are used to resolve the conflicts. A prioritized constraint looks like the following:

```
$constraint( st != 2'b11 ? a == PREV_a :
                (!b & !c) ? u == PREV_u :
                (!b & c & !(e == 0 || e == 4)) ?
```

```
        ((f < 7) & (f > 0)) :
(t == 0 || t == 4 || t == 7) ?
        !(z == 6 || z == 5) :
1 );
```

Note that *PREV_a* and *PREV_u* are auxiliary variables holding the values of *a* and *u* on the previous clock cycle. Because of the comparator "*a == PREV_a*", an interleaved ordering of bits of *a* and *PREV_a* should be used.

The constraints for this block were written in about 2 person-days. The end result was a concise specification of the environment in a 200-line Verilog file, including 34 constraints, the auxiliary variables and abstract state machines. The benefit of constraining can not be overemphasized. Unconstrained random simulations generally produce false negative results. We also noticed in unconstrained simulations that the X value was constantly generated on tri-state buses, indicating bus contentions, which made the simulations meaningless.

Developing input biasing was mostly straightforward. For instance, we wanted to limit the frequency of external errors when testing the essential functionality of a design. This was expressed as

```
$setprob1(error, 0.2);
```

There were also cases where we needed to consider the more involved dynamic biasing. For example, even after we give static biasing to over a dozen critical input signals, three major state machines stayed mostly idle through trial

simulations. After studying the design for about an hour (it should take the original designer much less time), we were able to find a set of dynamic input probabilities that stimulated many more events in active states, such as *read* and *write*. The following biasing instructs SymGen to give much higher input probability to u_i in state $IDLE$, which triggers a transition out of $IDLE$:

$setprob1($u_i$, addr_state == IDLE ? 0.9, 0.5);

We wrote a few dynamic biasing commands for each of the state machines. Note that we did not use constraints to express the condition, because "leaving" the idle state is not a mandatory action, it just serves our test purpose. Also, using input biasing instead of constraints can avoid potential constraint contradictions, because the constraints take priority over input probabilities.

Of course, in general, state transitions are controlled by both state and inputs, and justifying an enable condition can be a very hard problem. However, with the help of automatic tools such as SymGen, this problem can be lessened. Table 5.7 shows the effect of biasing on the number of active states (of three major state machines) visited during simulations of 1000 cycles each. The dynamically biased simulation increased the coverage 130 times over that of the unbiased simulation.

Simulations usually run with monitoring processes, or dump the results to log files for post-processing. Table 5.8 summarizes the run time overhead of SymGen on *block5*, with partitioned constraint BDDs and dynamic biasing. All simulations ran for 10000 cycles each. Row 2 to 5 respectively represent the simulations with pure random generation, stand alone SymGen, SymGen with Verilog dump,

biasing	*idle states*	*active states*	*total (sec)*
none	2977	14	6.8
static	2179	811	6.2
dynamic	1073	1918	6.3

Table 5.7: Result of biasing

$setting$	*overall (sec)*	*SymGen overhead*
random	44.9	–
SymGen	48.2	21.3%
w. dump	63.6	16.0%
w. monitor	635.6	1.7%

Table 5.8: Overhead of SymGen

and with property monitoring. It can be seen that the overhead of SymGen is fairly low.

In this specific example, SymGen, together with a simulation monitoring tool, discovered 30 design bugs, which basically fell into two categories:

1. Bugs caught because the design entered a state where there was no legal input possible. This implies the design has violated the constraints.

2. Bugs caught because the design entered an illegal state. This is usually manifested as property violations.

5.7 Summary

We have described an automated simulation-vector generation method. Constraints are used to generate legal vectors which are influenced by input biasing. Both constraints and biasing can depend on the state of the design, thus providing robust environment modeling capability. The implementation, SymGen, is based on an efficient symbolic algorithm which does not backtrack in solving the constraints. The effectiveness of SymGen is demonstrated in its application to commercial design verification.

Before the conclusion of this chapter, we would like to briefly mention our experiment in extending the biasing method in SymGen, which is based upon the probabilities of individual inputs. We looked at a more general biasing given by the *event probabilities* of the form $Pr(Event) = \pi$, where $Event$ is a characteristic function defining a set, for example, $In\_a == In\_b$. Unfortunately, Koller and Megiddo [68] showed that the problem of deciding whether there exists any distribution satisfying a set of event probabilities is NP-hard. We described in [113] a decision procedure using linear programming (LP), which gives a distribution, if it exists, with minimum number of positive LP variables. However, the number of variables in the worst case can be exponential to the number of event probabilities. In [114] we reported a method of directed simulation using event probabilities without actually solving them. In that case, we extracted a finite state machine (FSM) for each desired property, and treated each state transition as an event. The probability of each transition is dynamically assigned depending on the current state, i.e., the transitions that are most likely to lead to the "bad" state in the FSM are given

high probabilities. Experimental results showed that when a transition is possible, SymGen with the new biasing method has a very high probability of generating a vector that enables the transition.

Chapter 6

Constraint Diagnosis

6.1 Introduction

Any constraint-based programming will have to deal with the diagnosis of conflicting constraints, which has been well studied in many research disciplines, for example, in artificial intelligence [36]. In this chapter, we will focus on a problem unique to our application, that is, constraint conflicts conditioned upon the state of the design.

Recall in Section 5.4.1, the weight of the root node of the constraint BDD being zero indicates the design has entered an illegal state, i.e., one for which there is no input assignment satisfying the constraint. In constraint diagnosis, we refer to such states as the dead-end states (DES), since the simulation cannot proceed upon entering them. DES is a practical problem in constraint writing and has implication on the behavior of the design, thus meriting a closer look. Our constraint diagnosis will be focusing on DES.

A free-input design can be modeled as a Kripke structure wherein every state has a next state. Environment constraints modify design behavior by truncating transitions prohibited by the constraints, giving rise to states (DESs) that do not have any next states. Although in our definition DES is a property of and can be

derived statically from a constraint, not all DESs automatically warrant a debugging since some of them may be unreachable. Computation of the exact set of reachable states, however, is intractable for most designs of interest; therefore, the reachability information in constraint diagnosis is usually given as an approximation, e.g., as assertions or invariants on important control signals, or by approximate reachability analysis [86]. Should DESs ruled out by reachability assertions be encountered in simulation, we have an indication that the design is malfunctioning under the constraints.

Ideally, environment constraints should not produce any DESs, or all DESs should be covered by "known" unreachable states. More often than not, however, this is not the case and the designer either has to modify the constraints or be prepared to catch and debug DESs on-the-fly during simulation.

6.2 Static Analysis of DESs

Let $f(X, Y)$ be a constraint over the set of input variables X and state variables Y, then the set of DESs of f, denoted by its characteristic function $D(f)$, is computed as:

$$D(f) = \forall_X(\overline{f}) \tag{6.1}$$

which is a direct derivation of the definition of DES. It follows that DESs of one constraint are also DESs of the conjunction of all the constraints.

Lemma 6.1 Let $f_1, \ldots, f_n$ be the constraints, then for all $1 \leq i \leq n$

$$D(f_i) \leq D(\bigwedge_{j=1}^{n} f_j).$$

Proof: $D(\bigwedge_{j=1}^{n} f_j) = \forall_X(\overline{f_1} + \ldots + \overline{f_n}) \geq \forall_X(\overline{f_i}), 1 \leq i \leq n.$ ■

Also, in a disjoint-input-support partition of constraints, the DESs of all constraint is the union of DESs of all the components in the partition.

Lemma 6.2 Let $f_1, \ldots, f_n$ be constraints of pair-wise disjoint input supports, then

$$D(\bigwedge_{j=1}^{n} f_j) = \bigvee_{i=1}^{n} D(f_i)$$

Proof: Let $X_1, \ldots, X_n$ be the partition of X corresponding to $f_1, \ldots, f_n$, then

$$\begin{aligned} D(\bigwedge_{j=1}^{n} f_j) &= \forall_X(\overline{f_1} + \ldots + \overline{f_n}) \\ &= \forall_{X_1}(\overline{f_1}) + \ldots + \forall_{X_n}(\overline{f_n}) \\ &= \forall_X(\overline{f_1}) + \ldots + \forall_X(\overline{f_n}) \\ &= D(f_1) + \ldots + D(f_n) \end{aligned}$$

■

Suppose we a have a set of assertions on reachable states and we consider only the DESs that are not covered by these assertions. From Lemma 6.1 and 6.2, if we "eliminate" DESs from each constraint, and from each component of the partition, then the resulting constraints are DES-free. Since eliminating DESs is

essentially the problem of resolving constraint conflicts, we describe a straightforward approach that does this by relaxing the constraints, as stated in the following lemma.

Lemma 6.3 Let f be a constraint, d a subset of $D(f)$, and X the set of input variables. Then

1. $D(f+c) \leq D(f)$, and
2. $D(f) \Leftrightarrow D(f+c) = d$ for all c such that $\exists_X(c) = d$

Proof: Since

$$\begin{aligned} D(f+c) &= \forall_X(\overline{f+c}) \\ &= \forall_X(\overline{f}) \cdot \forall_X(\overline{c}) \\ &= D(f) \cdot \forall_X(\overline{c}) \end{aligned}$$

therefore, $D(f+c) \leq D(f)$, and

$$\begin{aligned} D(f) \Leftrightarrow D(f+c) &= D(f) \cdot \overline{D(f) \cdot \forall_X(\overline{c})} \\ &= D(f) \cdot \left(\overline{D(f)} + \exists_X(c)\right) \\ &= D(f) \cdot d \\ &= d \end{aligned}$$

■

The above lemma shows that a subset d of the DESs of f can be removed by relaxing f to $f + c$, where c allows some input assignments for every state in d. Further, the relaxation does not introduce new DESs.

6.3 Dynamic Methods

In the static analysis, DESs that are not covered by the reachability assertions are dealt with by relaxing the constraints. However, eliminating DESs, in whatever manner, will have to maintain the validity of the constraints being a correct model of the environment; when the constraints or the design are complex enough, eliminating all DESs can be a insurmountable task. Thus in reality, many DESs are left unresolved in the hope that they are not reachable by the design. This assumption, as well as the reachability assertions, however, needs to be validated in the execution of the design.

A naive approach is the reachability analysis by model checking on the design whose transition relation and initial states are modified by the constraints. However, the approach can be computationally expensive.

A reasonable approach is to turn to the static analysis and DESs removal, and then detect the remaining DESs on-the-fly during simulation. As shown in Theorem 5.1, Chapter 5, the current state is a DES iff the weight of the root node of the constraint BDD is zero. Because of constraint partitioning, this detection conveniently localizes the problem to the current group of constraints that is being processed by the *p-tree* algorithm. Furthermore, a minimal conflicting set within the group can be obtained, in the usual way, by forming the conjunction of constraints in

subsets of the group of increasing cardinality, until an empty conjunction is found. Of course, the constraints are first simplified by substituting in the DES.

There are two scenarios for the treatment of the conflicting constraints detected this way.

1. If the DES is allowed by the reachability assertions, then the constraints need to be relaxed.

2. If the DES is precluded by the reachability assertions, then either the design contains a bug, or the constraints are too loose to allow the design to enter this DES, in which case, the constraints need to be "tightened".

Whereas the relaxation can be done relatively simply as given in Lemma 6.3, tightening constraints to avoid a DES is much more involved and heuristic in nature because the fix must eliminate all transitions that lead to the DES while not overdoing it to incur new DESs. The topic merits separate research and is outside of the scope of this dissertation.

Chapter 7

Simplification of Constraint Solving

7.1 Introduction

The complexity of constraint solving can grow arbitrarily due to the highly combinatorial nature of the task. Therefore, even implicit representations (in our case, BDDs) will have to face the efficiency problem. An immediate step toward dealing with the problem is disjoint-support partitioning of the constraints, as described in Chapter 5. In this chapter, we present a technique called *hold-constraint extraction* which is aimed at simplifying the constraints and refining the aforesaid partition.

The technique is based on the observation that variables in hardware constraints are not homogeneous: state variables, unlike the inputs, are bounded by the design, and often, some inputs can be fully specified under certain state valuations, regardless the values of other inputs. We refer to the kind of constraint wherein the inputs only dependent upon the state variables as the *hold-constraint*, and the inference thereof from a constraint the *hold-constraint extraction*. The dichotomy comes from a more specific example of the mentioned dependency, which occurs frequently in writing constraints for hardware designs — under certain condition, an input variable maintains its value from the previous clock cycle, or is simply fixed

to a constant. A hold-constraint is either instantly solved, assigning constants to its input variables, or discharged as a tautology. In addition, the disjoint-input-support partitioning can be further refined due to two facts about the hold-constraints: (1) they do not need to be conjoined with any other constraints while being solved; (2) they can be used to simplify the original constraints, and the results often contain fewer input variables. Experiments of applying this simplification to SymGen on several commercial designs demonstrated significant reduction in the time and space needed for constructing the conjunction BDDs, and the time spent in vector generation during simulation.

It is worth mentioning that hold-constraints by themselves contribute to the speedup of vector generation since they can assign constant values to input variables once the state evaluation is known. In this regard, the technique is similar to the derivation of unit clauses in SAT problems [35]. Propagation of constants instantly simplifies the problem at hand.

We defer a detailed discussion of related works to Section 7.6, in which we compare hold-constraint extraction to a special type of functional decomposition. It is shown that our technique subsumes existing decomposition methods that are potentially useful in extracting hold-constraints.

The rest of the chapter is structured as follows: Section 7.2 gives the preliminaries on hold-constraint extraction. In Section 7.3 we briefly describe a syntactical extraction algorithm. A procedure of complete functional extraction of hold-constraints and how they are used to simplify the original constraints are given in Section 7.4. Related works are discussed in Section 7.6. We report experimental

results in Section 7.7 and summarize in Section 7.8.

7.2 Preliminaries

Definition 7.1 A constraint is a Boolean function $\{0,1\}^{m+n} \mapsto \{0,1\}$ defined over input variables $X = \{x_1, \ldots, x_n\}$ and state variables $Y = \{y_1, \ldots, y_m\}$.

As described in Chapter 5, the onset of a constraint represents the legal state-dependent input space regarding this constraint. The overall legal input space is the intersection of the onsets of all the constraints. In the sequel, we frequently use f, g, h, k and e as function symbols, y as a state variable, s_i, s_j as states, i.e., minterms of state variables, S as a set of states, and x, v as input variables.

Definition 7.2 An input variable x in the support of f is *positively* (resp. *negatively*) *bounded* with respect to a set of states S if in all minterms in $f^{on} \cdot S$, x evaluates to 1 (resp. 0).

More intuitively, x is *positively bounded* with respect to S if $f \rightarrow (S \rightarrow (x = 1))$, and x is *negatively bounded* with respect to S if $f \rightarrow (S \rightarrow (x = 0))$.

Definition 7.3 A *hold-constraint* on input variable x is a constraint $e(x, Y)$ in which x is positively or negatively bounded with respect to a nonempty set of states.

All hold-constraints can be written in the following *normal* form:

$$k \rightarrow (x = g) \tag{7.1}$$

where x is the input variable, and k, g are Boolean functions depending only on Y (the state variables), which we call the *condition* and *assignment* respectively. Note both k and g can be constants with the exception that k can not be 0.

We can infer a hold-constraint $k \rightarrow (x = g)$ from f iff the following *implication requirement* is met:

$$f \rightarrow (k \rightarrow (x = g)) \tag{7.2}$$

However, it immediately comes to mind that an arbitrary hold-constraint can meet the requirement as long as the condition k does not overlap f. For a meaningful inference, we must also enforce the *nonvacuousness requirement*:

$$f \cdot k \neq 0 \tag{7.3}$$

Definition 7.4 A hold-constraint $k \rightarrow (x = g)$ is *extractable* from constraint f if the two satisfy the implication and nonvacuousness requirements.

In the coming sections, we present how hold-constraints can be extracted syntactically and functionally.

7.3 Syntactical Extraction

Syntactical hold-constraint extraction consists of two phases — the decomposition phase in which the constraints are conjunctively decomposed, and the matching phase in which each of the conjuncts is checked to see if it transforms to a hold-constraint in the normal form.

The syntactical extraction algorithm proceeds as follows:

1. Conjunctively decompose the constraint according to the decomposition rules
2. For each conjunct f, begin
3. If f does not match a pattern that is transformable to a disjunction $k + h$, goto step 2
4. If k depends on input variables, swap k and h
5. If k depends on input variables, goto step 2
6. If h matches a pattern that is transformable to $x = g$, where x is an input variable and g does not depend on input variables, then add $k \rightarrow (x = g)$ to the hold-constraint set, and remove f from the set of conjuncts
7. End

The above extraction procedure satisfies the implication and nonvacuousness requirements. Its time and space complexities are both linear to the size of the constraint formulas. However, it is incomplete due to the limitations imposed by the finite sets of rules and patterns.

7.4 Functional Extraction

7.4.1 Condition and Extraction

We begin the description of a complete functional extraction with the recognition of *prime* hold-constraints.

Definition 7.5 A hold-constraint $k \rightarrow (x = g)$ is said to be *prime* if g is a constant, and the onset of k is a singleton, i.e., contains exactly one state.

The following lemma follows from Definitions 7.2 and 7.5.

Lemma 7.1 For any constraint f, for every input variable x and state s_i such that x is bounded in f with respect to $\{s_i\}$, the prime hold-constraint below is *extractable* from f

$$s_i \rightarrow (x = b_i)$$

where b_i is 1 if x is positively bounded, and 0 otherwise.

The theorem below indicates that the conjunction of the prime hold-constraints obtained as in Lemma 7.1 on an input variable is a hold-constraint on the same variable, and the converse is also true.

Theorem 7.1 The conjunction of a set of prime hold-constraints on x with mutually exclusive conditions is a hold-constraint, and vice versa. That is,

$$\bigwedge_{i=1}^{l} (s_i \rightarrow (x = b_i)) \Leftrightarrow k \rightarrow (x = g) \tag{7.4}$$

where $s_i \wedge s_j = 0$ for $i \neq j$, $b_i \in \{0, 1\}$. Furthermore, k and g can be derived from the prime hold-constraints as

$$k = \bigvee_{i=1}^{l} s_i,\ g \in [g^{on}, \overline{g^{off}}],\ g^{on} = \bigvee_{i=1}^{l} (b_i \cdot s_i),\ g^{off} = \bigvee_{i=1}^{l} (\overline{b_i} \cdot s_i), \tag{7.5}$$

and the prime hold-constraints can be derived from k and g as in the set

$$\{(s_i, b_i) \mid 1 \leq i \leq |\, k \,|,\ s_i \in k,\ b_i = g_{s_i}\}. \tag{7.6}$$

where $|k|$ is the number of minterms in the onset of k.

Proof: We prove the theorem by showing that, for both derivations, the valuations of the two sides of the equivalence in (7.4) under any state are equal.

Denote the state by s_j. If $s_j \notin k$, the theorem holds since both sides of (7.4) evaluate to 1 trivially, for both derivations. Conversely, if $s_j \in k$, we have, for both derivations,

$$\left(\bigwedge_{i=1}^{l} (s_i \rightarrow (x = b_i))\right)_{s_j} = (x = b_j), \tag{7.7}$$

$$(k \rightarrow (x = g))_{s_j} = (x = g_{s_j}). \tag{7.8}$$

Now for the derivation in (7.5), if $b_j = 1$, then $s_j \in g^{on}$ and $g_{s_i} = 1$, therefore both (7.7) and (7.8) evaluate to $x = 1$; similarly, if $b_j = 0$, then $s_j \in g^{off}$ and $g_{s_i} = 0$, therefore both (7.7) and (7.8) evaluate to $x = 0$.

For the derivation in (7.6), if $g_{s_j} = 1$, then $b_j = 1$, therefore both (7.7) and (7.8) evaluate to $x = 1$; similarly, if $g_{s_j} = 0$, then $b_j = 0$, therefore both (7.7) and (7.8) evaluate to $x = 0$. ■

Because f implies, and intersects with, the condition of every prime hold-constraint on x as obtained in Lemma 7.1, it must also imply the conjunction of the said prime hold-constraints and intersect with the union of the said conditions. Therefore, if all of the prime hold-constraints are extractable from f, the hold-constraint conjunctively constructed in Theorem 7.1 is also extractable from f.

Now we derive the procedure that actually "computes" the construction in Theorem 7.1. By abuse of notation, we denote the *set difference* operator by "$\Leftrightarrow$",

and the set of input variables $X \Leftrightarrow \{x\}$ by x'. The following is true for any constraint f:

1. $(\exists_{x'} f)_x \Leftrightarrow (\exists_{x'} f)_{\overline{x}}$ is the set of all the states with respect to which x is positively bounded

2. $(\exists_{x'} f)_{\overline{x}} \Leftrightarrow (\exists_{x'} f)_x$ is the set of all the states with respect to which x is negatively bounded

The union of the above two disjoint state sets is the Boolean differential denoted by $\partial(\exists_{x'} f)/\partial x$ (ref. Section 2.1, Chapter 2). The conjunction of the prime hold-constraints conditioned on this set is the *complete* hold-constraint on x, since it includes all the states for which x is bounded. The derivation of such a hold-constraint is formalized in the theorem below, which follows naturally from Theorem 7.1 and the above analysis.

Theorem 7.2 The complete hold-constraint of f on x is

$$k \rightarrow x = [g^{on}, \overline{g^{off}}]$$

where

$$k = \frac{\partial(\exists_{x'} f)}{\partial x},\ g^{on} = k \cdot (\exists_{x'} f)_x,\ g^{off} = k \cdot (\exists_{x'} f)_{\overline{x}}.$$

Obviously, this hold-constraint is nonvacuous iff

$$\frac{\partial(\exists_{x'} f)}{\partial x} \neq 0$$

Any function ψ, such that $\psi(g, c) \cdot c = g \cdot c$, can be used to select a g from the interval $[g^{on}, \overline{g^{off}}]$ for the above extraction. Note the careset $c = g^{on} + g^{off} = k$. We choose the BDD *Restrict* function [34, 76, 99] since it is efficient and usually decrease the BDD size and does not introduce new variables from the careset to the result.

Note the careset for $x = g$ is also k due to the hold-constraint. However, since $k \leq \exists_X f$, function g does not need to be simplified with respect to f.

It is clear that the above extraction is unique with respect to x and f up to the selection of ψ. Although the extraction is complete, the BDD representation of k can still be optimized with regard to f using BDD *Restrict*. Since $k \leq \exists_X f$, and due to properties of BDD *Restrict*, the onset of k only increase in the optimization, and increment only comes from $\overline{f}$. Thus all the side effect is the addition of "vacuous" prime hold-constraints, which does not destruct the completeness property of the extraction.

7.4.2 Constraint Simplification

A hold-constraint can be used to simplify the constraint from which it is extracted by applying the *conditional substitution*, as defined below.

Definition 7.6 Let e be a hold-constraint $k \rightarrow (x = g)$. The *conditional substitution* of e on a Boolean function f, written $\tau(f, e)$, is

$$\tau(f, e) = k \cdot f_{x:=g} + \overline{k} \cdot f$$

where $f_{x:=g}$ is the substitution of variable x with function g.

Conditional substitution often simplifies a function and removes the variable being substituted. For example, let $f = y + x + v$ and $e := \overline{y} \rightarrow \overline{x}$. Then $\tau(f, e) = \overline{y} \cdot (y + v) + y \cdot (y + x + v) = y + v$. On the other hand, BDD *Restrict*, although widely used as a simplification function, is sensitive to variable ordering and may not improve the result: if x is the top variable, then restricting f on e returns f itself.

It turns out the conditional substitution is a careset optimization function (i.e., a function $\psi(f, g)$ that returns a function that agrees with the function f everywhere in the careset g), just like BDD *Restrict* [34], but insensitive to variable ordering. It also possesses other nice properties:

Property 1: $\tau(f, e) \cdot e = f \cdot e$, i.e., $\tau(f, e)$ is a function equal to f over the careset e.

Property 2: $\tau(f, e)$ decreases the "diversity" of x in f, i.e., $\partial f/\partial x$, by the amount k. This implies that $\tau(f, e)$ is independent of x iff $k \geq \partial f/\partial x$

Property 3: If $\tau(f, e)$ is independent of x and $f \rightarrow e$, then $\tau(f, e) = \exists_x f$.

Property 4: If there exists a careset optimization function $\psi(f, e)$ that does not depend on x then it must be $\tau(f, e)$.

Note the last property makes conditional substitution a better choice than BDD *Restrict* in regard to input variable removal.

In the following, we give proofs of the above properties. We assume $e := k \to (x = g)$ to be the hold constraint and f another constraint.

Proof of Property 1: We need to prove

$$e \cdot \tau(f, e) = f \cdot e \tag{7.9}$$

First we compute the two sides of the above equation. The left-hand side is

$$\begin{aligned} e \cdot \tau(f,e) &= (k \to (x = g)) \cdot (k \cdot f_{x=g} + \overline{k} \cdot f) \\ &= (\overline{k} + (x = g)) \cdot (k \cdot f_{x=g} + \overline{k} \cdot f) \\ &= \overline{k} \cdot f + k \cdot f_{x=g} \cdot (x = g). \end{aligned}$$

while the right-hand side is

$$\begin{aligned} f \cdot e &= f \cdot (k \to (x = g)) \\ &= f \cdot (\overline{k} + (x = g)) \\ &= f \cdot (\overline{k} + k \cdot (x = g)) \\ &= \overline{k} \cdot f + k \cdot f \cdot (x = g). \end{aligned}$$

Then, for Equation (7.9) to hold, we only need to prove

$$f_{x=g} \cdot (x = g) = f \cdot (x = g) \tag{7.10}$$

which can be done again by expanding the two sides. Since the left-hand side is

$$\begin{aligned} f_{x=g} \cdot (x = g) &= (x \cdot f_x + \overline{x} \cdot f_{\overline{x}})_{x=g} \cdot (x = g) \\ &= (g \cdot f_x + \overline{g} \cdot f_{\overline{x}}) \cdot (x \cdot g + \overline{x} \cdot \overline{g}) \end{aligned}$$

$$= \quad x \cdot g \cdot f_x + \overline{x} \cdot \overline{g} \cdot f_{\overline{x}},$$

and the right-hand side is

$$\begin{aligned} f \cdot (x = g) &= (x \cdot f_x + \overline{x} \cdot f_{\overline{x}}) \cdot (x = g) \\ &= (x \cdot f_x + \overline{x} \cdot f_{\overline{x}}) \cdot (x \cdot g + \overline{x} \cdot \overline{g}) \\ &= x \cdot g \cdot f_x + \overline{x} \cdot \overline{g} \cdot f_{\overline{x}}, \end{aligned}$$

therefore, Equation (7.10) holds, and consequently Equation (7.9) holds. ■

Proof of Property 2: We need to prove

$$\frac{\partial \tau(f, e)}{\partial x} = \overline{k} \cdot \frac{\partial f}{\partial x}. \tag{7.11}$$

First, we prove two lemmas: let g and h be two constraints, where g depends on x and h does not, then

(Lemma 1) $\dfrac{\partial (g + h)}{\partial x} = \overline{g} \cdot \dfrac{\partial h}{\partial x}$

(Lemma 2) $\dfrac{\partial (g \cdot h)}{\partial x} = g \cdot \dfrac{\partial h}{\partial x}$

The lemmas are proved as follows:

$$\begin{aligned} \frac{\partial (g + h)}{\partial x} &= (g + h)_x \cdot \overline{(g + h)_{\overline{x}}} + \overline{(g + h)_x} \cdot (g + h)_{\overline{x}} \\ &= (g + h_x) \cdot \overline{(g + h_{\overline{x}})} + \overline{(g + h_x)} \cdot (g + h_{\overline{x}}) \\ &= (g + h_x) \cdot \overline{g} \cdot \overline{h_{\overline{x}}} + \overline{g} \cdot \overline{h_x} \cdot (g + h_{\overline{x}}) \\ &= \overline{g} \cdot h_x \cdot \overline{h_{\overline{x}}} + \overline{g} \cdot \overline{h_x} \cdot h_{\overline{x}} \\ &= \overline{g} \cdot \frac{\partial h}{\partial x} \end{aligned}$$

and

$$\begin{aligned}
\frac{\partial(g \cdot h)}{\partial x} &= (g \cdot h)_x \cdot \overline{(g \cdot h)_{\overline{x}}} + \overline{(g \cdot h)_x} \cdot (g \cdot h)_{\overline{x}} \\
&= (g \cdot h_x) \cdot \overline{(g \cdot h_{\overline{x}})} + \overline{(g \cdot h_x)} \cdot (g \cdot h_{\overline{x}}) \\
&= (g \cdot h_x) \cdot (\overline{g} + \overline{h_{\overline{x}}}) + (\overline{g} + \overline{h_x}) \cdot g \cdot h_{\overline{x}} \\
&= g \cdot h_x \cdot \overline{h_{\overline{x}}} + g \cdot \overline{h_x} \cdot h_{\overline{x}} \\
&= g \cdot \frac{\partial h}{\partial x}.
\end{aligned}$$

Now, since

$$\tau(f, e) = k \cdot f_{x=g} + \overline{k} \cdot f$$

where $k \cdot f_{x=g}$ and $\overline{k}$ are independent of x, by applying the above lemmas, we have

$$\begin{aligned}
\frac{\partial \tau(f, e)}{\partial x} &= \overline{k \cdot f_{x=g}} \cdot \overline{k} \cdot \frac{\partial f}{\partial x} \\
&= (\overline{k} + \overline{f_{x=g}}) \cdot \overline{k} \cdot \frac{\partial f}{\partial x} \\
&= \overline{k} \cdot \frac{\partial f}{\partial x}
\end{aligned}$$

therefore, the property is true. ■

Proof of Property 3: We prove that if $\tau(f, e)$ is independent of x and $f \to e$, then $\tau(f, e) = \exists_x f$.

Since $f \to e$, i.e., $f \leq \overline{k} + (x = g)$, by cofactoring both sides with respect to x we get $f_x \leq \overline{k} + g$, which implies $k \cdot f_x \leq k \cdot g$, thus $k \cdot g \cdot f_x = (k \cdot g) \cdot (k \cdot f_x) = k \cdot f_x$. Similarly, $k \cdot \overline{g} \cdot f_{\overline{x}} = k \cdot f_{\overline{x}}$. Therefore, we have

$$k \cdot f_{x=g} = k \cdot (g \cdot f_x + \overline{g} \cdot f_{\overline{x}}) \tag{7.12}$$

$$= \quad k \cdot (f_x + f_{\overline{x}}) \tag{7.13}$$

$$= \quad k \cdot \exists_x f. \tag{7.14}$$

Further, since $\tau(f, e) = k \cdot f_{x=g} + \overline{k} \cdot f$ is independent of x, by Property 2, we have

$$\frac{\partial \tau(f, e)}{\partial x} = \overline{k} \cdot \frac{\partial f}{\partial x} = 0.$$

Also, from the lemmas obtained in the proof of Property 2, we have

$$\frac{\partial \overline{k} \cdot f}{\partial x} = \overline{k} \cdot \frac{\partial f}{\partial x}.$$

Therefore, $\frac{\partial \overline{k} \cdot f}{\partial x} = 0$, thus $\overline{k} \cdot f$ is independent of x too. From this and the result in (7.14), we have

$$\begin{aligned} \tau(f, e) &= k \cdot f_{x=g} + \overline{k} \cdot f \\ &= k \cdot \exists_x f + \overline{k} \cdot f \\ &= \exists_x (k \cdot f) + \exists_x (\overline{k} \cdot f) \\ &= \exists_x f \end{aligned}$$

Hence, the property is true. ■

Proof of Property 4: We prove that for any function $\psi(f, e)$ such that $\psi(f, e) \cdot e = f \cdot e$ and $\psi(f, e)$ is independent of x, then $\psi(f, e)$ is $\tau(f, e)$.

Let α be a minterm of variables in f and e. If $\alpha \in e$, then $(\psi(f, e) \cdot e)(\alpha) = (f \cdot e)(\alpha)$, i.e., $\psi(f, e)(\alpha) = f(\alpha)$. Similarly, since $\tau(f, e) \cdot e = f \cdot e$ (Property 1), we have $\tau(f, e)(\alpha) = f(\alpha)$. So ψ and τ agree in e.

Now we check the case $\alpha \in \overline{e}$, i.e., $\alpha \in (k \cdot (x \neq g))$. Let $\alpha' \in (k \cdot (x = g))$ be a minterm which differs from α only at variable x. From the definition of $\tau(f, e)$, we know $\tau(f, e)(\alpha) = f(\alpha')$. On the other side, for $\psi(f, e)$ to not depend on x, we must have $\partial(\psi)/\partial(x) = 0$, i.e., $\psi_x = \psi_{\overline{x}}$. Therefore, $\psi(f, e)(\alpha) = \psi(\alpha')$ (actually for all α). Now since $\alpha' \in (k \cdot (x = g)) \subseteq e$, we have $\psi(\alpha') = f(\alpha')$, thus $\psi(f, e)(\alpha) = f(\alpha')$. Hence, ψ and τ also agree in $\overline{e}$. ■

7.4.3 Recursive Extraction

Extracted hold-constraints can be used in extracting more hold-constraints which would otherwise be impossible. The complete and nonvacuous extraction for a set of constraints can be done using the procedure in Theorem 7.2 on the conjunction of the constraints. While we exclude conjoining the original constraints due to efficiency concerns, we can try to extract from the conjunction of the concerned constraint and the already extracted hold-constraints, whose size is usually small. In fact, we can even avoid explicitly conjoining a hold-constraint with a constraint due to the following theorem.

Theorem 7.3 For any hold-constraint $e := k \rightarrow (x = g)$, and Boolean function f, an input variable $v \neq x$ is extractable from $\tau(f, e)$ iff it is extractable from $f \cdot e$, or more precisely,

$$\frac{\partial(\exists_{v'} \tau(f, e))}{\partial v} = \frac{\partial(\exists_{v'} (f \cdot e))}{\partial v}$$

where $v' = X \Leftrightarrow \{v\}$.

Proof: We only need to prove $\exists_{v'}(\tau(f,e)) = \exists_{v'}(f \cdot e)$. This equality is shown in the following computation. Note $f \cdot e = \tau(f,e) \cdot e$ because of Property 1.

$$\begin{aligned}
\exists_{v'}(f \cdot e) &= \exists_{v'}(\tau(f,e) \cdot e) \\
&= \exists_{v'}((k \cdot f_{x=g} + \overline{k} \cdot f) \cdot (\overline{k} + (x = g))) \\
&= \exists_{v'}(\overline{k} \cdot f + k \cdot f_{x=g} \cdot (x = g) + \overline{k} \cdot f \cdot (x = g)) \\
&= \exists_{v'}(\overline{k} \cdot f + k \cdot f_{x=g} \cdot (x = g)) \\
&= \exists_{v'-\{x\}}(\exists_x(\overline{k} \cdot f + k \cdot f_{x=g} \cdot (x = g))) \\
&= \exists_{v'-\{x\}}(\exists_x(\overline{k} \cdot f) + \exists_x(k \cdot f_{x=g} \cdot (x = g))) \\
&= \exists_{v'-\{x\}}(\exists_x(\overline{k} \cdot f) + k \cdot f_{x=g} \cdot \exists_x(x = g)) \\
&= \exists_{v'-\{x\}}(\exists_x(\overline{k} \cdot f + k \cdot f_{x=g})) \\
&= \exists_{v'}\tau(f,e)
\end{aligned}$$

■

The above theorem implies that, given a constraint and a hold-constraint, conditional substitution is an exact method for finding hold-constraints for input variables other than the one being substituted.

Take the same example from Section 7.4.2. Conditional substitution $\tau(f,e) = \overline{y} \cdot (y + v) + y \cdot (y + x + v)$ results in $y + v$, which is another hold-constraint. As expected from Theorem 7.3, the conjunction $f \cdot e = y + \overline{x} \cdot v$ yields the same two hold-constraints. Whereas $f \cdot e$ gives a function more complicated than $\tau(f,e)$, and BDD *Restrict* of f with respect to e returns f, which does not allow the extraction of the second hold-constraint.

Unfortunately, Theorem 7.3 cannot be extended to substitution of more than one hold-constraints. However, extraction preceded by multiple substitutions has shown in our experiments to be effective in extracting more hold-constraints.

7.5 The Overall Algorithm

We always perform the syntactical extraction first because it is fast, and it simplifies the constraint formula prior to BDD building. The ensuing functional extraction is iterative. In the first iteration, the extraction is done for each constraint. Subsequently, if there are extractions from the last iteration, they are substituted in to find more extractions for the input variables that do not have an extraction yet. At the end of each iteration, the new extractions are used to simplify the remaining constraints. The procedure will terminate because the number of input variables in each constraint is finite.

7.6 Related Works

The idea of extracting hold-constraints stemmed from our observation of real-life design constraints in which inputs are constantly assigned values stored in memory elements. A syntactical extraction was the natural choice at the conception of this idea. The attempt on a functional extraction was inspired by the *state assignment extraction* work by Yang, Simmons, Bryant, and O'Hallaron [111]. The key result in their work is as follows:

```
extract(C) {
    E = currE = ∅;
    (C,E) = synt_extract(C);
    do {
        prevE = currE;
        currE = ∅;
        foreach f in C {
            f'=cond_subst(f, prevE);
            foreach not-yet-extracted input variable x in f' {
                e_x = func_extract(f', x);
                if (e_x ≠ nil) currE = currE ∪ {e_x};
            }
        }
        E = E ∪ currE;
        C = simplify(C, currE);
    } while (currE ≠ ∅)
}
```

Figure 7.1: Hold-constraint extraction

Theorem 7.4 Let f be a Boolean formula, then

$$f \Leftrightarrow (x \in g) \cdot h$$

where x is a variable whose possible values are in the set L, and $h = \exists_x f$ and $g = \psi(t, h)$ — ψ is a simplification function which uses the careset h to minimize t. The relation $t \subseteq h^{on} \times L$ is computed as:

$$\bigvee_{l \in L} (ITE(f|_{v \leftarrow l}, \{l\}, \emptyset))$$

If t is a partial function (i.e., each minterm in h^{on} correspondents to a unique value in L), then g is a function, and $f \Leftrightarrow (x = g) \cdot h$.

However, there is no distinction of state and input variables in their work and the assignments are unconditional. We attempted to modify the above approach to meet our needs in the "natural" way as given by the following theorem.

Theorem 7.5 Let f be a Boolean formula, $k = \overline{\forall_x f}$ and $h = \exists_x f$. Let ψ be a simplification function which uses the careset h to minimize t. Let $e = f \cdot k$ and $g = \psi(e_x, \exists_x e)$. Then $f \Leftrightarrow (k \rightarrow (x = g)) \cdot h$.

We needed to make sure k and g do not depend on any input variables by applying careset optimization. Even so, we failed to obtain some obviously extractable hold-constraints, for example, $y_1 + x_1$ in the constraint

$$f = (y_1 + x_1) \cdot (y_2 \cdot x_2 + x_1) \tag{7.15}$$

where y_1, y_2 are the state variables, and x_1, x_2 the inputs.

It turns out that the above method works only if f has a *conjunctive bi-decomposition* such that the intended input variable and the rest of the input variables belong to different conjuncts. This is an obvious limitation.

Bertacco and Damiani [14] proposed a method to build the decomposition tree for a Boolean function from its BDD representation. Their method has a similar restriction that the variable supports of the components be disjoint, and is therefore not suitable for our application.

An earlier work by McMillan [81] gives similar results as that of Yang *et al.* His method utilizes the BDD *constrain* operator, and can factor out *dependent variables* from Boolean functions. However, because the dependency is unconditional (i.e., in our case, for all state valuations), the method can not be adopted for a complete extraction, either. The example in (7.15) above also showcases the inablility of this method to extract all hold-constraints.

It can be proved that a hold-constraint on x is extractable from f iff there exists a conjunctive bi-decomposition of f such that one conjunct depends on x and some state variables, and the other can depend on all the variables. Our test in Theorem 7.2 detects exactly such a decomposition.

The closest works on similar decompositions are those of Bochmann, Dresig, Steinbach [16] and Mishchenco, Steinbach, Perkowski [84], from the synthesis and optimization community. They proposed a set of criteria for various *groupabilities*, including one that tests whether two (disjoint) groups of variables can be separated in a conjunctive bi-decomposition. This seemed to match our need of checking if an input variable can be factored out from a constraint. However, the grouping also depends on a third group of variables that is shared by the conjuncts. In our case, this can be any subset of the set of state variables. Therefore, it can take multiple (in the worst case, exponential to the number of state variables) groupability tests to decide if a hold constraint exists for an input variable. Our approach needs just one test.

7.7 Experiments

7.7.1 Impact on building conjunction BDDs

The experiments are intended to compare the effect of hold-constraint extraction on building BDDs for the partitioned constraints. Six commercial designs are used in the experiments. Four configurations are compared, namely

no-extraction: with no extraction

syntactical: with the syntactical extraction

functional1: with the nonrecursive functional extraction

functional2: with the recursive functional extraction

First, we demonstrate in Table 7.1 the effect of the three types of extractions. For this experiment only, the functional extractions are run without first applying syntactical extraction for sanity check that the former always subsumes the latter. Columns 1 and 2 gives the number of constraints and input variables of the designs, respectively. The *#e_c* and *#e_i* columns give the numbers of constraints and input variables with extractions, respectively. As can be seen, the functional extractions always perform better, and a recursive extraction is more powerful than a nonrecursive extraction.

Tables 7.2 through 7.5 compare the results of building BDDs for the partitioned constraints. The reported times and BDD node counts include times and BDD nodes used in extraction. Dynamic variable reordering is enabled in all examples except in *rio*, where a fixed order is used to avoid BDD blowup. Table 7.2

	circuit stats		syntactical		functional1		functional2	
circuit	#cons	#input	#e_c	#e_i	#e_c	#e_i	#e_c	#e_i
mmq	117	207	18	25	42	49	77	53
qbc	93	174	58	169	75	174	75	174
qpag	215	283	149	282	173	282	197	283
qpcu	109	34	75	34	93	34	93	34
rio	198	371	80	283	89	289	96	292
sbs	108	423	95	422	96	423	97	423

Table 7.1: Result of extractions

circuit	#conj	#part	peak	result	time
mmq	92	26	82782	24535	17.0
qbc	60	34	10220	2689	2.5
qpag	187	29	968856	142943	272.4
qpcu	94	11	14308	4563	1.0
rio	133	66	1299984	375723	3.3
sbs	69	40	12264	2940	1.5

Table 7.2: Result of building conjunction BDDs (*no extraction*)

shows the results of building conjunction BDDs without any extraction. Column 1 is the number of BDD conjunction operations performed during partitioning. Column 2 shows the number of resulting parts. Column 3, 4, and 5 show the peak number of BDD nodes, number of BDD nodes in the result, and the time for building the BDDs, respectively.

The impact of extractions on BDD building is shown in Table 7.3, 7.4 and 7.5, respectively for *syntactical*, *functional1* and *functional2*. First, it should be clarified that the sharp increase of number of parts between the functional and

circuit	#conj	#part	peak	result	time
mmq	80	37	32704	9501	8.9
qbc	19	77	9198	1754	0.1
qpag	65	156	155344	89195	61.4
qpcu	24	84	4088	983	0.0
rio	97	127	1040396	149152	1.5
sbs	10	104	15330	1663	0.1

Table 7.3: Result of building conjunction BDDs (*syntactical*)

the syntactical extractions is partially due to the fact that the former extracts signals bit by bit, while the latter can extract bus signals at once. Therefore, the effect of partition refinement is more proportionally represented in the number of conjunctions performed among the simplified versions of the original constraints (note we do not conjoin the hold-constraints). It can be seen that as the extraction gets more powerful, the number of parts increases. Although the number of conjunctions decreases quickly in *syntactical* and more in *functional1*, we observed that *functional2* does not improve the number further, although it has the most extractions. As a result, BDD sizes in *functional2* increase slightly over those in *functional1*. Obviously, simplification from the extra extractions in *functional2* in our examples did not remove more input variables to refine the partition, although in theory it could. Overall, functional extractions have large improvement over its syntactical counterpart. In all examples, extracting hold-constraints has a clear advantage in time (up to 23 times faster) and space (up to 7 times smaller) usages over not extracting such constraints.

circuit	#conj	#part	peak	result	time
mmq	72	58	33726	9529	6.2
qbc	10	92	5110	1294	0.0
qpag	60	216	47012	20613	12.3
qpcu	17	136	4088	801	0.0
rio	94	140	1072460	142997	1.5
sbs	10	503	34748	1612	0.1

Table 7.4: Result of building conjunction BDDs (*functional1*)

circuit	#conj	#part	peak	result	time
mmq	72	58	33726	9529	6.2
qbc	10	113	6132	1428	0.0
qpag	60	268	79716	28221	26.7
qpcu	17	146	13286	1042	0.0
rio	94	144	1226400	143071	1.6
sbs	10	528	49056	1812	0.1

Table 7.5: Result of building conjunction BDDs (*functional2*)

circuit	without extract	with extraction	speedup
mmq	6.98	4.57	1.53
qbc	1.55	1.24	1.25
qpag	6.46	2.57	2.51
qpcu	0.49	0.55	0.89
rio	302.63	132.00	2.30
sbs	2.12	1.98	1.07

Table 7.6: Impact on simulation generation

7.7.2 Impact on Simulation

As discussed in Section 7.1, simulation directly benefits from finer partition that results in smaller conjunction BDDs. Hold-constraints by themselves also contribute to the speedup of vector generation since they can produce quick solution to input variables once the state evaluation is known. Table 7.6, each design is simulated 3 times, each with 1000 cycles, using randomly generated inputs from the conjunction BDDs. Nonrecursive extraction was used in this experiment. Column 1 and 2 report the average times spent in simulation generation from BDDs with and without extraction, respectively. Column 3 gives the ratio of generation time without extract to that with extraction. The speedup is more proportional to the reduction in BDD size when the conjunction BDDs get more complex, for example, in *rio*, *qpag* and *mmq*. For smaller BDDs, the overhead of handling finer partitions may offsets the size reduction, which results in increase of generation time in *qpcu*. Nonetheless, our concern is more with the long generation time from large conjunction BDDs, in which cases we achieved a speedup ratio of about 2.5.

7.8 Summary

We have presented a method for simplifying constraint solving in random test generation. The source of the simplification is the refining of constraint partition by extracting deterministic assignments to input variables. The result is a faster construction and smaller size of BDD representation of the constraints. Simulation vector generation time is also reduced due the smaller BDD size, and the constant assignments from the fast solution of hold-constraints.

Chapter 8

Constraint Synthesis

8.1 Introduction

The algorithms in SymGen provides an efficient way of generating vectors from constraints and input biases without backtracking. However, as it is tailored for efficient simulation vector generation, SymGen's one-vector-at-a-time approach may not be suitable for other types of verification methods, e.g., model checking, which explores the design state space by checking all possible input combinations simultaneously. Similarly, hardware-accelerated simulation (also known as emulation [70]) gives another example of SymGen's limitation, though from a different perspective. In emulation, the design is mapped to some programmable hardware, such as the Field Programmable Gate Arrays (FPGAs); the stimuli are fed in from a computer, usually in batch mode to save time. Although in theory we can implement the SymGen algorithms in software program on the computer to generate the stimulus, the overhead of communication between the emulation hardware and the computer at each simulation cycle would defeat the very purpose of emulation being a faster simulator. The problems in both examples can be remedied if the vectors are generated as the outputs of a Boolean circuit, instead of directly from the constraints: combinations of values at these outputs represent the full input space for a model checker, and the functions can be mapped to the emulation hardware

thus eliminating the communication cost. We refer to this approach to vector generation as *constraint synthesis*. Figure 8.1 illustrates synthesized constraints being an alternative to SymGen in modeling verification environment.

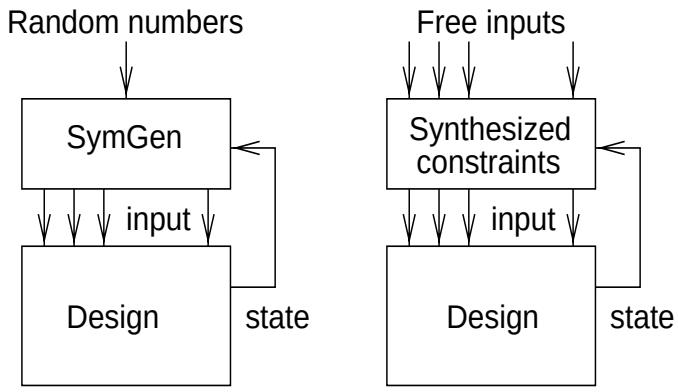


Figure 8.1: SymGen and Synthesized Constraints

In this chapter, we present a method of synthesizing hardware constraints, referred to as the *cascaded synthesis*. In general, constraint synthesis can be viewed as the following problem: for a constraint $f : B^n \mapsto B$, derive a mapping $B^n \mapsto B^n$ whose range is the onset of f. This formulation is equivalent to the *Boolean Unification* (BU) [34, 79, 110] problem of finding a substitution σ that unifies the constraint f and the constant 1 , i.e., $\sigma(f) = 1$. However, our approach is necessarily different from BU because in the context of synthesizing hardware constraints, we are facing with the heterogeneousness of the variables, specifically, the fact that the state variables, unlike the inputs, are bounded by the design. More specifically, when synthesizing constraints of both state and input variables, the method in [34] would give wrong results, while the method in [79, 110] would give suboptimal results. We defer a detailed discussion of this to Section 8.4.

The remainder of this chapter is organized as follows: Section 8.2 gives a formal formulation of the constraint synthesis problem. In Section 8.3, we present the synthesis method that handles state-dependent constraints. Section 8.4 covers the related works. We summarize in Section 8.5.

8.2 Problem Formulation

Let $f : B^{m+n} \mapsto B$ of input variables $X = \{x_1, \ldots, x_n\}$ and state variables $Y = \{y_1, \ldots, y_m\}$ be the constraint of concern. Denote the set of *ground solutions* to

$$f(X, Y) = 1 \tag{8.1}$$

by S, with

$$S = \{\{\alpha, \beta\} \mid \alpha \in B^n,\ \beta \in B^m,\ f(\alpha, \beta) = 1\}. \tag{8.2}$$

Let

$$S_\beta = \{\alpha \in B^n \mid \exists\beta(\{\alpha, \beta\} \in S)\} \tag{8.3}$$

be the set of input vectors pertaining to state β in S. A state β is *legal* in f iff $S_\beta \neq \emptyset$. Let

$$L = \{\beta \in B^m \mid S_\beta \neq \emptyset\} \tag{8.4}$$

be the set of legal states in f. We adopt Rudeanu's [97] terminology, with modifications to accommodate states, for describing different kinds of solutions to equations: A vector of functions $F : B^{n+m} \mapsto B^n$ is called a *solution* to (8.1) iff

$$\forall\beta \in L,\ F(B^n \times \{\beta\}) \subseteq S_\beta, \tag{8.5}$$

F is called a *general solution* iff

$$\forall \beta \in L, F(B^n \times \{\beta\}) = S_\beta, \tag{8.6}$$

and F is called a *reproductive solution* iff F is a general solution, and

$$\forall (\beta \in L, \alpha \in S_\beta),\ F(\alpha, \beta) = \alpha. \tag{8.7}$$

Reproductive solutions hold an important property: under a legal state, any solution G of f can be obtained as an instance of a reproductive solution F.

Theorem 8.1 Let F be a reproductive solution to the equation $f(X, Y) = 1$, and L the set of legal states of $f(X, Y)$. Then for any solution G of this equation,

$$G(B^n \times L) = F(G(B^n \times L)).$$

Since our goal is to be able to generate all possible vectors, constraint synthesis can therefore be formulated as the problem of finding a general or reproductive solution of the constraint.

8.3 The Cascaded Synthesis Method

We describe a synthesis method which generates values for inputs iteratively in a "cascaded" fashion wherein the previously generated values are used in the generation of the next input.

8.3.1 Cascaded Solution Generation

Given the constraint f of input variables $X = \{x_1, \ldots, x_n\}$ and state variables $X = \{y_1, \ldots, y_m\}$, we first define the *projection* of f onto the variables $x_i, \ldots, x_n$ and $y_1, \ldots, y_m$, written f^i, for $1 \leq i \leq n+1$, to be the existential quantification of f with respect to the rest of the variables. Note the two special cases $f^1 = f$, and $f^{n+1} = L$, where L is the set of legal states. The notion of projection is critical to our synthesis procedure because of the following theorem.

Theorem 8.2 Given a constraint f and a state, either all, or none, of the projections $f^1, \ldots, f^{n+1}$ are satisfiable.

Proof: The theorem follows directly from the definition of projections, which are existential quantifications of f. ■

Note by definition projection f^1 is f. Thus, given a state valuation and a canonical representation of f, such as one in BDD, we can quickly determine whether f is satisfiable (in time $O(n+m)$), and therefore whether all of its projections are satisfiable. In addition, if there is a satisfying assignment, say π, to variables in f^i, then there exists an assignment to x_{i-1}, which together with π satisfies f^{i-1}. This can be generalized to the following theorem.

Theorem 8.3 Under a legal state, any input assignment satisfying f^i is a suffix of an input assignment satisfying f^j, for all $j < i$.

Proof: It suffices to prove for the case $j = i \Leftrightarrow 1$. Let α be an input assignment to $x_i, \ldots, x_n$ and β a legal state, such that $f^i(\alpha, \beta) = 1$. From the definition of

projections of f, we have

$$f^i(x_i, \ldots, x_n, Y) = f^{i-1}(0, x_i, \ldots, x_n, Y) + f^{i-1}(1, u_i, \ldots, u_n, Y),$$

plugging in α and β, we have

$$f^i(\alpha, \beta) = f^{i-1}(0, \alpha, \beta) + f^{i-1}(1, \alpha, \beta).$$

The left-hand side of the above equation is 1, therefore at least one of the terms on the right-hand side must be true. That is, either $(0, \alpha)$ or $(1, \alpha)$ or both, under the state β, is a solution to f^{i-1}. ■

The above theorem demonstrates that it is feasible to construct a solution for f by successively solving for inputs for the projects, from f^n to f^1, and therefore f. We now give a decision procedure as to how each input is computed.

First we note that all Boolean functions, and in particular the projection f^i, can be decomposed as in the following.

$$f^i = f^i_{x_i} \cdot f^i_{\overline{x_i}} + x_i \cdot f^i_{x_i} \cdot \overline{f^i_{\overline{x_i}}} + \overline{x_i} \cdot \overline{f^i_{x_i}} \cdot f^i_{\overline{x_i}} \tag{8.8}$$

where $f^i_{x_i}$ is the cofactor of f^i with respect to x_i. This decomposition, over the orthogonal basis $\{f^i_{x_i} \cdot f^i_{\overline{x_i}},\ f^i_{x_i} \cdot \overline{f^i_{\overline{x_i}}},\ \overline{f^i_{x_i}} \cdot f^i_{\overline{x_i}}\}$, is obvious from the Shannon decomposition.

Now we show how to derive the value of x_i from a partial assignment to $x_{i+1}, \ldots, x_n$. Let α be a partial assignment under a legal state β such that

$$f^{i+1}(\alpha, \beta) = 1.$$

Since $\exists_{x_i} f^i = f^{i+1}$, then $\exists_{x_i} f^i(\alpha, \beta) = 1$, wherein by substituting f^i for the right-hand side of Equation (8.8) and applying the existential quantification, we have

$$(f^i_{x_i} \cdot f^i_{\overline{x_i}} + f^i_{x_i} \cdot \overline{f^i_{\overline{x_i}}} + \overline{f^i_{x_i}} \cdot f^i_{\overline{x_i}})(\alpha, \beta) = 1.$$

Furthermore, since the above three disjunctive terms are orthogonal, exactly one of them evaluates to 1. Denote this term by t. An assignment to x_i that satisfies $f^i(x_i, \alpha, \beta)$ can then be derived from t and Equation (8.8): If t is the first term in the equation, then f^i is satisfied no matter what value x_i takes, i.e., x_i is a *don't care*; if t is the second (resp. third) term, then x_i has to take the value 1 (resp. 0). To completely specify x_i, we assign x_i as a don't care in the default case. Therefore, we have

$$x_i = \begin{cases} u_i & \text{if } f^i_{x_i} \cdot \overline{f^i_{\overline{x_i}}} \\ 1 & \text{if } \overline{f^i_{x_i}} \cdot \overline{f^i_{\overline{x_i}}} \\ 0 & \text{if } \overline{f^i_{x_i}} \cdot f^i_{\overline{x_i}} \\ d_i & \text{if } f^i_{x_i} \cdot f^i_{\overline{x_i}} \end{cases} \tag{8.9}$$

where u_i and d_i are the two don't cares.

We now come to a critical observation: the condition of d_i is false iff the underlying state is legal; therefore, under an illegal state, d_i actually dictates the computation of x_i.

Lemma 8.1 The condition for d_i in (8.9) is false iff the underlying state is legal.

Proof: From Theorem 8.2, constraint f is satisfiable, i.e., the underlying state is legal, iff every projection f^i for $1 \leq i \leq n+1$ is satisfiable; in the meantime, the don't care condition in (8.9), $\overline{f^i_{x_i}} \cdot \overline{f^i_{\overline{x_i}}}$, is false iff $\exists_{x_i} f^i$, that is f^{i+1}, is satisfiable. Therefore the lemma holds. ■

Although the computation of x_i under an illegal state will be voided eventually, keeping d_i as a don't care (instead of fixing it to 0, which is valid as well as intuitive) can simplify (8.9) in general. We will give an example of this simplification later.

Up to now we have shown that the above procedure generate a solution; but will it be possible to generate all solutions? To answer this question, we look at the only flexibility in the computation of x_i, u_i, since we know d_i does not contribute to x_i under a legal state. By checking the condition for which u_i determines x_i, we conclude that u_i needs to be a free variable to meet the requirement of a general solution, i.e., when possible, x_i should be able to take both values 1 and 0. From now on we treat u_i as a variable, and d_i still a don't care, which can be a function. From Equation (8.9), we derive a solution, or substitution, for x_i:

$$\sigma_i(x_i) := (f^i_{x_i} \cdot f^i_{\overline{x_i}}) \cdot u_i + (f^i_{x_i} \cdot \overline{f^i_{\overline{x_i}}}) + (\overline{f^i_{\overline{x_i}}} \cdot \overline{f^i_{x_i}}) \cdot d_i. \tag{8.10}$$

8.3.2 The Algorithm

In this section, we give an algorithm which finds the substitution in (8.10) for each input variable. We also prove the set of substitutions form a reproductive solution to the constraint.

Figure 8.3 illustrates the cascaded synthesis of a 3-input constraint.

Theorem 8.4 Given a constraint $f(x, y)$ and a legal state, the array of substitutions σ returned by the algorithm in Figure 8.2 is a reproductive solution to f; or more specifically, σ is a mapping $B^n \mapsto B^n$, such that for any $\alpha, \alpha' \in B^n$, and $\sigma(\alpha) = \alpha'$

```
 /* f ∘ g means applying f after g.  */
cascade(f, X, Y) {
    if (X == ∅) return ∅;
    else {
        let X = {x_1, ..., x_n}, X' = {x_2, ..., x_n};
        let f_{x_1} = f(1, X', Y), f_{x̄_1} = f(0, X', Y);
        σ = cascade(f_{x_1} + f_{x̄_1}, X', Y);
        let σ ∘ f_{x_1} = f(1, σ(X'), Y), σ ∘ f_{x̄_1} = f(0, σ(X'), Y);
        σ_1 = (σ ∘ f_{x_1}) · (σ ∘ f_{x̄_1}) · u_1 + (σ ∘ f_{x_1}) · ¬(σ ∘ f_{x̄_1}) + ¬(σ ∘ f_{x_1}) · ¬(σ ∘ f_{x̄_1}) · d_1;
        return {σ_1} ∪ σ;
}
```

Figure 8.2: Cascaded synthesis

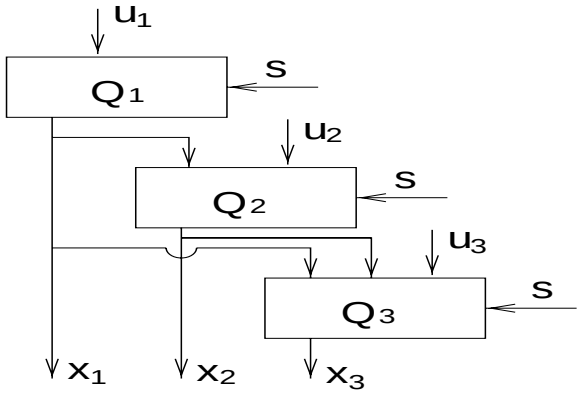


Figure 8.3: Example: Cascaded synthesis of a 3-input constraint

1. α' satisfies f, and

2. if α satisfies f, then $\alpha' = \alpha$.

Proof: We prove the first statement by induction on the length of α'. Let $\alpha' = \alpha'_1 \ldots \alpha'_n$. Since we are under a legal state, from the substitution in Equation (8.10), we know α'_n satisfies f^n. Assume we already have a partial solution $\alpha'_{i+1} \ldots \alpha'_n$ that satisfies f^{i+1}, by applying this solution to the substitution intended for x_i, as indicated in the algorithm, and from Theorem 8.3, the resulting $\alpha'_i \ldots \alpha' n$ must satisfy f^i. Eventually we get α' that satisfies f^1, and therefore f.

The second statement is proved by examining the substitutions. Given a partial solution, α'_i has exactly one choice from u_i, 1, or 0. In the first choice, α'_i is equal to u_i no matter what value u_i takes; in the second choice, $\alpha'_i = 1$, and 1 is the only possible value for the ith position of an α that satisfies f, therefore α'_i also takes 1; similarly, both α'_i and α are 0 in the third choice. ■

8.3.3 Detection of illegal states

We have two options in detecting illegal states. In the first option, we just need to evaluate f with the current state and the solution (or any input vector). The current state is illegal iff the result is 0, as is indicated by the definition of a legal state. We can then pad the solution with this result to indicate the validity of the solution.

In the second option, we need to modify a substitution function σ_i, so that when the condition for the don't care (the fourth term in Equation (8.10)) is true, an

extra output is asserted to indicate an illegal state. Lemma 8.3.1 shows that any σ_i can be used for this test.

The second option seems to be more desirable: we can pick a σ_i with small i whose implementation can be much "simpler" than that of f.

8.4 Comparisons to other synthesis methods

In this section, we compare cascaded synthesis to *Boolean Unification* which is developed for solving Boolean equations, and to a work on building circuits from relations [101].

Boolean unification (BU) is a method for finding solutions for equations of Boolean functions. A solution σ to equation $f = g$ is a substitution to variables in f and g, such that $\sigma \circ f \Leftrightarrow \sigma \circ g$. For this reason, σ is also called a *unifier* of f and g. Since the equation can be rewritten as $f \oplus g = 0$, it suffices for BU to consider only the form $f = 0$.

There are several approaches to the BU problem. Büttner and Simonis [110] presented a BU algorithm that computes the *most general unifier mgu* [102], from which any other unifier can be derived as an instance. Note the reproductive solution produced by cascaded synthesis is an *mgu*. Martin and Nipkow [79] provided a historical perspective on the algorithm and traced the origin back to Boole himself. They also presented a second BU algorithm attributed to Löwenheim [77]. Since this algorithm in general produces a solution more complex than that of Boole's method [79], we will not discuss it further. Fujita *et al.* [53] applied the BU al-

gorithm based on Boole's method in several logic synthesis examples, including minimization of Boolean relations. Coudert and Madre [33] presented an orthogonal approach to BU which utilized a special operator called *Constrain*.

Our approach is necessarily different from the BU methods in that hardware constraints depend on both state and input variables, where the former, being fixed by the design, are in fact parameters. Specifically, when applied to hardware constraint synthesis, the method in [33] can give wrong results because it is based on a mapping that can alter the values of state variables, and Boole's method can give suboptimal results because it does not take the advantage of the optimization space represented by state valuations for which there are no satisfying inputs. Furthermore, even if we apply our synthesis method to the same problem dealt with in BU, the same optimization is still valid which can result in a range of solutions, including the one based on Boole's method, and ones that are even simpler. We discuss the details in the coming subsections.

8.4.1 Constrain-based Synthesis

Coudert and Madre [34] showed that the image of a set can be computed as the range of the transition functions *Constrained* with respect to the set. In addition, the resulting functions form a reproductive solution to the characteristic function representing that set. This fact is used by Coudert and Madre to generate "functional vectors" in [33]. The *Constrain* operation, which we will refer to as the *Generalized Cofactoring* (GC) as it is also known, is defined below.

GC takes two function f and g of variables $\{x_1, \ldots, x_n\}$, and returns a

function which agrees with f in the onset of g. The operation is a mapping decided by g and the variable ordering. The latter determines the "distance" between two vectors, as defined in the following.

Definition 8.1 Let $x_1 \prec \ldots \prec x_n$ be the variable ordering. Let $\alpha = \alpha_1 \ldots \alpha_n$ and $\beta = \beta_1 \ldots \beta_n$ be two minterms. The distance between α and β, in symbols $\| \alpha \Leftrightarrow \beta \|$, is given by:

$$\| \alpha \Leftrightarrow \beta \| = \Sigma_{i=1}^{n} \mid \alpha_i \Leftrightarrow \beta_i \mid \cdot 2^{n-i}. \tag{8.11}$$

This definition of distance reflects the dissimilarity between the two vectors quantified by the variable ordering. We now define GC precisely.

Definition 8.2 Given functions f, g, and a variable ordering, GC of f with respect to g, in symbols $f \downarrow g$, is defined by:

$$(f \downarrow g)(\alpha) = \begin{cases} f(\alpha) & \text{if } g(\alpha) = 1 \\ f(\beta) & \text{if } g(\alpha) = 0 \end{cases} \tag{8.12}$$

where $g(\beta) = 1$ and $\| \alpha \Leftrightarrow \beta \|$ is minimum under the given variable ordering.

Then image computation in [34] can be summarized in the following theorem.

Theorem 8.5 Let $F = [f_1, ..., f_n]$ be a vector of Boolean functions, and $R(X)$ a nonempty set. Define GC of f with respect to $R(X)$ as

$$F \downarrow R(X) := [f_1 \downarrow R(X), ..., f_n \downarrow R(X)]$$

Then the image of R under F is equal to the range of the vectorial function $F \downarrow R$, i.e.,

$$Img(R(X), F) = Img(1, F \downarrow R(X)).$$

Now, let $f_i = x_i$ for $1 \leq i \leq n$ in the above theorem, we obtain the following corollary:

Corollary 8.1 Let X be a set of Boolean variables $\{x_1, ..., x_n\}$, and $R \subseteq B^n$ a BDD representing a nonempty set. Then R is the range of the vectorial function $[x_1 \downarrow R, ..., x_n \downarrow R]$.

This result is applicable to constraint synthesis in the following sense: let R be the constraint, and X the set of input variables; let $X' = [x'_1, ..., x'_n]$ be such that $x'_i = x_i \downarrow R$, then the evaluations of X' always satisfy R; specifically, for any $\alpha \in R$, $X'(\alpha) = \alpha$, hence X' is a reproductive solution to R. In addition, from the definition of GC, for any input vector α, X' generates a vector in R that is "closest" to α.

However, this method may produce wrong results if applied to synthesizing constraints involving state variables, because GC may change the valuation of the state variables in its mapping of an vector of both state and input variables from outside of R to one in R; as a result, what we have generated is an input vector that satisfies R under the *mapped* state, but may be in conflict with the actual state asserted by the design.

In the appendices, we will report two attempts (prior to our work on cascaded synthesis) to modify GC to suit our need. In Appendix A, we describe an extension to GC for handling multiple constraints without explicitly conjoining the constraints. In Appendix B, we report another modification to GC which handles constraints with state variables correctly. However, in both cases, since they involve complicated recursions not suitable for caching, the resulting algorithms are inefficient in practice.

To conclude the comparison with constrain-based synthesis, we show that our approach achieves the shortest distance mapping even when state variables are involved. It also implies that, in the absence of state variables, the two synthesis methods produce identical results.

Theorem 8.6 The substitution returned by the algorithm in Figure 8.2 maps a vector α to a vector α' in the constraint that has the shortest distance (as given in Formula (8.11) from α.

Proof: Suppose we have arrived at a partial solution $\alpha'_{i+1} \ldots \alpha'_n$ which is at a distance l from $\alpha_{i+1} \ldots \alpha_n$. According to the algorithm, the choice of α'_i either maintains the distance, or increases it to $l + 2^{i-1}$. If we were to do things differently and still produce a satisfying partial solution, we could (1) assign $\overline{u_i}$ to α'_i, or (2) backtrack and change an α'_j, $j > i$, so that α'_i would not be forced to 1 or 0. But (1) would increase instead of maintaining l, and (2) would increase l by 2^{j-1}, which is greater than sum of any potential savings at positions $i < j$. ■

8.4.2 BU Based On Boole's Method

BU based on Boole's method is summarized in the following theorem [79]:

Theorem 8.7 Let $\sigma(y) : B^{n-1} \mapsto B^{n-1}$, where $y = (x_2, \ldots, x_n)$, be a *mgu* to

$$f(0, x_2, \ldots, x_n) \cdot f(1, x_2, \ldots, x_n)$$

Then

$$F(x) = ((f(0, \sigma(y)) \oplus f(1, \sigma(y)) \oplus 1) \cdot x_1 \oplus f(0, \sigma(y)),\ \sigma(y))$$

is an *mgu* to $f(x_1, \ldots, x_n) = 0$.

Note "$\oplus$" is the Exclusive-Or operator. The BU algorithm computes the substitution σ recursively using the above theorem. To facilitate a comparison, we expand the Exclusive-Or's and list the algorithm in Figure 8.4, and give the variation of our synthesis method on the problem $f = 0$, which is the dual of our original problem ($f = 1$).

Apparently, there are three differences: (1) We have extra state variables, z. (2) We do not perform the 0-test when the input variables are exhausted; as far as the original f is not a constant zero, i.e., is satisfiable under *some* state and input, the synthesis always succeeds. (3) last but most importantly, our result differs from BU at one place in the formula for computing the substitution of an input variable: whereas we have $f(0, \sigma(y), z) \cdot f(0, \sigma(y), z) \cdot d_1$ as the third term, BU has $f(0, \sigma(y)) \cdot \overline{u_1}$. As we have explained in Section 8.3.1, the cascaded synthesis opts to keep the third term because it handles a second type of variables – the state

```
unify(f(x)) {
    if (x == ∅) {
        if (f(x) ⇔ 0) return (); else fail;
    } else {
        let x := {x_1} ∪ y;
        σ = unify(f(0, y) · f(1, y));
        return (
            f(1, σ(y)) · f(0, σ(y)) · u_1 +
            f(1, σ(y)) · f(0, σ(y)) +
            f(0, σ(y)) · u_1,
            σ
        );
    }
}
```

Figure 8.4: Boolean Unification

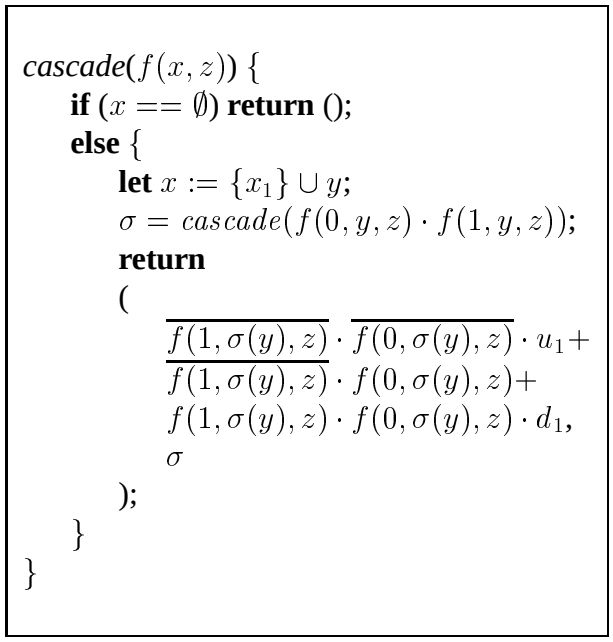


Figure 8.5: Cascaded synthesis — another form

variables. Further, even if we apply cascaded synthesis to constraints where there is only one type of variables, our decision to keep the the third (don't care) term is still sound as far as there is a solution. Note the substitution, and the don't care optimization as a consequence, does not alter the detection of a solution, e.g., the the 0-test in the case of BU.

Therefore, cascaded synthesis gives a range of reproductive solutions. In particular, one instance is BU, since for any d_1 in Figure 8.5 (note we omit the state variables z) that satisfies

$$f(1,\sigma(y)) \cdot f(0,\sigma(y)) \cdot d_1 = f(0,\sigma(y)) \cdot \overline{u_1}$$

cascaded synthesis becomes BU. In addition, we can obtain a much simpler substitution, ρ, for BU by assigning $d1$ to 1:

$$\begin{aligned}
\rho(x_1) &= \overline{f(0,\sigma(y))} \cdot \overline{f(1,\sigma(y))} \cdot x_1 + && (8.13)\\
&\quad \overline{f(1,\sigma(y))} \cdot f(0,\sigma(y)) + f(1,\sigma(y)) \cdot f(0,\sigma(y)) \cdot 1 && (8.14)\\
&= \overline{f(0,\sigma(y))} \cdot \overline{f(1,\sigma(y))} \cdot x_1 + f(0,\sigma(y)) && (8.15)\\
&= \overline{f(1,\sigma(y))} \cdot x_1 + f(0,\sigma(y)). && (8.16)
\end{aligned}$$

The substitution ρ is again an *mgu* since all other unifiers can be derived by applying a substitution to ρ as shown in the following proof.

First, let $\overline{f(1,\rho(y))} := a$ and $f(0,\rho(y)) := b$, i.e., $f = a \cdot x + b \cdot \overline{x}$. Also let τ be an arbitrary unifier of f and 0, and pick a substitution $\lambda(u) := \tau(x)$.

We show that composing λ and ρ leads to τ. First, we have

$$\lambda \circ \rho(x) = \overline{a} \cdot \lambda(u) + b = \overline{a} \cdot \tau(x) + b. \tag{8.17}$$

Now since τ is a unifier of f and 0, we have

$$a \cdot \tau(x) + b \cdot \tau(\overline{x}) = 0,$$

hence

$$a \cdot \tau(x) = 0 \text{ and } b \cdot \overline{\tau(x)} = 0,$$

and furthermore

$$\overline{a} \cdot \tau(x) = \tau(x) \text{ and } \tau(x) + b = \tau(x).$$

Finally, apply the above equations to (8.17), we have

$$\lambda \circ \rho(x) = \overline{a} \cdot \tau(x) + b = tau(x) + b = \tau(x).$$

■

Lastly, we acknowledge the work by Fujita *et al.* [53] that outlined applications of BU to logic synthesis problems similar to ours which can involve two groups of variables; only one group is unified. Nevertheless, they used the original BU algorithm which unifies all the variables and performs the 0-test. It is thus unclear how they dealt with the 0-test, since the test is bound to fail without unifying all the variables; neither is it likely, from their paper, that they recognized our way of using the don't cares. This can lead to sub-optimal result in their application

of minimizing Boolean relations; we give a simple example to demonstrate this limitation.

Let $a + y \cdot z = 0$ be the relation of concern, where y and z are to be unified. Ignoring the 0-test, and using the BU algorithm as is, they would have the solution for y and z

$$\sigma(y, z) = (\overline{a} \cdot y\overline{z} + a \cdot \overline{y},\ a\overline{z} + \overline{a} \cdot z) \tag{8.18}$$

which can not be simplified further; whereas by applying the cascaded synthesis, we have

$$\sigma(y, z) = (\overline{a} \cdot y\overline{z} + a \cdot d_1,\ a \cdot z + \overline{a} \cdot d_2)$$

now choosing the don't cares $d_1 := y \cdot \overline{z}$ and $d_2 := z$, we obtain

$$\sigma(y, z) = (y\overline{z},\ z) \tag{8.19}$$

which is considerably simpler than the BU solution in (8.18).

8.4.3 Shiple and Kukula's Method

Shiple and Kukula [101] reported a method of synthesizing Boolean relations. The synthesis starts from a BDD representation of the relation, and constructs a gate level logic which can be thought of as the union of two circuits, both of which are isomorphic, topologically, to the graph of the BDD. The first circuit evaluates the current state and generates, for each of its nodes, a signal similar to the node *weight* in SymGen. Each node in the second circuit, according to the weights, assigns to an output to 0, 1, or a random value presented at a free input; however,

only one assignment to each output is selected depending the selected assignments to outputs that are ranked higher in a fixed ordering.

The method can be viewed as a smart implementation of SymGen in hardware. The complexity of their method, as is that of SymGen, is linear in the size of the BDD, or more precisely, O(k) for a BDD with k nodes, since there are two circuits isomorphic to the BDD. The cascaded synthesis can be purely algebraic, in which case, its complexity cannot be measured with respect to BDD node count. Suppose we perform cascaded synthesis on a BDD with k nodes, its space complexity can then be formulated as $k_1 + \cdots + k_n$, where k_i is the number of gates in the ith substitution function σ_i. Roughly speaking, k_n is comparable to k. Note $\sigma_n, \ldots, \sigma_1$ depend on descending numbers of inputs. Also, they can be based upon different BDD variable ordering.

8.5 Summary and Discussion

We have described a method of synthesizing Boolean constraints. Using constraints is an effective alternative to writing up complicated environment models in functional verification. Constraint synthesis facilitates the application of this methodology to areas where no proprietary constraint solving capability is available, e.g., model checking and emulation.

We have also provided a survey of constraint synthesis methods that are known to us. Although Boolean Unification is shown to be *unitary* [110], that is, if there exists a solution, then there is a unique *mgu* from which all solutions can be derived as its instances, the uniqueness is nevertheless only up to an equivalence

relation [102]. For example, a BU problem can have two *equivalent mgu*'s which can be derived from each other. The fact that there actually can be multiple functionally distinct *mgu*'s gives rise to the need of optimization. Our application of don't cares in cascaded synthesis is such an attempt. It is also obvious that synthesis result varies with variable ordering in both constrain-based BU and the one built upon Boole's method. In the latter case, Büttner [109] has shown how to decide the variable ordering in order to obtain an *mgu* with the minimal number of variables.

An approach that allows maximal optimization should be one in which both the don't cares and variable ordering are exploited. In general, we do not have to follow any particular order in the synthesis, e.g., we can solve variable x before y on one path of the decision making, and in the reverse order on another. This would lead us to the departure from the class of "shortest-distance" mappings, which is a sufficient but not necessary condition for constrained vector generation — all we need here is a general solution that encompasses all the ground solutions.

Chapter 9

Conclusion

This dissertation is a summary of cooperative research works that took place over six years while the author was a part-time graduate student and a full-time verification engineer. A colleague of mine once had a remark which I will remember for a long time to come:"verification is a problem that is easy to understand but hard to contribute to." Indeed, as most verification problems are deemed NP-complete or harder, fundamental breakthroughs have been rare; instead, most advances in this area are of an applicative nature, for example, employment of new data structures, and combinations of different methodologies. It is our hope that this dissertation has contributed to verification problems in these manners. In addition to various improvements shown in experimental results, we are also encouraged by the fact that the idea of mixing formal and informal verification techniques, of which our work on saturated simulation and retrograde analysis is widely recognized as one of the pioneering works, is being embraced by the design automation community. It is also a rewarding experience to witness the growth of our vector generation tool, SymGen, from conception into maturity while being adopted in many design projects in Motorola throughout the past years.

9.1 Summary

We have presented applications of symbolic methods to verification problems. In Chapter 3, we introduced to invariant checking the concept of "saturated simulation", which improves the verification coverage by focusing on the control behavior of the design. The improvement comes from an abstraction of equivalent classes of states or state transitions, with the aid of BDD based image computation and abstraction function. We described in Chapter 4 the technique of "retrograde analysis", also applicable to invariant checking. The technique works by enlarging the set of target states, accomplished with BDD based pre-image computation, and by using the hamming-distance heuristic in selecting starting states that are "closest" to the target. Consequently, the simulation has a better chance of reaching a target state in shorter time.

The remaining chapters are devoted to the problem of constrained simulation vector generation. In Chapter 5, we developed an efficient BDD algorithm that takes a set of constraints and input biases, and generates vectors accordingly. The algorithm, implemented in SymGen, is linear to the size of the BDD graph representing the conjunction of the constraints, and requires no backtracking – a common problem in many constraint based vector generation tools. In the next chapter, we discussed the problem of diagnosing illegal states. Simplification of constraint solving, based on hold-constraint extraction, was presented in Chapter 7. This simplification, together with the disjoint-input-support partitioning given in Chapter 5, has greatly enhanced the constraint solving capability of SymGen, thus enabling its handling of nontrivial commercial designs. Lastly in Chapter 8, we

described an alternative method of generating constrained vectors. The problem is formulated as one that derives the "reproductive solution" of the constraints: the constraints are synthesized into a set of functions, whose inputs consist of the state variables and auxiliary free variables, and whose outputs represent the full input vector space in which the constraints are satisfied. Vector generation in this way can be implemented in gate-level (netlist) Boolean logic, thus enabling the use of input constraints in other forms of verifications, e.g., model checking and hardware emulation, where a proprietary constraint solver such as SymGen is not applicable.

9.2 Future Work

There are several areas in which we wish to make improvements. In saturated simulation, the capacity and efficiency can be further enhanced by performing better abstraction. For example, instead of selecting one or several data states from each equivalent class arbitrarily, we can select a largest cube so that a maximum amount of data information is preserved, and a minimum number of next state functions are needed for the image computation.

In regard to constrained vector generation, a desirable extension is to solve constraints based on *word-level* operations (e.g., $=$, $+$, and $\leq$) directly, instead of first compiling them into Boolean operations. We also envision two extensions to the partition-based simplification of constraint solving. First, we can lift the restriction that the partitioning of input variables only comes from a conjunctive decomposition of the constraints — in fact, SymGen can be generalized to solving decompositions of arbitrary types. Here is a sketch. Let f' be the binary decom-

position tree of a constraint f of variables X; the set of leaf nodes of f' is X, and the other nodes, denoted by G, represent the subfunctions. A SymGen-like vector generation can proceed as in the following:

1. Assign to each g in G an auxiliary variable a_g, assign to a_g a probability (between 0 and 1).

2. Build BDDs for f' and all subfunctions in G using X and the auxiliary variables

3. Apply to the BDD of f' the *Weight* procedure augmented with the following rules:

   (a) The weight of a node corresponding to an auxiliary variable a_g is equal to

      i. the weight of the left child, if the weight of g is 0

      ii. the weight of the right child, if the weight of g is 1

      iii. the sum of weights of the children weighted by the probability of a_g, if the weight of g is between 0 and 1

   (b) In the following *Walk* procedure, treat a_g as a state variable in the above first two cases; treat it as an input variable in the last case

4. Apply to the BDD of f' the *Walk* procedure, with the modification that if the current input node corresponds to an auxiliary variable a_g, then

   (a) If the assignment to a_g is 0, replace every branching probability p in the BDD of g with $1 \Leftrightarrow p$

(b) Apply *Walk* to the BDD of g

It obvious that the above procedure is insensitive to the types of decompositions and can be easily extended to handle arbitrary levels of decomposition. It is shown in [40] that *fully sensitive* functions, including all Boolean functions, have a finest decomposition wherein the subfunctions have disjunctive support. Although in general the cost of finding such a decomposition is exponential in the number of variables, there are methods for obtaining coarser decompositions, for example, the algorithms based on BDD in [13, 14]. We conjecture efficient algorithm exists for finding similar decompositions wherein only the *input supports* are disjoint among the subfunctions.

An even more aggressive extension is to allow overlaps of input variables in the support of the subfunctions, which can lead to decompositions finer than that obtained in [40]. In principle, if there is a partial order among the involved subfunctions such that, for any satisfiable assignments to the overlapping inputs in a subfunction, there is at least one satisfiable assignment to the same inputs in all lower-ranked subfunctions, then the given constraints can be solved without backtracking in the extended SymGen described above. Although finding such a decomposition poses a nontrivial challenge, it is nevertheless an attractive optimization which has the potential to outperform the finest disjunctive decomposition.

Appendix A

Generalized Cofactoring for Multiple Constraints

For reference, we first give the BDD implementation of GC with respect to one constraint, as presented by Coudert and Madre [34], in Figure A.1.

```
        f ↓ c {
a1:         assert(c ≠ 0));
a2:         if (c = 1 || f = 1 || f = 0) return f;
a3:         let x_i be the top variable of c
a4:         if (c_{x_i} = 0) return f_{x_i'} ↓ c_{x_i'}
a5:         if (c_{x_i'} = 0) return f_{x_i} ↓ c_{x_i}
a6:         return x_i · (f_{x_i} ↓ c_{x_i}) + x_i' · (f_{x_i'} ↓ c_{x_i'})
a7:     }
```

Figure A.1: Generalized cofactor

We show how GC with respect to multiple constraints can be achieved without first forming the conjunction of the constraints. Let $c = \{c_1, ..., c_m\}$ be a set of constraint BDDs defined over variables $X = \{x_1, ..., x_n\}$, $c \downarrow x_i$ be the vectorial function $[c_1 \downarrow x_i, ..., c_m \downarrow x_i]$, $c \simeq 0$ stand for $\exists c_j \in c, c_j = 0$, and $c \simeq 1$ stand for $\forall c_j \in c, c_j = 1$. Let $c \not\simeq 0$ (resp. $c \not\simeq 1$) mean that it is not the case that $c \simeq 0$ (resp. $c \simeq 1$). Let $\hat{c}$ denote the conjunction $\bigwedge_{c_j \in c} c_j$. Also, let the *top variable of c* denote

the one with the highest rank among the top variables of c_j. It is obvious that $\hat{c} = 0$ if $c \simeq 0$, and $\hat{c} = 1$ iff $c \simeq 1$.

```
        f ⇓ c {
b1:         if (c ≃ 0) return nil;
b2:         if (c ≃ 1) return f;
b3:         let x_i be the top variable of c;
b4:         if (c_{x_i} ≃ 0) return f_{x_i'} ⇓ c_{x_i'};
b5:         if (c_{x_i'} ≃ 0) return f_{x_i} ⇓ c_{x_i};
b6:         t = f_{x_i} ⇓ c_{x_i};
b7:         e = f_{x_i'} ⇓ c_{x_i'};
b8:         if (t = nil) return e;
b9:         if (e = nil) return t;
b10:        return x_i(f_{x_i} ⇓ c_{x_i}) + x_i'(f_{x_i'} ⇓ c_{x_i'});
b11:    }
```

Figure A.2: Generalized cofactor with respect to multiple constraints

The operator $\Downarrow$, defined in Figure A.2, computes the generalized cofactor of f with respect to all the constraints in c, without explicitly building the conjunction $\hat{c}$. We shall prove that $f \Downarrow c = f \downarrow \hat{c}$. Before we do, we need the following lemmas.

Lemma A.1 $\hat{c} = 0$ iff $f \Downarrow c = nil$

Proof:

"$\leftarrow$":

- **Base1**: $c \simeq 0 \rightarrow \hat{c} = 0 \rightarrow$ "$\leftarrow$"

- **Base2**: $c \simeq 1 \rightarrow f \Downarrow c = f$ (line *b2*), which is not nil, hence, "$\leftarrow$"
- **IH**: $f_{x_i} \Downarrow c_{x_i} = nil \rightarrow \hat{c}_{x_i} = 0 \bigwedge f_{x_i'} \Downarrow c_{x_i'} = nil \rightarrow \hat{c}_{x_i'} = 0$
- **IS**: $f \Downarrow c = nil \rightarrow \hat{c} = 0$

  $f \Downarrow c = nil$ implies that at least one of the following is true:

$$c_{x_i} \simeq 0 \bigwedge f_{x_i'} \Downarrow c_{x_i'} = nil \qquad (line\ b4) \tag{A.1}$$

$$c_{x_i'} \simeq 0 \bigwedge f_{x_i} \Downarrow c_{x_i} = nil \qquad (line\ b5) \tag{A.2}$$

$$f_{x_i'} \Downarrow c_{x_i'} = nil \bigwedge f_{x_i} \Downarrow c_{x_i} = nil \quad (line\ b10 \neq nil) \tag{A.3}$$

  apply *IH* to above cases, we get:

$$c_{x_i} \simeq 0 \bigwedge \hat{c}_{x_i'} = 0 \tag{A.4}$$

$$c_{x_i'} \simeq 0 \bigwedge \hat{c}_{x_i} = 0 \tag{A.5}$$

$$\hat{c}_{x_i} = 0 \bigwedge \hat{c}_{x_i'} = 0 \tag{A.6}$$

  Since $c_{x_i} \simeq 0 \rightarrow \hat{c}_{x_i} = 0$ and $c_{x_i'} = 0 \rightarrow \hat{c}_{x_i'} = 0$, (4) and (5) rewrite to (6), which is equivalent to $\hat{c} = 0$, hence "$\leftarrow$".

  "$\rightarrow$":

- **Base1**: $c \simeq 0 \rightarrow f \Downarrow c = nil \rightarrow$ "$\rightarrow$".
- **Base2**: $c \simeq 1 \rightarrow \hat{c} \neq nil \rightarrow$ "$\rightarrow$".
- **IH**: $\hat{c}_{x_i} = 0 \rightarrow f_{x_i} \Downarrow c_{x_i} = nil \bigwedge \hat{c}_{x_i'} = 0 \rightarrow f_{x_i'} \Downarrow c_{x_i'} = nil$

- **IS**: $\hat{c} = 0 \rightarrow f \Downarrow c = nil$

  Since $\hat{c} = 0 \rightarrow \hat{c}_{x_i} = \hat{c}_{x_i'} = 0$, applying *IH*, we get:

  $f_{x_i} \Downarrow c_{x_i} \ = \ f_{x_i'} \Downarrow c_{x_i'} \ = \ nil$

  Excluding the returns on lines *b1* and *b2* which are the base cases, the recursion $f \Downarrow c$ returns only at one of the lines *b4*, *b5*, and *b8*, all of which return nil.

■

Lemma A.2 $\hat{c} \neq 0 \ \bigwedge \ (f = 0 \| f = 1) \rightarrow f \Downarrow c = f$

Proof: Assume $\hat{c} \neq 0$. Let's first look at the case $f = 1$. Since $1_{x_i} = 1_{x_i'} = 1$, all of the f arguments in the successive recursions of $f \Downarrow c$ will be 1, therefore the final return value will be nil (line *b1*) or 1 (line *b2*) or the recursive composition of $x_i \cdot 1 + x_i' \cdot 1$, which is also 1. Since $\hat{c} \neq 0$, by Lemma A.1, $f \Downarrow c \neq nil$, therefore, $f \Downarrow c = 1$. The case $f = 0$ is similar. ■

Theorem A.1 $\hat{c} \neq 0 \rightarrow f \Downarrow c = f \downarrow \hat{c}$

Proof: We do induction on $f \Downarrow c$ to show that the theorem, which we refer to as p, is true.

- **Base1**: $c \simeq 0$

  $c \simeq 0 \rightarrow \hat{c} = 0 \rightarrow p$

- **Base2**: $c \simeq 1$

  $c \simeq 1 \rightarrow \hat{c} = 1 \rightarrow f \Downarrow c = f \downarrow \hat{c} = f \rightarrow p$.

- **IH**: $\hat{c}_{x_i} \neq 0 \rightarrow f_{x_i} \downarrow \hat{c}_{x_i} = f_{x_i} \Downarrow c_{x_i} \ \bigwedge \ \hat{c}_{x_i^{'}} \neq 0 \rightarrow f_{x_i^{'}} \downarrow \hat{c}_{x_i^{'}} = f_{x_i^{'}} \Downarrow c_{x_i^{'}}$

- **IS**: $\hat{c} \neq 0 \rightarrow f \Downarrow c = f \downarrow \hat{c}$

  The top variable x_i of c can be in or not in the support of $\hat{c}$.

  - **case I**: x_i is not in the support of $\hat{c}$

    In this case, $\hat{c}_{x_i} = \hat{c}_{x_i^{'}} = \hat{c} \neq 0$, therefore $c_{x_i} \not\simeq 0$ and $c_{x_i^{'}} \not\simeq 0$, hence $f \Downarrow c$ doesn't return on lines *b4* and *b5*. And from Lemma A.1, t and e can't be nil at lines *b8* and *b9*. Therefore, $f \Downarrow c$ returns only at line *b10*, i.e.,

$$\begin{aligned}
f \Downarrow c &= x_i(f_{x_i} \Downarrow c_{x_i}) + x_i^{'}(f_{x_i^{'}} \Downarrow c_{x_i^{'}}) \\
&= x_i(f_{x_i} \downarrow \hat{c}_{x_i}) + x_i^{'}(f_{x_i^{'}} \downarrow \hat{c}_{x_i^{'}}) \ ;; IH \\
&= (x_i \downarrow, \hat{c})(f_{x_i} \downarrow \hat{c}) + (x_i^{'} \downarrow, \hat{c})(f_{x_i^{'}} \downarrow \hat{c}) \ ;; x_i \notin sup(\hat{c}) \\
&= (x_i(f_{x_i})) \downarrow \hat{c} + (x_i^{'}(f_{x_i^{'}})) \downarrow \hat{c} \ ;; property\ of\ \downarrow \\
&= (x_i f_{x_i} + x_i^{'} f_{x_i^{'}}) \downarrow \hat{c} \ ;; property\ of\ \downarrow \\
&= f \downarrow \hat{c}
\end{aligned}$$

    Hence, p.

  - **case II**: x_i is in the support of $\hat{c}$

    this allows $f \Downarrow c$ to return at any line of *b4,5,8,9,10* (ref. Figure A.2).

Let's analyze those lines to show that $f \Downarrow c = f \downarrow \hat{c}$ in all the cases, therefore p holds.

* **case II(b4)**: $c_{x_i} \simeq 0$

  $f \Downarrow c = f_{x_i'} \Downarrow c_{x_i'}$;

  also $c_{x_i} \simeq 0 \rightarrow \hat{c}_{x_i} = 0 \rightarrow f \downarrow \hat{c} = f_{x_i'} \downarrow \hat{c}_{x_i'}$, by *IH*, $f \Downarrow c = f \downarrow \hat{c}$.

* **case II(b5)**: $c_{x_i} \not\simeq 0 \ \bigwedge \ c_{x_i'} \simeq 0$

  similar to case II(b4).

* **case II(b8)**: $c_{x_i} \not\simeq 0 \ \bigwedge \ c_{x_i'} \not\simeq 0 \ \bigwedge \ t = nil$

  $f \Downarrow c = f_{x_i'} \Downarrow c_{x_i'}$;

  $t = nil$ and Lemma A.1 $\rightarrow \hat{c}_{x_i} = 0$, therefore

  $f \downarrow \hat{c} = f_{x_i'} \downarrow \hat{c}_{x_i'}$, by *IH*

  $f \Downarrow c = f \downarrow \hat{c}$.

* **case II(b9)**: $c_{x_i} \not\simeq 0 \ \bigwedge \ c_{x_i'} \not\simeq 0 \ \bigwedge \ t \neq nil \ \bigwedge \ e = nil$

  similar to case II(b8).

* **case II(b10)**: $c_{x_i} \not\simeq 0 \ \bigwedge \ c_{x_i'} \not\simeq 0 \ \bigwedge \ t \neq nil \ \bigwedge \ e \neq nil$

$$f \Downarrow c = x_i(f_{x_i} \Downarrow c_{x_i}) + x_i^{'}(f_{x_i'} \Downarrow c_{x_i'}) \quad \text{(A.7)}$$

$$= x_i(f_{x_i} \downarrow \hat{c}_{x_i}) + x_i^{'}(f_{x_i'} \downarrow \hat{c}_{x_i'}) \ ;; IH \quad \text{(A.8)}$$

  Now check what $f \downarrow \hat{c}$ can return (ref. Figure A.1):

  · *a2*: $\hat{c} = 1 \ iff \ c \simeq 1$, while $c \simeq 1$ is the base case already discussed, therefore $\hat{c} \neq 1$, and *a2* returns $f, f = 0/1$. By Lemma A.2, "$f \Downarrow c = f$, when $f = 0/1$", hence, $f \Downarrow c = f \downarrow \hat{c}$.

- *a4*: since $t, i.e., f_{x_i} \Downarrow c_{x_i}$, is not nil, by Lemma A.1, $\hat{c}_{x_i} \neq 0$, therefore *a4* doesn't return.
- *a5*: similar to *a4*, *a5* doesn't return.
- *a6*: the line is exactly (14) above, therefore, $f \Downarrow c = f \downarrow \hat{c}$.

■

Appendix B

Generalized Cofactoring for State-dependent Constraints

In Section 8.4.1, we described Coudert and Madre's method of generating simulation vectors through generalized cofactoring for constraints that depends only on input variables. Here, we extend the method by introducing a new generalized cofactoring operator, written $\downarrow_s$, to handle state-dependent constraints. We will restrict us to the line of thinking in Section 8.4.1, where the functions to which $\downarrow_s$ is applied are in the form $f_i = u_i$ where u_i is an input variable.

The operator $\downarrow_s$ takes six arguments f, c, f^0, c^0, α, and β. Respectively, the first two arguments are the function and constraint for the current recursion; the next two are the original function and constraint; and the last two are input and state cubes corresponding to the path leading to the current recursion.

Figure B.1 gives the definition of $\downarrow_s$, wherein function $Find\_Closest\_Match$, given in Figure B.2, is used to find a match α' for the input cube α that is legal (satisfying the constraint in the current recursion) under the state β, and that the distance between α' and α, as given in Definition 8.1, is minimum. The function returns the cofactors of c^0 and f^0 with respect to the new path (α', β). Such matches do not exist under illegal states, for which we are not obligated to generate any vectors;

this is reflected by the two don't care assignments on lines *c13* and *c17*.

Theorem B.1 For any legal state β, $\downarrow_s \ (f, c, f, c, 1, 1)$ computes the generalized cofactor of f with respect to c_β.

Proof: (Sketch) The key difference between $\downarrow_s$ and $\downarrow$ (generalized cofactor) is that when the current path (α, β) does not satisfy c, the former finds the "shortest-distance" match for α in c_β, instead of a match for the whole path in c. Therefore, the functionality of generalized cofactoring is preserved for the input vectors only, for all legal states. ■

Since illegal states can be easily detected by checking if the constraint evaluates to zero under the current state, the operator $\downarrow_s$ can be used in place of $\downarrow$ in Lemma 8.1 to generate input vectors from state-dependent constraints. However, due to multiple arguments that need to be carried along the recursions, as required by the $Find\_nearest\_match$ function, cache hit rate will be very low for the above algorithm, making it computationally expensive.

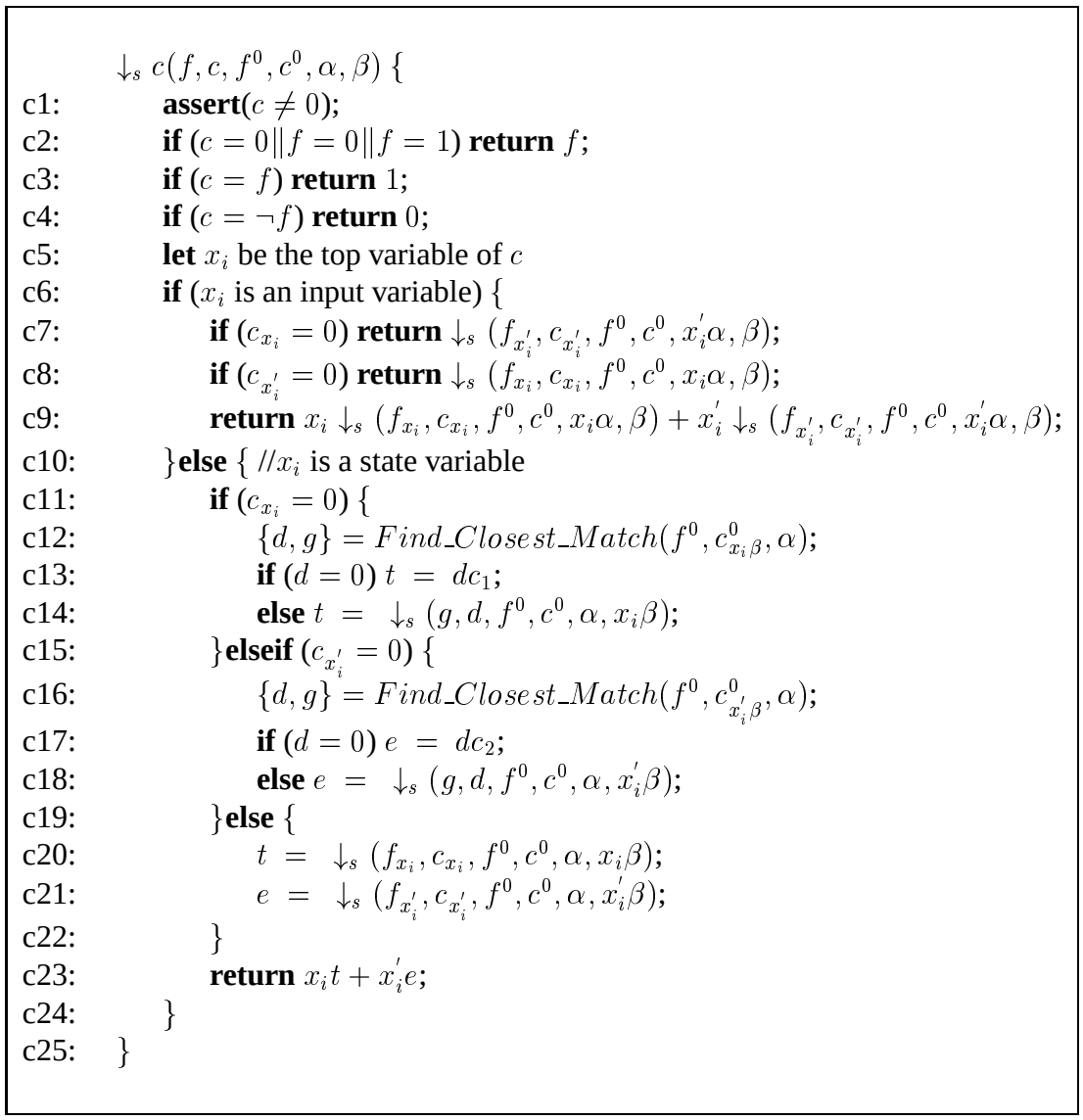


Figure B.1: Generalized cofactor with respect to state-dependent constraints

```
      Find_nearest_match(f, c, α) {
d1:       if (c = 1) return {f, c};
d2:       if (c = 0) return {0, 0};
d3:       let l be the top literal of α;
d4:       if ((c_l ≠ 0)
d5:           return Find_nearest_match(f_l, c_l, α_l);
d6:       else
d7:           return Find_nearest_match(f_¬l, c_¬l, α_¬l);
d8:   }
```

Figure B.2: Find the nearest match

Bibliography

[1] M. Aagaard, R. B. Jones, and C.-J. H. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In *Proceedings of the Design Automation Conference*, pages 538–541, 1998.

[2] A. Aharon, A. Bar-David, B. Dorfman, E. Gofman, M. Leibowitz, and V. Schwartzburd. RTPG-A Dynamic Biased Pseudo-Random Test Program Generator for Processor Verification. *IBM Technical Report 88.290*, July 1990.

[3] A. Aharon, A. Bar-David, B. Dorfman, E. Gofman, M. Leibowitz, and V. Schwartzburd. Verification of the IBM RISC System/6000 by a Dynamic Biased Pseudo-random Test Program Generator. *IBM Systems Journal*, 30(4):527–538, July 1991.

[4] S. B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-37:509–516, June 1978.

[5] P. Ashar and S. Malik. Fast Functional Simulation Using Branching Programs. In *Proceedings of International Conference on Computer-Aided Design*, November 1995.

[6] A. Aziz, F. Balarin, V. Singhal, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. The Temporal Logic of Stochastic Systems. In *Proceedings of the Computer Aided Verification Conference*, July 1995.

[7] A. Aziz, J. Kukula, T. Shiple, and J. Yuan. Efficient Control State Space Search. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2, October 2001.

[8] A. Aziz, V. Singhal, G. M. Swamy, and R. K. Brayton. Minimizing Interacting Finite State Machines: A Compositional Approach to the Language Containment Problem. In *Proceedings of International Conference on Computer Design*, pages 255–261, October 1994.

[9] A. Aziz, S. Tasiran, and R. K. Brayton. BDD Variable Ordering for Interacting Finite State Machines. In *Proceedings of the Design Automation Conference*, June 1994.

[10] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and their Applications . In *Proceedings of International Conference on Computer-Aided Design*, pages 188–192, 1993.

[11] D. L. Beatty. A methodology for formal hardware verification with application to microprocessors. In *Ph.D Thesis, published as technical report CMU-CS-93-190, School of Computer Science, Carnegie Mellon University*, August 1993.

[12] B. Beizer. The pentium bug, an industry watershed. In *Testing Techniques Newsletter On-Line Edition*, September 1995.

[13] T. Bengtsson, A. Martinelli, and E. Dubrova. A Fast Heuristic Algorithm for Disjoint Decomposition of Boolean Functions. *Proceedings of International Workshop on Logic Synthesis*, pages 51–55, 2002.

[14] V. Bertacco and M. Damiani. The Disjunctive Decomposition of Logic Functions. *Proceedings of International Conference on Computer-Aided Design*, pages 78–82, 1997.

[15] M. Blum, A. Chandra, and M. Wegman. Equivalence of Free Boolean Graphs Can Be Decided Probabilistically in Polynomial Time. In *Information Processing Letters*, pages 10:80–82, 1980.

[16] D. Bochmann, F. Dresig, and B. Steinbach. A New Decomposition Method for Multilevel Circuit Design. *Proceedings of European Design Automation Conference*, pages 374–377, 1991.

[17] B. Bollig, M. Löbbing, and I. Wegener. Simulated annealing to improve variable orderings for OBDDs. *Proceedings of International Workshop on Logic Synthesis*, pages 5b:5.1–5.10, 1995.

[18] S. Bose and A. Fisher. Verifying Pipelined Hardware using Symbolic Logic Simulation. In *Proceedings of International Conference on Computer Design*, pages 217–221, 1989.

[19] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Academic Press, New York, 1988.

[20] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: A system for Verification and Synthesis. In *Proceedings of the Computer Aided Verification Conference*, July 1996.

[21] R. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35:677–691, August 1986.

[22] R. Bryant and Y.A. Chen. Verification of Arithmetic Circuits with Binary Moment Diagrams. In *Proceedings of the Design Automation Conference*, pages 535–541, June 1995.

[23] R. E. Bryant. Formal verification of memory circuits by switch level simulation. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 10, No. 1, pages 94–102, January 1991.

[24] R. E. Bryant, D. L. Beatty, and C.-J. H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *Proceedings of the Design Automation Conference*, pages 397–402, June 1991.

[25] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[26] K. M. Butler, D. E. Ross, R. Kapur, and M. R. Mercer. Heuristics to compute variable ordering for the efficient manipulation of binary decision dia-

grams. *Proceedings of the Design Automation Conference*, pages 417–420, June 1991.

[27] CADENCE. In *Verilog-XL User Reference*.

[28] A. K. Chandra and V. S. Iyengar. Constraint Solving for Test Case Generation - A Technique for High Level Design Verification. In *Proceedings of International Conference on Computer Design*, pages 245–248, 1992.

[29] H. Cho, G. D. Hachtel, E. Macii, M. Poncino, and F. Somenzi. A Structural Approach to State Space Decomposition for Approximate Reachability Analysis. In *Proceedings of International Conference on Computer Design*, October 1994.

[30] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching Time Logic. In *Proceedings of Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[31] Toshiba Corporation. http://www.toshiba.com.

[32] O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Based on Symbolic Execution. In J. Sifakis, editor, *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373, June 1989.

[33] O. Coudert and J. C. Madre. Verification of Sequential Machines Using Functional Boolean Vectors. In *Proceedings of the IFIP International Workshop, Applied Formal Methods for Correct VLSI Design*, November 1989.

[34] O. Coudert and J. C. Madre. A Unified Framework for the Formal Verification of Sequential Circuits. In *Proceedings of International Conference on Computer-Aided Design*, pages 126–129, November 1990.

[35] M. Davis and H. Putman. A Computing Procedure for Quantification Theory. In *Journal of the Association for Computing Machinery*, pages 201–215, July 1960.

[36] R. Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410, 1984.

[37] S. Devadas, A. Ghosh, and K Keutzer. An observablity-based code coverage metric for functional simulation. *Proceedings of the Design Automation Conference*, pages 418–425, 1996.

[38] D.Geist, G.Brian, T.Arons, M.Slavkin, Y.Nustov, M.Farkas, K.Holtz, A.Long, D.king, and S.Barret. A Methodology For the Verification of a "System On Chip". *Proceedings of the Design Automation Conference*, pages 574–579, 1999.

[39] R. Drechsler, B. Becker, and G$\acute{}$ockel. A genetic algorithm for variable ordering of OBDDs. *Proceedings of International Workshop on Logic Synthesis*, pages 5c:5.55–5.64, 1995.

[40] E. V. Dubrova, J. C. Muzio, and B. von Stengel. Finding Composition Trees for Multiple-valued Functions. *Proceedings of the 27th International Symposium on Multiple-Valued Logic (ISMVL '97)*, pages 19–26, 1997.

[41] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, pages 996–1072. Elsevier Science, 1990.

[42] E. A. Emerson and E. M. Clarke. Characterizing Correctness Properties of Parallel Programs as Fixpoints. (85), 1981.

[43] E. A. Emerson and C. L. Lei. Modalities for Model Checking: Branching Time Strikes Back. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 84–96, 1985.

[44] E. A. Emerson and C. L. Lei. Efficient model checking in fragments of the propositional mu-calculus(extended abstract). In *Proceedings of IEEE Symposium on Logic in Computer Science*, pages 267–278, 1986.

[45] F. Somenzi et al. CUDD: CU Decision Diagram Package. In *ftp://vlsi.colorado.edu/pub/*.

[46] A. Evans, A.Silburt, G.Vrckovnik, T.Brown, M.Dufresne, G.Hall, T.Ho, and Y.liu. Functional Verification of Large ASICs. In *Proceedings of the Design Automation Conference*, pages 650–655, 1998.

[47] Karem A. Sakallah Fadi A. Aloul, Igor L. Markov. Mince: A static global variable-ordering for sat and bdd. In *Proceedings of International Workshop on Logic Synthesis*, June 2001.

[48] F.Casaubieilh, A.McIsaac, M.Benjamin, M.Bartly, F.Pogodalla, F.Rochetean, M.Belhadj, J.Eggleton, G.Mas, G.Barrett, and C.Berthet. Functional Verification Methodology of Chameleon Processor. In *Proceedings of the Design Automation Conference*, 1996.

[49] E. Felt, G. York, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Dynamic variable reordering for BDD minimization. *Proceedings of European Conference on Design Automation*, pages 130–135, 1993.

[50] J. Freeman, R. Duerden, C. Taylor, and M. Miller. The 68060 microprocessor functional design and verification methodology. In *On-Chip Systems Design Conference*, pages 10.1–10.14, 1995.

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

[52] H. Fujii, G. Ootomo, and C. Hori. Interleaving Variable Ordering Methods for Ordered Binary Decision Diagrams. In *Proceedings of International Conference on Computer-Aided Design*, pages 38–41, November 1993.

[53] M. Fujita, Y. Tamiya, Y. Kukimoto, and K.-C. Chen. Application of Boolean Unification to Combinational Logic Synthesis. *Proceedings of International Conference on Computer-Aided Design*, pages 510–513, 1991.

[54] N. Ganguly, M. S. Abadir, and M. Pandey. Powerpc(tm) array verification methodology using formal techniques. In *Proceedings of International Test*

Conference, pages 857–864, 1996.

[55] M. Gordon. HOL: A Proof Generating System for Higher-order Logic. In G. Birwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–127. Academic Press, Boston, 1988.

[56] G. D. Hachtel, E. Machii, A. Pardo, and F. Somenzi. Symbolic Algorithms to Calculate Steady-State Probabilities of a Finit State Machine. In *The European Design and Test Conference*, pages 214–218, 1994.

[57] P.-H. Ho, A. J. Isles, and T. Kam. Formal verification of pipeline control using controlled token nets and abstract interpretation. In *Proceedings of International Conference on Computer-Aided Design*, pages 529–536, 1998.

[58] Richard C. Ho, C. Han Yang, Mark A. Horowitz, and David L. Dill. Architectural Validation for Processors. In *Proceedings of the International Symposium on Computer Architecture*, June 1995.

[59] R. Hojati, S. C. Krishnan, and R. K. Brayton. Early Quantification and Partitioned Transition Relations. *Proceedings of International Conference on Computer Design*, pages 12–19, October 1996.

[60] Y. Hoskote, D. Moundanos, and J. Abraham. Automatic Extraction of the Control Flow Machine and Application to Evaluating Coverage of Verification Vectors. In *Proceedings of International Conference on Computer Design*, Austin, TX, October 1995.

[61] Y. Hoskote, D. Moundanos, and J. A. Abraham. Automatic Extraction of the Control Flow Machine and Application to Evaluating Coverage of Verification Vectors. *Proceedings of International Conference on Computer Design*, pages 532–537, October 1995.

[62] C. Norris Ip and David L. Dill. Verifying systems with replicated components in murϕ. In *Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 147–158. Springer-Verlag, July 1996.

[63] H. Iwashita, T. Nakata, and F. Hirose. CTL Model Checking Based on Forward State Traversal. *Proceedings of International Conference on Computer-Aided Design*, pages 82–87, 1996.

[64] Robert B. Jones, David L. Dill, and Jerry R. Burch. Efficient validity checking for processor verification. In *International Conference on Computer-Aided Design (ICCAD)*, pages 2–6. IEEE Computer Society Press, November 1995.

[65] G. Kamhi, O. Weissberg, and L. Fix. Automatic datapath extraction for efficient usage of hdd. In *Proceedings of the Computer Aided Verification Conference*, pages 95–106, 1997.

[66] M. Kaufmann, A. Martin, and C. Pixley. Design Constraints in Symbolic Model Checking. In *Proceedings of the Computer Aided Verification Conference*, 1998.

[67] M. Kaufmann and J. S. Moore. Acl2: An industrial strength version of nqthm. In *Proceedings of the 11th Annual Conference on Computer Assurance, IEEE Computer Society press*, pages 23–34, June 1996.

[68] D. Koller and N. Megiddo. Constructing small sample spaces satisfying given constraints. *SIAM Journal on Discrete Mathematics*, pages 260–274, 1994.

[69] R. Krieger, B. Becker, and R. Sinkovic. A BDD-based Algorithm for Computation of Exact Fault Detection Probabilities. In *International Symposium on Fault-Tolerant Computing*, pages 186–195, 1993.

[70] J. Kumar, N. Strader, J. Freeman, and M. Miller. Emulation Verification of the Motorola 68060. In *Proceedings of International Conference on Computer Design*, 1995.

[71] R. P. Kurshan. Reducibility in Analysis of Coordination. In *Discrete Event Systems: Models and Applications*, volume 103 of *LNCS*, pages 19–39. Springer-Verlag, 1987.

[72] R. P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes*. Princeton University Press, 1993.

[73] R. P. Kurshan. Computer-aided verification of coordinating processes: The automata-theoretic approach. In *Princeton University Press*, 1994.

[74] C. Y. Lee. Representation of switching circuits by binary-decision programs. In *Bell System Technical Journal*, volume 38, No. 4, pages 985–999, July 1959.

[75] B. Lin and R. Newton. Implicit Manipulation of Equivalence Classes Using Binary Decision Diagrams. In *Proceedings of International Conference on Computer Design*, Cambridge, MA, October 1991.

[76] B. Lin, H. J. Touati, and A. R. Newton. Don't Care Minimization of Multi-level Sequential Logic Networks. In *Proceedings of International Conference on Computer-Aided Design*, pages 414–417, November 1990.

[77] L. Löwenheim. Über das Auflösungsproblem im logischen Klassenkalkül. In *Sitzungsber. Berl. Math. Gessel. 7*, pages 89–94, 1908.

[78] T. Luba and H. Selvaraj. A Gneral Approach to Boolean Function Decomposition and its Application in FPGA-based Synthesis. *VLSI Design, Special Issue on Decompositions in VLSI Design*, 3(3-4):289–300, 1995.

[79] U. Martin and T. Nipkow. Boolean unification - the story so far. In *Journal of Symbolic Computation*, volume 7, pages 275–293, 1989.

[80] P. McGeer, K. McMillan, A. Saldanha, A. Sangiovanni-Vincentelli, and P. Scaglia. Fast Discrete Function Evaluation. In *Proceedings of International Conference on Computer-Aided Design*, November 1995.

[81] K. L. McMillan. A conjunctively decomposed boolean representation for symbolic model checking. *Proceedings of the Computer Aided Verification Conference*, 1996.

[82] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[83] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw Hill, 1994.

[84] A. Mishchenco, B. Steinbach, and M. Perkowski. An Algorithm for Bi-Decomposition of Logic Functions. *Proceedings of the Design Automation Conference*, 2001.

[85] J. Monaco, D. Holloway, and R. Raina. Functional Verification Methodology for the PowerPC 604 Microprocessor. In *Proceedings of the Design Automation Conference*, 1996.

[86] I. Moon, J. Jang, G. D. Hachtel, F. Somenzi, J. Yuan, and C. Pixley. Approximate Reachability Don't Cares for CTL Model Checking. *Proceedings of International Conference on Computer-Aided Design*, November 1998.

[87] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verificatino for fault-tolerant architectures: Prolegomena to the design of pvs. In *IEEE Transaction on Software Engineering*, volume 21(2), pages 107–125, February 1995.

[88] S. Panda and F. Somenzi. Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams. *Proceedings of International Conference on Computer-Aided Design*, 1994.

[89] M. Pandey, R. Raimi, R. E. Bryant, and M. S. Abadir. Formal verification of content addressable memories using symbolic trajectory. In *Proceedings of the Design Automation Conference*, pages 167–172, 1997.

[90] D. Park. Fixpoint Induction and Proof of Program Semantics. 5:59–78, 1970.

[91] C. Pixley. A Theory and Implementation of Sequential Hardware Equivalence. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(12):1469–1494, December 1992.

[92] C. Pixley, K. Shultz, and J. Yuan. Integrated Formal and Informal Design Verification of Commercial Integrated Circuits. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 1061–1067, June 1999.

[93] C. Pixley, N. R. Strader, W. C. Bruce, J. Park, M. Kaufmann, K. Shultz, M. Burns, J. Kumar, J. Yuan, and J. Nguyen. Commercial Design Verification: Methodology and Tools. In *Proceedings of International Test Conference*, 1997.

[94] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In *Proceedings of the Computer Aided*

Verification Conference, pages 84–97, July 1995.

[95] R. K. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R. K. Brayton. Efficient BDD Algorithms for FSM Synthesis and Verification. *Proceedings of International Workshop on Logic Synthesis*, May 1995.

[96] K. Ravi and F. Somenzi. High Density Reachability Analysis. In *Proceedings of International Conference on Computer-Aided Design*, Santa Clara, CA, November 1995.

[97] S. Rudeanu. Boolean Functions and Equations. 1974.

[98] R. Rudell. Dynamic Variable Ordering for Binary Decision Diagrams. In *Proceedings of International Conference on Computer-Aided Design*, pages 42–47, November 1993.

[99] H. Savoj, R. K. Brayton, and H. Touati. Extracting Local Don't Cares for Network Optimization. In *Proceedings of International Conference on Computer-Aided Design*, pages 514–517, November 1991.

[100] J. Shen and J. A. Abraham. Native Mode Functional Test Generation for Microprocessors with Applications to Self Test and Design Validation. *Proceedings of International Test Conference*, pages 990–999, October 1998.

[101] T. Shiple and J. Kukula. Building Circuits From Relations. In *Proceedings of the Computer Aided Verification Conference*, 2000.

[102] J. H. Siekmann. Universal unification. In *7th International Conference on Automated Deduction*, Lecture Notes in Computer Science, pages 1–42, 1984.

[103] Vigyan Singhal. *Design Replacements for Sequential Circuits*. PhD thesis, University of California Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, 1996.

[104] S. Taylor, M. Quinn, D. Brown, N.Bohm, S. Hildebrandt, J. Huggins, and C. Ramey. Functional Verification of a Multiple-issue, Out-of-Order, Superscalar Alpha Processor. In *Proceedings of the Design Automation Conference*, pages 638–643, 1998.

[105] S. M. Thatte and J. A. Abraham. Test Generation for Microprocessors. *IEEE Transactions on Computers*, C-29:429–441, June 1980.

[106] K. Thompson. Retrograde analysis of certain endgames. *ICCA Journal*, 9(3):131–139, 1986.

[107] H. Touati, R. K. Brayton, and R. P. Kurshan. Checking Language Containment using BDDs. In *Proceedings of International Workshop on Formal Methods in VLSI Design*, Miami, FL, January 1990.

[108] M. N. Velev and R. E. Bryant. Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic. In *CHARME*, pages 37–53, 1999.

[109] W. Büttner. Unification in finite algebras is unitary. In *Proceedings of CADE-9*, volume 310 of *Lecture Notes in Computer Science*, pages 368–377, 1988.

[110] W. Büttner and H. Simonis. Embedding boolean expressions into logic programming. In *Journal of Symbolic Computation*, volume 4, pages 191–205, 1987.

[111] B. Yang, R. Simmons, R.R. Bryant, and D.R. O'Hallaron. Optimizing Symbolic Model Checking for Constraint-rich Models. *Proceedings of the Computer Aided Verification Conference*, 1999.

[112] J. Yuan, K. Albin, A. Aziz, and C. Pixley. Simplifying constraint solving in random simulation generation. In *Proceedings of International Workshop on Logic Synthesis*, pages 185–189, June 2002.

[113] J. Yuan and A. Aziz. Random vector generation using event probabilities. *Technical report*, 2000.

[114] J. Yuan, A. Aziz, and K. Albin. Enhancing simulation coverage through guided vector generation. *Technical report*, 2002.

[115] J. Yuan, A. Aziz K. Albin, and C. Pixley. Simplifying boolean constraint solving for random simulation-vector generation. *Proceedings of International Conference on Computer-Aided Design*, 2002. to appear.

[116] J. Yuan, J. Shen, J. A. Abraham, and A. Aziz. On Combining Formal and Informal Verification. In *Proceedings of the Computer Aided Verification*

Conference, Lecture Notes in Computer Science, pages 376–387. Springer-Verlag, June 1997.

[117] J. Yuan, K. Shultz, J. Havlicek, K. Albin, and A. Aziz. A method for synthesizing boolean constraints. In *Proceedings of International Workshop on Logic Synthesis*, pages 351–353, June 2002.

[118] J. Yuan, K. Shultz, C. Pixley, and H. Miller. A tool for Automatically Generating Simulation Environments from Constraints. *Proceedings of the ITC Microprocessor Test and Verification Workshop*, October 1998.

[119] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz. Modeling Design Constraints and Biasing in Simulation Using BDDs. *Proceedings of International Conference on Computer-Aided Design*, pages 584–589, 1999.

[120] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz. Automatic Simulation Generation Using Constraints and Biasing. *Journal of Electronic Testing: Theory and Applications*, pages 107–120, 2000.

Vita

Jun Yuan was born in Chongqing, China, the son of Wei Yuan and Ping Lin. He received the degree of Bachelor of Science in Engineering at Tsinghua University, Beijing, China in 1989, in the area of Automotive Engineering. He received the degree of Master of Science in Engineering at University of Texas at Austin in 1995, in the area of Computer Engineering. From 1989 to 1991, he was employed by Sichuan Biomedical Engineering R&D Center in Chengdu, China. From 1994 to 1995, he worked in Advanced Micro Devices Inc. in Austin, Texas. He has been employed by the Motorola Corporation, in Austin, Texas, since 1995.

Permanent address: 6911 Gentle Oak Drive
Austin, TX 78749

This dissertation was typeset with LaTeX† by the author.

†LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.