Formal Methods in Computer-Aided Design

by

Hratch Mangassarian

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Formal Methods in Computer-Aided Design

Hratch Mangassarian

Doctor of Philosophy

Graduate Department of Electrical and Computer Engineering

University of Toronto

2012

The VLSI CAD flow encompasses an abundance of critical NP-complete and PSPACE-complete problems. Instead of developing a dedicated algorithm for each, the trend during the last decade has been to encode them in formal languages, such as Boolean satisfiability (SAT) and quantified Boolean formulas (QBFs), and focus academic resources on improving SAT and QBF solvers. The significant progress of these solvers has validated this strategy. This dissertation contributes to the further advancement of formal techniques in CAD.

Today, the verification and debugging of increasingly complex RTL designs can consume up to 70% of the VLSI design cycle. In particular, *RTL debug* is a manual, resource-intensive task in the industry. The first contribution of this thesis is an in-depth examination of the factors affecting the theoretical computational complexity of debugging. It is established that most variations of the debugging problem are NP-complete.

Automated debugging tools return all potential error sources in the RTL, called *solutions*, that can explain a given failing error trace. Finding each solution requires a separate call to a formal engine, which is computationally expensive. The second contribution of this dissertation comprises techniques for reducing the number of such iterations, by leveraging dominance relationships between RTL blocks to *imply solutions*. Extensive experiments on industrial designs show a three-fold reduction in the number of formal engine calls due to solution implications, resulting in a 1.64x overall speed-up.

The third contribution aims to advance the state-of-the-art of QBF solvers, whose progress has not been as impressive as that of SAT solvers. We present a framework for using *complete dominators* to preprocess and reduce QBFs with an inherent circuit structure, which is com-

mon in encodings of PSPACE-complete CAD problems. Experiments show that three modern QBF solvers together solve 55% of preprocessed QBF instances, compared to none without preprocessing.

The final contribution consists of a series of QBF encodings for evaluating the reconfigurability of partially programmable circuits (PPCs). The metrics of *fault tolerance*, *design error tolerance* and *engineering change coverage* are defined for PPCs and encoded using QBFs. These formulations along with experimental results demonstrate the theoretical and practical appropriateness of QBFs for dealing with reconfigurability.

Acknowledgements

First and foremost, I would like to convey my sincere gratitude to my Ph.D. supervisor, Professor Andreas Veneris, for his conscientious guidance and for being both a mentor and a friend.

I would like to acknowledge my Ph.D. committee members, Professors Farid Najm, Jason Anderson, Zeljko Zilic and Andreas Moshovos for their thorough reviews of my dissertation and their insightful feedback.

I am indebted to several colleagues at the University of Toronto for our productive discussions, as well as their technical contributions to my research. Special thanks to (in alphabetical order of last names) Alexandra Goultiaeva, Brian Keng, Bao Le, Sean Safarpour and Duncan Smith. I would also like to thank my family and friends for their unwavering support.

Finally, I would like to express my gratitude to the University of Toronto and the Ontario Graduate Scholarship (OGS) for their financial support of my research.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The semiconductor industry has products pervading most commercial and consumer markets. Its growth is driven by a constant demand for electronic devices with continuously expanding functionalities, better performance and lower power consumption. With the reducing feature size of modern integrated circuits (ICs), the number of transistors in the chips used in these devices is already in the billions. The design and verification of such complex very large scale integration (VLSI) systems has been made possible by steady advances in computer-aided design (CAD) tools.

Figure 1.1 [109] shows the major stages in a CAD flow for the fabrication of a VLSI chip. The behavioral specification of the design, written in C or a behavioral hardware description language (HDL), is first compiled into a register transfer level (RTL) description given in a structural HDL such as VHDL or Verilog. This is synthesized into a gate-level circuit, which is optimized and analyzed. A transistor-level netlist is then created, which is placed and routed. The physical layout is finally sent to a fabrication plant for silicon manufacturing.

Each design stage is followed by a verification step. Functional verification techniques such as model checking ensure that the behavior of the synthesized design corresponds to its specification. After each optimization iteration in logic synthesis, equivalence checking guarantees that no new errors have been inserted into the design. Timing simulations are

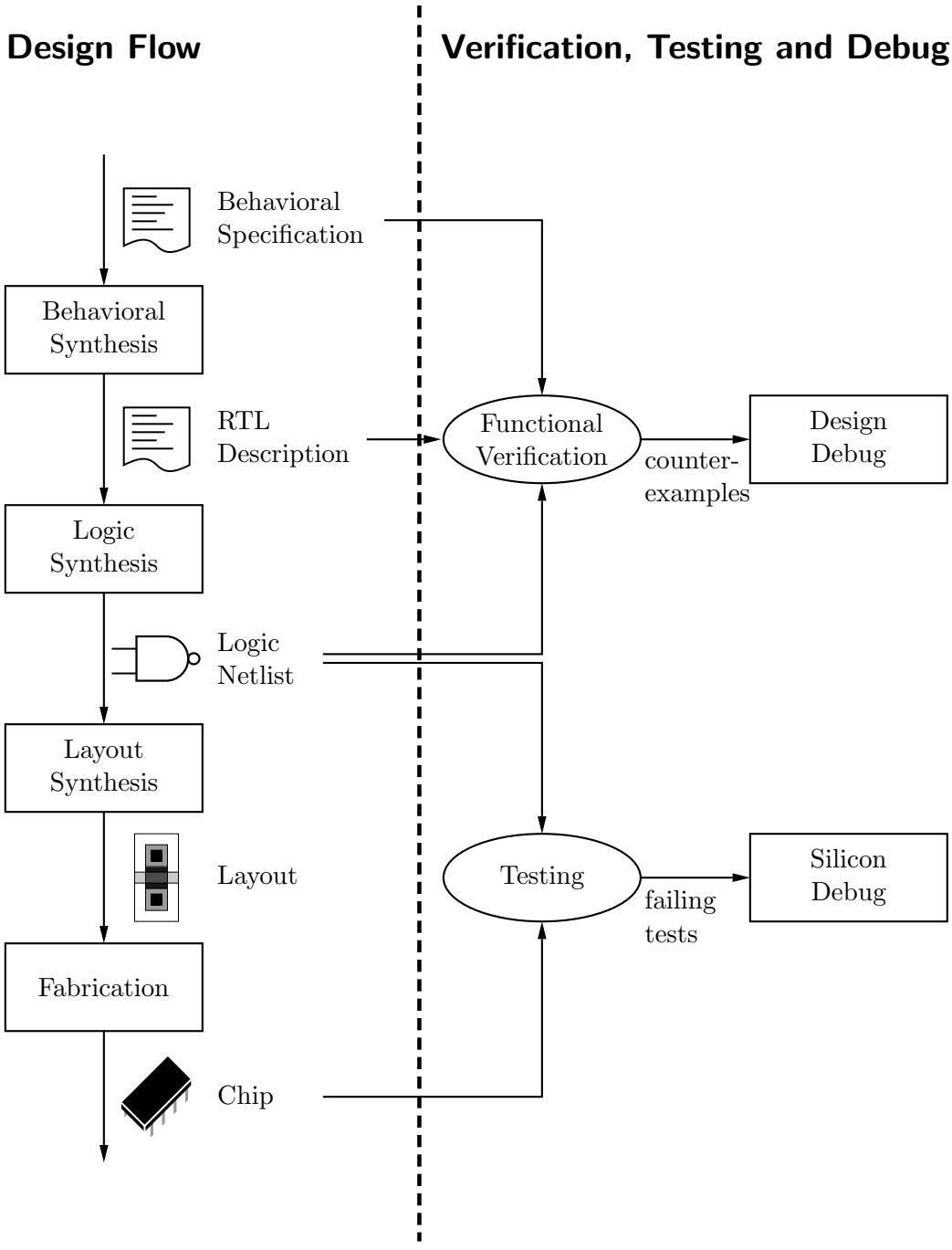


Figure 1.1: VLSI CAD flow

carried out on the transistor netlist to verify that the design conforms to its performance requirements. Finally, testing is performed after fabrication, before the chips are shipped. In the event where any of these verification stages fails, the design or chip is debugged in an effort to locate the root cause of the failure. If a failure is found during functional verification, design debugging, or *RTL debug*, is performed to identify the bug(s) in the RTL. On the other hand, if chip-level testing fails, *silicon debug* is performed.

Many of the stages shown in Figure 1.1, both on the design and on the verification side, involve problems that are known to be NP-complete, co-NP-complete or PSPACE-complete. For instance, field-programmable gate array (FPGA) routing [118], combinational automatic test pattern generation (ATPG) [57], combinational equivalence checking (CEC) [49], peak power estimation in combinational circuits [43], and discrete gate sizing [90] are all NP-complete or co-NP-complete. On the other hand, model checking (MC) for properties given in linear temporal logic (LTL) [108], sequential equivalence checking (SEC) [12] and sequential power estimation [43] are PSPACE-complete. Instead of developing a dedicated algorithm for each of these problems, the trend during the last decade has been to encode them in *formal languages*, such as Boolean satisfiability (SAT), quantified Boolean formulas (QBF), maximum satisfiability (MAX-SAT) and satisfiability modulo theories (SMT), and solve them using their respective solvers. This strategy allows the academic community to focus its resources on improving a handful of formal solvers, rather than tens or hundreds of specialized algorithms. The tremendous improvements in these solvers have validated this approach.

SAT is considered the canonical NP-complete language, and all other NP-complete problems can be efficiently reduced to it. The significant advancements in SAT solvers over the last two decades [38, 84, 87] have stimulated a plethora of SAT formulations for NP-complete and NP-hard CAD problems and have led to the eventual integration of SAT engines in many CAD tools. One of the advantages of such a strategy is that any improvement to the SAT solver itself immediately benefits all SAT-based CAD solutions. This is especially relevant due to yearly SAT solver competitions [69], which consistently improve the state-of-the-art. Today, SAT solvers are used as underlying engines in logic synthesis [86], ATPG [67], CEC [50], MC [20], peak power estimation [80], timing analysis [107], FPGA routing [52, 88] and design debugging [110],

among other problems.

QBFs are a powerful extension of SAT, considered the canonical PSPACE-complete language. All other PSPACE-complete problems can be efficiently encoded as QBFs. Following the dramatic growth of SAT solvers, the last decade has also seen substantial advances in QBF solvers [13, 18, 47, 53, 74, 75, 102], with a yearly QBF solver competition [93] encouraging consistent improvements. Although the progress of QBF solvers has not been as impressive as that of SAT solvers, competitive QBF formulations have been developed for LTL MC [35, 59, 79], design debugging [79, 113], fault localization and correction [112] and FPGA logic synthesis [72], among other problems. An overview of QBF-based formal verification techniques is given in [17]. Again, any improvement to the QBF solver would directly benefit all QBF-based approaches for CAD.

1.1.1 Design Debugging

With the growing size and complexity of VLSI designs, the required verification effort has increased disproportionately [42]. Significant portions of engineering time and resources are devoted to achieving functional correctness [2]. As a result, the ratio of verification engineers to designers in the industry reaches $2:1$ for complex designs [2]. To make things worse, this trend has been projected to increase almost seven-fold by 2015 [85]. Despite the allocation of inordinate amounts of resources to design verification, the latter can consume up to 70% of the VLSI design cycle [95].

Several functional verification methodologies have been developed by the academic and industrial communities in order to bridge the gap between our ability to design and to verify circuits, referred to as the *verification gap*. These techniques can be categorized into *simulation-based* and *formal* verification approaches. Mainstream verification strategies in industry today rely on logic simulations, where input patterns are applied repeatedly to exercise corner cases, and coverage metrics are used to provide confidence in the correctness of the design [2]. The principal drawback of this approach is that it is non-exhaustive and usually has low coverage. As such, it is unable to guarantee the correctness of the design. Nevertheless, due to their simplicity and relative scalability, simulation-based methodologies are widely adopted in the

industry. In contrast, formal verification techniques explore the design space exhaustively by employing formal engines such as SAT and QBF solvers. This enables them to prove that the design conforms to its specification. However, in practice, their applicability is limited to the block or the sub-system level due to their lack of scalability [2].

If a design does not pass verification, both formal and simulation-based verification tools return a *counter-example*, which consists of a sequence of input stimuli exhibiting a discrepancy between the actual and expected responses of the design and its specification, respectively. The *length* of a counter-example refers to the number of clock-cycles in the sequence of inputs producing a mismatch. Given a buggy design and a counter-example, *design debugging* is the process of tracking down the root cause of the observed erroneous behavior. Once a bug is localized in the RTL, engineers can rectify the design by appropriately modifying it at the given location and restarting the verification process to ensure that the fix is adequate. Today, design debugging is still a predominantly manual task in the CAD industry. It entails the manual analysis of long and complex counter-examples to identify the source of the problem [2]. Recent technical roadmaps and market studies suggest that once a design fails verification, debugging it and fixing it can consume up to 32% of the total verification effort [42].

Traditionally, engineers use waveform viewers and various graphical user interfaces to analyze counter-examples by hand. However, with counter-examples consisting of tens of thousands of clock-cycles and typical design blocks containing millions of gates, such time-consuming manual processes place an increasingly large burden on the engineer. Furthermore, bugs in complex interactions between heterogeneous components implemented at multiple levels of abstraction and using different languages are notoriously difficult to find by human inspection [2]. Finally, the interwoven ecosystem of intellectual property (IP) blocks in the design necessitates a sophisticated understanding of both the design and verification environments. With different blocks being designed by different groups that are sometimes in different geographical locations, this requirement is often too stringent for a single engineer to satisfy. As such, it is not uncommon for a failure to be passed from designer to designer until the error source is discovered [99], threatening the time-to-market of the product.

With the aim of alleviating the design debugging cost, several methodologies have been

proposed over the years to automate this process. Historically, the focus has been on *fault diagnosis* algorithms that are based on simulations and binary decision diagrams (BDDs) [3, 55, 56, 70, 73]. Due to the recent advances in formal tools, a new genre of satisfiability-based automated debugging methodologies has gained a competitive advantage [117]. These techniques reduce the design debugging problem into a satisfiability problem which can be solved using a formal engine. Over the years, the original SAT-based gate-level debugging formulation [111] has been extended to handle hierarchical RTL blocks [6], and its scalability and performance have improved significantly [6, 41, 61, 63, 79, 100, 110]. However, despite these advances, increasing design sizes and counter-example lengths still present a challenge to automated debugging techniques.

1.1.2 Circuit-based Quantified Boolean Formulas

Many CAD problems dealing with the analysis, optimization and verification of sequential circuits are PSPACE-complete [12, 43, 108]. As the state-space of sequential designs continues to grow, QBF offers a concise encoding alternative for these problems, which are currently being reduced to SAT at the expense of exponential-size worst-case formulations, due to the power of SAT solvers. Model checking is an example of a PSPACE-complete problem that is commonly translated to SAT [20, 21]. Significant advances in QBF solving procedures are still necessary in order to robustly handle QBF encodings of challenging CAD tasks.

In most modern QBF solvers, the problem constraints are given as a propositional formula in *conjunctive normal form* (CNF), which is essentially a product-of-sums representation. The standardization of the CNF input format has carried over from SAT to QBF due to the efficient data-structures and solving strategies developed for CNF-based SAT solvers, which are also used in many QBF solvers. However, QBF instances encoding CAD problems normally have an internal circuit structure, which is lost during the conversion into formulas in CNF. Several recent works [97, 122] have pointed out the inadequacies of CNF for QBF and have suggested a switch to alternative representations. Circuit representations have been used in SAT solvers [98, 123], and more recently, circuit-based QBF solvers have shown promising results [40, 53]. Exploiting observability don't-cares is one of the most common search-space

pruning techniques that uses circuit information, and it has been successfully carried over from SAT to QBF [53, 115]. However, there are more ways to exploit the circuit structure of QBFs than just using don't-cares.

1.1.3 Synthesis for Reconfigurability

Larger, denser and more complex digital circuits are leading to an increase in hardware faults and design errors that slip into production silicon, decreasing manufacturing yield and lengthening the design cycle. In fact, manufacturing defect levels are expected to increase sharply in future technologies [1], further decreasing yield. In order to combat these trends, adding space redundancy and using reconfigurability have been proposed in different contexts to reduce the number of silicon respins [76, 105]. Double and triple modular redundancy (DMR and TMR) are examples of design techniques that replicate parts of a design with the aim of yield enhancement as well as chip reliability improvement. Embedded FPGAs have also been used for yield improvement [4, 36]. However, these methods are costly because they incur significant area or performance overhead.

Partially programmable circuits (PPCs), recently introduced in [120], attempt to achieve a flexible balance between yield improvement and the associated costs. PPCs are obtained from conventional combinational logic circuits by replacing some subcircuits with reconfigurable elements such as look-up tables (LUTs) and configurable multiplexers (MUXs). The authors of [120] use heuristics to pick which subcircuits to replace by LUTs. They then employ *sets of pairs of functions to be distinguished* (SPFDs) [119] to add redundant connections to these LUTs and configurable MUXs, such that a large number of faults can be "bypassed" by reprogramming the PPC post-silicon. However, there are currently no available tools for evaluating the power of reconfigurability of different PPC architectures.

1.2 Contributions

The four central contributions of this thesis are summarized in the following subsections.

1.2.1 The Computational Complexity of Design Debugging

Automated design debugging has always been known to be a difficult problem. As such, it made sense to use the SAT and QBF platforms for encoding it [79, 110]. However, reducing RTL debug to SAT or QBF does not prove the theoretical computational complexity of debugging, which is a topic that has not yet been investigated.

The first contribution of this thesis is an in-depth examination of the factors affecting the theoretical computational complexity of debugging. It is first shown that the problem of (a) combinational, (b) gate-level debugging, where (c) no primary input or initial-state variable is unassigned in the counter-example, and (d) assuming the presence of a single bug, is solvable in polynomial-time. Next, we provide four proofs showing that relaxing *any one* of the above assumptions (a), (b), (c) or (d) makes the debugging problem NP-complete. This effectively draws the line where RTL debug moves from the complexity class P to that of NP-completeness, and establishes that the general debugging problem is NP-complete.

1.2.2 Debugging using Dominance

Formal debugging methodologies must return all potential bug locations in the RTL that can explain the given counter-example. Each such location in the RTL is called a *solution*. With typical design sizes containing millions of gates, the number of solutions where *corrections* can fix the counter-example, can be in the hundreds [61]. Finding each solution requires a new call to the formal engine, which is computationally expensive. We address this issue by generating *implied* solutions on-the-fly, thus reducing the number of formal iterations for returning all solutions. This is done by using dominators in the circuit.

The first contribution here is an algorithm that iteratively computes *dominance relationships* between RTL blocks (*e.g.,* always blocks, if statements or module definitions). Next, we prove that for each solution RTL block returned by the automated debugger, an RTL block that dominates it is a separate implied solution. As such, applying our algorithm as a preprocessing step, the number of formal engine calls for finding all solutions can be significantly reduced. Furthermore, we prove that corrections for implied solutions can be automatically extracted without formally analyzing these solutions. These results are shown to be valid for any error

cardinality. An extensive set of experiments on real industrial designs demonstrates that 66% of solutions are discovered early due to dominator implications, resulting in a three-fold reduction in the number of formal engine calls and a 1.64x overall performance speed-up.

1.2.3 Leveraging Dominators in Circuit-based QBFs

A new framework is presented for exploiting the circuit structure of QBFs. The idea is to simplify circuit-based QBFs by leveraging *complete dominators*, which are nodes that dominate all their fanin-cones. A methodology and a rigorous proof are given for the removal of subcircuits where all nodes are dominated by a single output in a circuit-based QBF, irrespective of input quantifiers or the structure of the remaining circuit. More precisely, the complete dominator of a subcircuit is shown to be replaceable by an appropriately computed constant or quantified input variable, without affecting the truth of the original QBF.

We present a circuit-based QBF preprocessor, called `PReDom`, which efficiently automates the process of reducing dominated subcircuits according to the presented methodology. In our experimental results, three state-of-the-art QBF solvers are able to solve 27% to 45% of the QBF instances preprocessed using `PReDom`, collectively solving 55% of all instances, compared to *none* without preprocessing.

1.2.4 Reconfigurability in Partially Programmable Circuits

A series of QBF encodings are provided for evaluating the effectiveness of reconfigurability in PPCs. We define the *fault tolerance* (respectively, *design error tolerance*) of a PPC to be the percentage of stuck-at-faults (respectively, localized design errors) that can be made unobservable using post-silicon reconfigurations, and we give QBF formulations for computing these metrics exactly. Then, we present a QBF encoding for performing synthesis for *engineering change orders* (ECOs) in PPCs using reconfigurations. ECOs are minor modifications in the specification at the later stages of the design cycle. Synthesis for ECOs strives to make the smallest number of changes to the implementation so that the design conforms to its new specification. We define a measure for quantifying the effectiveness of a PPC in implementing ECOs. We refer to this as the *ECO coverage* of a PPC architecture and we show how to

compute it using QBF. Our formulations and experimental results demonstrate the theoretical and practical appropriateness of QBF for dealing with reconfigurability.

1.3 Thesis Outline

This thesis is structured as follows. Chapter 2 provides background on Boolean satisfiability and quantified Boolean formulas, along with their respective solvers. Chapter 3 introduces design debugging and SAT-based automated RTL debug, and presents our theoretical results on the computational complexity of the problem. Chapter 4 illustrates our novel RTL debug framework that uses dominance relationships between RTL blocks for early bug discovery. Chapter 5 gives a new theory for reducing circuit-based QBFs by leveraging complete dominators and illustrates the QBF preprocessor `PReDom`. Chapter 6 gives QBF formulations for evaluating the power of reconfigurability in PPCs for correcting faults, design errors and performing ECOs. Chapter 7 discusses future work related to each of these contributions and concludes this thesis.

Chapter 2

Boolean Satisfiability and Quantified Boolean Formulas

2.1 Introduction

This chapter introduces some background material and concepts related to the contributions of this dissertation. Section 2.2 presents the canonical NP-complete problem of Boolean satisfiability (SAT). Section 2.3 introduces quantified Boolean formulas (QBF), which are a generalization of SAT used to encode PSPACE-complete problems. Finally, Section 2.4 illustrates polynomial-time reductions, NP-completeness and PSPACE-completeness.

2.2 Boolean Satisfiability

Boolean satisfiability (SAT) solvers are used as back-end engines in a variety of scientific domains such as planning [96], computational biology [32], as well as CAD for VLSI [21, 50, 67, 86, 88, 110]. This section introduces the Boolean satisfiability problem and examines the prevalent strategies used by modern SAT solvers.

A *Boolean variable* assumes a value from the set $\mathbb{B} = \{\texttt{0}$ (or false), $\texttt{1}$ (or true)$\}$. A *propositional formula* Φ is a formula constructed over a set of Boolean variables $\mathbf{x} = \{x_1, \ldots, x_n\}$ using *Boolean connectives* such as $\neg$ (negation), $\wedge$ (conjunction, *i.e.,* AND), $\vee$ (disjunction, *i.e.,*

OR), $\rightarrow$ (implication) and $\leftrightarrow$ (equivalence), along with parentheses. A *truth assignment* to a set of Boolean variables assigns each variable to 0 or 1.

Definition 2.1 *Given a propositional formula* Φ*, the* formula satisfiability *problem, or* FORMULA-SAT*, is to determine whether* Φ *has a* satisfying assignment*: a truth assignment to its variables that makes* Φ *evaluate to* 1*. If such an assignment exists,* Φ *is said to be* satisfiable *or* SAT*. Otherwise, it is* unsatisfiable *or* UNSAT*.*

The following propositional formula will be used as an example on several occasions in this section:

$$\Phi = x_1 \wedge (x_2 \leftrightarrow x_3) \wedge (x_1 \rightarrow (x_2 \vee \neg x_3)) \tag{2.1}$$

This formula is SAT because $\{x_1 = 1, x_2 = 0, x_3 = 0\}$ is a satisfying assignment making $\Phi = 1$.

The *decision tree* of a propositional formula $\Phi(x_1, \ldots, x_n)$ is a complete binary tree of height n, constructed as follows. At each level in the tree, all the internal nodes are labeled by the same variable x_i. The two branches stemming from any given node x_i are labeled by 0 and 1, corresponding to the assignment of x_i to 0 and 1, respectively. Each leaf of the tree shows the evaluation of Φ under the truth assignment given by the path from the root to that leaf. The decision tree of (2.1) is shown in Figure 2.1. Clearly, a propositional formula is SAT if and only if its decision tree has at least one leaf that evaluates Φ to 1.

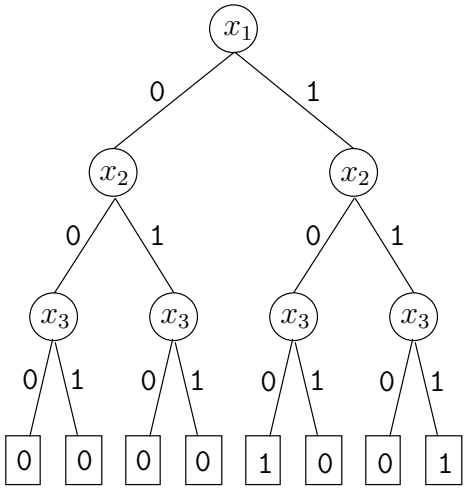


Figure 2.1: The decision tree of (2.1)

A closely related problem to formula satisfiability is *circuit satisfiability*. Formula satisfiability and circuit satisfiability are both types of SAT problems.

Definition 2.2 *Given a Boolean combinational circuit $\mathcal{C}$ with a single primary output y, the* circuit satisfiability *problem, or* CIRCUIT-SAT*, is to determine whether $\mathcal{C}$ has a* satisfying assignment*: a truth assignment to its primary inputs that makes y evaluate to `1`. If such an assignment exists, $\mathcal{C}$ is said to be `SAT`. Otherwise, it is `UNSAT`.*

2.2.1 Complexity of SAT

CIRCUIT-SAT holds a special place in the history of Computer Science as it is the first problem to be proven to be NP-complete [30]. FORMULA-SAT is also NP-complete. In addition, a plethora of important CAD for VLSI problems in synthesis, test and verification are NP-complete [101]. Section 2.4 gives a detailed introduction to NP-completeness.

No polynomial-time algorithms are currently known for any NP-complete problems. However, all NP-complete problems are efficiently reducible to one another. As such, other NP-complete problems can be *encoded* as SAT problems and solved using SAT solvers. With the tremendous improvements in SAT solvers over the last two decades [38, 84, 87] and yearly SAT solver competitions [69] consistently advancing the state-of-the-art, this approach has become very common. In VLSI CAD, SAT solvers are used as underlying engines in synthesis [86], test pattern generation [67], formal verification [21, 50], automated design debugging [110] and FPGA routing [52, 88], among other problems. In other words, instead of dedicated algorithms to solve each of these VLSI CAD problems, they are encoded as SAT problems and generic SAT solvers are used to solve them efficiently.

2.2.2 CNF Representation

A propositional formula given in *conjunctive normal form* (CNF) consists of a conjunction of *clauses*, where each clause is a disjunction of *literals*. A literal is an occurrence of a variable in its positive or negative polarity, x or $\neg x$. The FORMULA-SAT problem with the input formula given in CNF is called CNF-SAT, which is also NP-complete [33]. Most modern SAT solvers

accept their inputs as CNF formulas rather than general formulas or circuits. In order for the CNF formula to be SAT, the solver must satisfy each clause by setting at least one literal in it to 1. A literal $l = x$ (respectively, $l = \neg x$) evaluates to 1 if its corresponding variable $x = 1$ (respectively, $x = 0$).

Consider the following CNF formula, which is logically equivalent to (2.1):

$$\Phi = (x_1) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \tag{2.2}$$

Here, Φ contains four clauses (x_1), $(x_2 \vee \neg x_3)$, $(\neg x_2 \vee x_3)$, $(\neg x_1 \vee x_2 \vee \neg x_3)$, three variables x_1, x_2, x_3 and six literals x_1, $\neg x_1$, x_2, $\neg x_2$, x_3, $\neg x_3$. The clause (x_1) is called a *unit clause* because it contains only one literal, effectively forcing $x_1 = 1$ in any satisfying assignment. The truth assignment $\{x_1 = 1, x_2 = 0, x_3 = 0\}$, which is a satisfying assignment of (2.1), is also a satisfying assignment of (2.2) because it satisfies every clause in it.

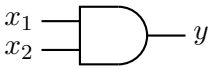
Two SAT instances are said to be *equisatisfiable* if the first is SAT whenever the second is SAT and vice versa. Instances of FORMULA-SAT can be translated to equisatisfiable instances of CNF-SAT in polynomial-time. Although no extra variables were used in the translation of (2.1) to (2.2), in general a polynomial-time translation requires the use of auxiliary variables [33]. In what follows, we will illustrate the translation of combinational circuits to CNF formulas, which is much more common in VLSI CAD.

A combinational circuit can be encoded, or *modeled*, as a CNF formula as follows. First, each primary input, primary output and internal gate in the circuit is associated with a unique Boolean variable. Next, for each gate, a CNF formula is generated which evaluates to 1 if the values assigned to the variables corresponding to the inputs and the output of the gate are consistent, and to 0 otherwise. Figure 2.2 illustrates the generation of the CNF formula encoding an AND gate. Table 2.1 gives the CNF clauses for a number of gate types based on the Tseitin transformation [116]. The CNF formula modeling the whole circuit is formed by conjuncting the CNF formulas of each gate. It evaluates to 1 if and only if all the values assigned to the primary inputs, primary outputs and internal gates of the circuit are consistent. As such, any combinational circuit can be encoded as a CNF formula expressing the same constraints. This translation requires linear time as long as "counting"-type gates, such as XORs, are assumed

to have a fixed maximum number of inputs [67, 109].

Example 2.1 *The CNF formula for an* `AND` *gate* $y = x_1 \wedge x_2$ *is shown in Figure 2.2.*

x_1	x_2	y	Φ_{AND}
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



$$\Phi_{\text{AND}}(x_1, x_2, y) = (x_1 \vee \neg y) \wedge (x_2 \vee \neg y) \wedge (\neg x_1 \vee \neg x_2 \vee y)$$

Figure 2.2: CNF formula for a two-input `AND` gate

The translation of Boolean circuits to CNF formulas can be used to reduce instances of CIRCUIT-SAT to instances of CNF-SAT by adding a final constraint that forces the circuit output to `1`. This can be done by conjuncting the unit clause (y) to the CNF formula generated by the procedure above, where y denotes the primary output of the circuit. A satisfying assignment of the resulting CNF formula would correspond to a truth assignment to the inputs and internal gates of the original circuit, which would make its primary output $y =$ `1`.

Many NP-complete CAD for VLSI problems involve a circuit representation coupled with additional problem-specific constraints. Encoding these problems as SAT problems often boils down to translating these additional constraints into a CNF formula efficiently. For instance, Subsection 3.2.1 shows how to formulate the design debugging problem using SAT.

2.2.3 SAT Solvers

Most SAT solvers are based on a backtrack search algorithm devised in 1962 by Davis, Logemann and Loveland, called DPLL [34]. The DPLL algorithm can be seen as a depth-first search (DFS) of the decision tree of the CNF formula. The formula is simplified after each variable is assigned to `0` or `1`. If the variable assignments from the root to a given node in the tree make a clause evaluate to `0`, a *conflict* is identified and the solver backtracks.

Original DPLL SAT solvers performed *chronological backtracking*, where in the event of a conflict, the polarity of the most recent variable assignment that is not tried both ways is

Gate	Function	CNF Formula
AND	$y = x_1 \wedge x_2 \wedge \cdots \wedge x_n$	$\left[\bigwedge_{i=1}^{n}(x_i \vee \neg y)\right] \wedge \left(\bigvee_{i=1}^{n} \neg x_i \vee y\right)$
NAND	$y = \neg(x_1 \wedge x_2 \wedge \cdots \wedge x_n)$	$\left[\bigwedge_{i=1}^{n}(x_i \vee y)\right] \wedge \left(\bigvee_{i=1}^{n} \neg x_i \vee \neg y\right)$
OR	$y = x_1 \vee x_2 \vee \cdots \vee x_n$	$\left[\bigwedge_{i=1}^{n}(\neg x_i \vee y)\right] \wedge \left(\bigvee_{i=1}^{n} x_i \vee \neg y\right)$
NOR	$y = \neg(x_1 \vee x_2 \vee \cdots \vee x_n)$	$\left[\bigwedge_{i=1}^{n}(\neg x_i \vee \neg y)\right] \wedge \left(\bigvee_{i=1}^{n} x_i \vee y\right)$
XOR	$y = x_1 \oplus x_2$	$(\neg x_1 \vee \neg x_2 \vee \neg y) \wedge (x_1 \vee x_2 \vee \neg y) \wedge$ $(\neg x_1 \vee x_2 \vee y) \wedge (x_1 \vee \neg x_2 \vee y)$
XNOR	$y = \neg(x_1 \oplus x_2)$	$(\neg x_1 \vee \neg x_2 \vee y) \wedge (x_1 \vee x_2 \vee y) \wedge$ $(\neg x_1 \vee x_2 \vee \neg y) \wedge (x_1 \vee \neg x_2 \vee \neg y)$
BUFFER	$y = x$	$(x \vee \neg y) \wedge (\neg x \vee y)$
NOT	$y = \neg x$	$(x \vee y) \wedge (\neg x \vee \neg y)$
MUX	$y = s \ ? \ x_1 : x_0$	$(\neg x_0 \vee s \vee y) \wedge (x_0 \vee s \vee \neg y) \wedge$ $(x_1 \vee \neg s \vee \neg y) \wedge (\neg x_1 \vee \neg s \vee y)$

Table 2.1: CNF formulas for elementary gate types.

Algorithm 2.1: Typical CDCL SAT algorithm [38]

```
   input  : CNF formula
   output: SAT/UNSAT
 1 while true do
 2     PROPAGATE();                          // Boolean constraint propagation
 3     if no conflict then
 4         if all variables assigned then
 5             return SAT;
 6         else
 7             DECIDE();                     // assign a new variable
 8     else
 9         ANALYZECONFLICT();                // add a conflict clause
10         if unresolvable conflict then
11             return UNSAT;
12         else
13             BACKTRACKFROMCONFLICT();
```

flipped [34]. In modern SAT solvers, each conflict is analyzed and a small subset of conflicting variable assignments is identified. Based on this, a *learned clause* is added to the original clause database in order to block the conflicting truth assignment, disallowing the solver from retrying that assignment combination in the future. This also makes it possible for the solver to skip decision variables unrelated to the conflict while backtracking upwards in the decision tree, a process referred to as *non-chronological backtracking* or *backjumping* [84]. Backjumping solvers are called *conflict-driven clause learning* (CDCL) SAT solvers [83]. Today, CDCL SAT solvers implement a number of additional key techniques, such as using lazy data structures for the representation of formulas [87], advanced branching heuristics with low computational overhead [87], periodic restarts of the search [51], preprocessing algorithms [37], among many others.

The structure of a typical CDCL SAT solver [38] is shown in Algorithm 2.1. On line 2, the PROPAGATE() function performs *Boolean constraint propagation* (BCP), which propagates forced assignments in the CNF formula due to unit clauses until convergence. In the event that a conflict is found, the function ANALYZECONFLICT() on line 9 finds the subset of variable assignments responsible for it, adding them to a conflict clause which is unsatisfied. If the conflict is unresolvable, the solver returns `UNSAT`. Otherwise, the function BACKTRACKFROMCONFLICT() on line 13 undoes assignments until the conflict clause learned in ANALYZECONFLICT() becomes unit. In other words, the solver backjumps in the decision tree until the newly unit conflict clause forces it to switch the polarity of a variable. On the other hand, if PROPAGATE() does not produce a conflict, a new variable is picked and assigned to `0` or `1` in DECIDE() on line 7, based on heuristics [87]. If all the variables have been assigned without leading to conflicts after BCP, then the CNF formula is `SAT`.

Although CDCL and DPLL have exponential time worst-case complexities, modern SAT solvers (*e.g.,* [10, 19, 38]) are able to efficiently handle industrial SAT problems with millions of variables and clauses. It is sometimes necessary to find *all* satisfying assignments of a SAT problem rather than just one, such as in SAT-based design debugging [110]. A regular SAT solver can be modified to return all satisfying assignments by adding a *blocking clause* for each newly found solution, which makes that assignment unsatisfiable in future runs, and re-solving

iteratively until the problem becomes UNSAT. Such a solver is called an *all-solution* SAT solver.

Example 2.2 *Consider the CNF formula given in (2.2). When the satisfying assignment* $\{x_1 = 1, x_2 = 0, x_3 = 0\}$ *is discovered, the all-solution SAT solver adds the blocking clause* $(\neg x_1 \vee x_2 \vee x_3)$ *to the CNF formula. The solver is rerun and finds another satisfying assignment,* $\{x_1 = 1, x_2 = 1, x_3 = 1\}$*. The corresponding blocking clause* $(\neg x_1 \vee \neg x_2 \vee \neg x_3)$ *is added to the CNF formula. At this point, the CNF formula becomes UNSAT, which means that all solutions have been returned.*

2.3 Quantified Boolean Formulas

Quantified Boolean formulas (QBFs) are a natural extension of propositional formulas where variables can be *bound* by *existential* ($\exists$) or *universal* ($\forall$) *quantifiers* [24]. Given a propositional formula Φ on one variable x, the QBF $\exists x.\Phi(x)$ is true (*i.e.,* evaluates to 1) if and only if there exists an $x \in \{0, 1\}$, such that $\Phi(x) = 1$. The QBF $\forall x.\Phi(x)$ is true if and only if for all $x \in \{0, 1\}$, $\Phi(x) = 1$. Formally, we can write:

$$\exists x.\Phi(x) = \Phi(0) \vee \Phi(1) \tag{2.3}$$

$$\forall x.\Phi(x) = \Phi(0) \wedge \Phi(1) \tag{2.4}$$

In this dissertation, we are interested in QBFs in *prenex form*, written as:

$$q_1 x_1 \; q_2 x_2 \; \cdots \; q_m x_m \; . \; \Phi(x_1, \ldots, x_n) \tag{2.5}$$

where $m \leq n$, $q_i \in \{\exists, \forall\}$ and Φ is a propositional formula over $\mathbf{x} = \{x_1, \ldots, x_n\}$. This formula can be abbreviated as $Q.\Phi$, where

$$Q = q_1 x_1 \; q_2 x_2 \cdots \; q_m x_m \tag{2.6}$$

is called the *prefix*, while Φ is referred to as the *matrix*.

A variable that is not bound in the prefix is called *free*. A variable is labeled as an existential (respectively, universal) variable if it is bound to $\exists$ (respectively, $\forall$). A QBF is termed *closed* if it contains no free variables, *i.e.,* $m = n$. Otherwise, it is *open*. A closed QBF is either true or

false. An open QBF is a function of its free variables. Notice that a closed QBF of the form:

$$\exists x_1 \cdots \exists x_n \ . \ \Phi(x_1, \ldots, x_n), \tag{2.7}$$

where all variables are existential, is true if and only if $\Phi(x_1, \ldots, x_n)$ is `SAT`. As such, the SAT problem can be encoded as a closed QBF.

In general, the quantification order in the prefix matters. *E.g.,* $\exists x_1 \forall x_2 . \Phi(x_1, x_2)$ is generally different than $\forall x_2 \exists x_1 . \Phi(x_1, x_2)$. However, the quantification order of consecutive existential (respectively, universal) variables is interchangeable. Therefore it is common practice to group each set of consecutive existential (respectively, universal) variables in the prefix as follows:

$$Q = q_1 \mathbf{v_1} \ q_2 \mathbf{v_2} \ \cdots \ q_r \mathbf{v_r} \tag{2.8}$$

such that the q_i's are alternating quantifiers (*i.e.,* $q_i \neq q_{i+1}$), and the $\mathbf{v_i}$'s are mutually disjoint variable sets partitioning $\mathbf{x}$, called *scopes*. In other terms, $\bigcup_{i=1}^{r} \mathbf{v_i} = \mathbf{x}$ and $\bigcap_{i=1}^{r} \mathbf{v_i} = \emptyset$. A scope $\mathbf{v_i}$ or variable $x \in \mathbf{v_i}$ is said to be *wider* (respectively, *narrower*) than a scope $\mathbf{v_j}$ or variable $x' \in \mathbf{v_j}$ if $i < j$ (respectively, $i > j$). q_r (respectively, q_1) is the innermost (respectively, outermost) quantifier. Similarly, $\mathbf{v}_r$ (respectively, $\mathbf{v}_1$) is the innermost (respectively, outermost) scope.

The *reduction* of a QBF $Q.\Phi$ by a literal x (respectively, $\neg x$) is denoted by $Q.\Phi|_x$ (respectively, $Q.\Phi|_{\neg x}$), which replaces occurrences of x by `1` (respectively, `0`) and $\neg x$ by `0` (respectively, `1`). More generally, if π is a truth assignment over a subset of $\{x_1, \ldots, x_n\}$, then $Q.\Phi|_\pi$ denotes the formula $Q.\Phi$ after assigning these variables to their truth values in π. The notations $\pi = \{x_i = \mathtt{1}, x_j = \mathtt{0}, \ldots\}$ and $\pi = \{x_i, \neg x_j, \ldots\}$ are equivalent and used interchangeably.

Semantically, a QBF can be evaluated by the recursive application of the following two rules [53]:

$$\exists x Q.\Phi = Q.\Phi|_x \vee Q.\Phi|_{\neg x} \tag{2.9}$$

$$\forall x Q.\Phi = Q.\Phi|_x \wedge Q.\Phi|_{\neg x} \tag{2.10}$$

Example 2.3 *Consider the following closed QBF:*

$$\exists x_1 \forall x_2 \ . \ (x_1 \leftrightarrow x_2) \tag{2.11}$$

Applying rules (2.9) and (2.10), we get:

$$\begin{aligned}\exists x_1 \forall x_2 \,.\, (x_1 \leftrightarrow x_2) &= [\forall x_2.(1 \leftrightarrow x_2)] \vee [\forall x_2.(0 \leftrightarrow x_2)] \\ &= [\forall x_2.(x_2)] \vee [\forall x_2.(\neg x_2)] \\ &= (1 \wedge 0) \vee (0 \wedge 1) \\ &= 0.\end{aligned}$$

This process is equivalent to asking whether there exists an x_1 such that for all x_2, $x_1 \leftrightarrow x_2$. No such x_1, x_2 exist, making the QBF false.

The generalization of a single satisfying assignment in SAT is called a *Q-model* in a closed QBF, and it is defined as follows. Consider a decision tree of the QBF matrix that respects the prefix quantification order. In other terms, the root of the tree is labeled by a variable from the widest prefix scope, and the internal nodes at the lowest level are labeled by a variable from the narrowest scope. A Q-model is any subtree of this decision tree that (a) has the same root, (b) includes exactly one branch (`0` *or* `1`) at every existential node, (c) includes both branches (`0` *and* `1`) at every universal node, and (d) whose leaves correspond to satisfying assignments of the QBF matrix (*i.e.,* assignments that make $\Phi = \mathtt{1}$). If such a Q-model exists, then the QBF is true. Otherwise, it is false.

A Q-model can be equivalently thought of as a set of Boolean functions, called *Skolem functions*, defined as follows. For each existential variable x_i, its Skolem function f_i is a Boolean function of all universal variables wider than x_i, such that for all truth assignments to the universal variables in the QBF, assigning the existential variables according to the valuation of their Skolem functions yields $\Phi = \mathtt{1}$.

Example 2.4 *Consider the following closed QBF in prenex form:*

$$\exists x_1 \; \forall x_2 \; \exists x_3 \,.\, (x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1) \tag{2.12}$$

Here, the prefix consists of two existential scopes $\mathbf{v}_1 = \{x_1\}$ and $\mathbf{v}_3 = \{x_3\}$, and one universal scope $\mathbf{v}_2 = \{x_2\}$. Variable x_1 has wider scope than x_2, which has wider scope than x_3.

Figure 2.3 shows the decision tree associated with the QBF in (2.12), which follows the prefix quantification order. At each leaf, the value in the box gives the evaluation of the QBF matrix

under the truth assignment given by the path from the root to that leaf. The symbol $\vee$ is shown under existential variables indicating that only one branch needs to be included in a Q-model, and the symbol $\wedge$ is shown under universal variables indicating that both branches need to be included. The bolded subtree of truth assignments in Figure 2.3 is the only Q-model of (2.12). In words, when $x_1 = 1$, for all values of x_2, there exists an assignment to x_3 ($x_3 = 1$ when $x_2 = 0$ and $x_3 = 0$ when $x_2 = 1$) that satisfies the matrix. As such, this QBF is true.

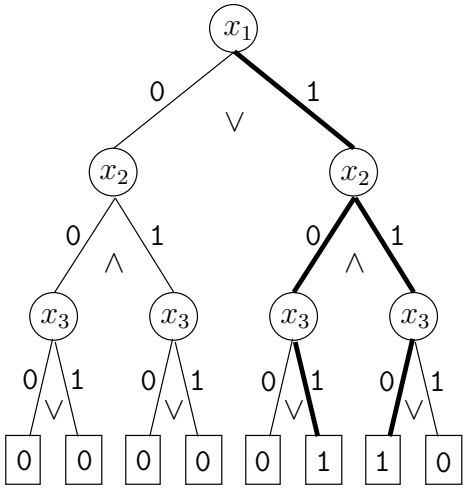


Figure 2.3: A Q-model of the QBF in (2.12), shown in bold

The Q-model shown in Figure 2.3 can be equivalently represented using the following two Skolem functions, one for each existential variable. The Skolem function of x_1 is $f_1 = 1$, which is constant since there are no universals wider than x_1. The Skolem function of x_3 is $f_3(x_2) = \neg x_2$, which is a function of x_2. For both assignments to x_2 in (2.12), replacing x_1 and x_3 by their Skolem function valuations (if $x_2 = 0$, then $\{x_1 = 1, x_3 = 1\}$, and if $x_2 = 1$, then $\{x_1 = 1, x_3 = 0\}$) satisfies the matrix of (2.12).

A QBF is said to be in *prenex normal form* if the matrix Φ is in CNF. The QBF in Example 2.4 is in prenex normal form. Most modern QBF solvers accept their inputs as closed QBFs in prenex normal form.

2.3.1 Complexity of QBF

Section 2.4 gives some background on PSPACE-completeness. QBF is the canonical PSPACE-complete problem, which makes it a more expressive encoding formalism than SAT, given the widely accepted conjecture that NP $\subset$ PSPACE. Many interesting problems in AI, such as adversarial games [92], as well as a number of CAD for VLSI problems, such as LTL model checking [108], sequential reachability analysis [106], and sequential equivalence checking [12], are known to be PSPACE-complete and can therefore be efficiently encoded as QBF instances. Similarly to SAT solvers, QBF solvers have improved significantly during the last decade with a yearly QBF solver competition [93] encouraging consistent advancements.

2.3.2 QBF Solvers

Modern QBF solvers are more diverse than SAT solvers in terms of their underlying solving strategies. Approaches based on backtrack search [47, 75, 102], resolution and expansion [18] and skolemization [13] can all be competitive. Recently, several powerful QBF solvers have been developed that accept matrices in a circuit-based format as opposed to CNF [40, 53, 74]. Finally, there are also multi-engine QBF solvers that dynamically select among other existing solvers according to certain heuristics [94]. Today, QBF solvers can handle industrial problems containing tens to hundreds of thousands of variables and clauses. In what follows, we briefly examine major QBF solving approaches.

Backtrack search-based QBF solvers [26, 47, 75] extend CDCL SAT algorithms to deal with universal quantification. Algorithm 2.2 shows a typical search-based QBF algorithm, which is similar to Algorithm 2.1 with two important differences. First, on line 11, the DECIDE() function must now respect the prefix quantification order by first branching on all variables in the outermost scope $\mathbf{v}_1$, then on all variables in the next scope $\mathbf{v}_2$, and so on. In other terms, it is illegal to branch on a variable if there exists a wider unassigned variable. On the other hand, the decision order within a particular scope is arbitrary. This is a generalization of DECIDE() in Algorithm 2.1 since all variables in a SAT problem belong to a single (existential) scope.

The second difference is how to deal with satisfactions. On line 5, the matrix is satisfied but the QBF solver cannot return because of universal quantification. Instead, the function

Algorithm 2.2: Typical search-based QBF algorithm

input : Closed QBF in prenex normal form

output: True/false

```
1  while true do
2      propagate();                                  // Boolean constraint propagation
3      if no conflict then
4          if all variables assigned then
5              analyzeSatisfaction();                        // add a satisfaction cube
6              if unresolvable satisfaction then
7                  return true;
8              else
9                  backtrackFromSatisfaction();
10         else
11             decide();          // assign a new variable following the prefix order
12     else
13         analyzeConflict();                                  // add a conflict clause
14         if unresolvable conflict then
15             return false;
16         else
17             backtrackFromConflict();
```

ANALYZESATISFACTION() implements *satisfiability-driven learning* to learn from satisfactions. It has the dual functionality of ANALYZECONFLICT(): Just as the function ANALYZECONFLICT() prunes the non-solution space of the decision tree, ANALYZESATISFACTION() prunes the space where solutions are known to exist. This is useful because a QBF solver does not stop when a satisfying assignment is found, and must instead find a Q-model which is a tree of satisfying assignments. ANALYZESATISFACTION() will learn a *satisfaction cube*, which is a conjunction of literals making the formula true. Later in the search, a satisfaction cube will allow the QBF solver to avoid revisiting that combination of assignments, since it is already known that it will lead to a satisfaction.

Example 2.5 *Consider the following QBF:*

$$\exists x_1 \; \forall x_2, x_3 \; \exists x_4 \; . \; (x_1 \vee x_2) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4) \tag{2.13}$$

Let us suppose that during the search process, the QBF solver finds the satisfying assignment $\{x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 0\}$*. If the function* ANALYZESATISFACTION*() determines that whenever* $\{x_1 = 1, x_3 = 0\}$*, irrespective of other assignments, the matrix is satisfiable, it can learn the cube* $(x_1 \wedge \neg x_3)$*. At another point in the decision tree, if* $x_1 = 1$*, the* PROPAGATE*() function will use this cube to force* $x_3 = 1$ *because it is already known that* $x_3 = 0$ *would lead to a satisfying assignment.*

Several alternatives exist to search-based QBF solvers. The solver `quantor` [18] evaluates QBFs using *variable elimination*. It eliminates variables from the innermost existential scope using *Q-resolution* [18, 25] and variables from the innermost universal scope using *expansion* [18]. In general, both of these techniques can introduce an exponential number of intermediate clauses. As such, a major component of `quantor` is a subroutine deciding which variable to eliminate at a given point in order to limit the matrix size increase. On hard QBF instances, search-based solvers usually time-out, whereas variable elimination-based solvers tend to run out of memory.

Another QBF solver, `sKizzo` [13, 15], relies on a completely different strategy. It uses *Skolemization* to successively compute the Skolem functions of the existential variables. `sKizzo`

computes and stores these functions compactly using *binary decision diagrams* (BDDs) [23]. It also uses a number of other techniques, such as *quantifier trees* to represent the QBF prefix [16], instead of the traditional linear representation given by 2.8.

Finally, QBF instances originating from CAD problems usually have a circuit structure which is lost during the conversion into CNF. A number of recent papers [97, 122] have analyzed the limitations of the use of CNF for QBF matrices and have offered alternative representations addressing some of these issues. QBF solvers that accept problems directly in a circuit-based format have shown promising results using various solving methods [40, 53, 74]. In Chapter 5, we present a new preprocessing technique which takes advantage of a circuit-based QBF matrix.

2.4 NP-completeness and PSPACE-completeness

The *Kleene star* of the binary alphabet $\{0, 1\}$, denoted by $\{0, 1\}^*$, is the set of all binary strings composed of symbols from $\{0, 1\}$. A *binary encoding* of an abstract object O is a mapping of this object to a binary string, written as $\langle O \rangle$. Graphs, polygons, integers, programs, and so on, can all be encoded as binary strings[1]. As such, the input of any problem, called a *problem instance*, can be encoded as a binary string in $\{0, 1\}^*$.

A *decision problem* is one whose answer is one of two values (*e.g.,* yes/no, 1/0, true/false, SAT/UNSAT). The *language* corresponding to a decision problem denotes the set of binary strings (encoding problem instances) that map to 1. A decision problem can be entirely characterized by its corresponding language. For example, the CIRCUIT-SAT language consists of all binary encodings of single-output combinational circuits that are satisfiable:

$$\text{CIRCUIT-SAT} = \{\langle \mathcal{C} \rangle \mid \mathcal{C} \text{ is SAT}\} \tag{2.14}$$

A language and its corresponding decision problem are often referred to interchangeably.

Now we can assign languages (or decision problems) to *complexity classes* according to certain criteria. We say that a language (respectively, a decision problem) L is *decidable* (respectively, *solvable*) in *polynomial-time* if there exists an algorithm A and a constant c such

[1] In fact, during translation into machine code in a modern computer, these objects are automatically encoded as binary strings.

that for any length-n input string $x \in \{0,1\}^*$, A determines whether or not x belongs to L in time $O(n^c)$ [33]. If $x \in L$, A *accepts* x (*i.e.,* $A(x) = 1$), otherwise A *rejects* x (*i.e.,* $A(x) = 0$). The complexity class P contains all languages that are decidable in polynomial-time:

$$\text{P} = \{L \mid L \text{ in decidable in polynomial-time}\} \tag{2.15}$$

In Computer Science, problems that are solvable in polynomial-time are considered tractable, whereas those that require superpolynomial-time are considered intractable.

For a given language L, a *verification algorithm* is a two-argument algorithm A, where one argument is the input string x and the other is a binary string y called a *certificate* [33]. Algorithm A *verifies* an input string x if there exists a certificate y such that $A(x,y) = 1$. Intuitively, a certificate y which makes $A(x,y) = 1$ is an object that a verification algorithm can use as "proof" that x belongs to the language. We say that a language L is *verifiable* in polynomial-time if there exists a verification algorithm A and constants c and d, such that (a) for any length-n $x \in L$, there exists a certificate y of size $O(n^d)$ which A accepts (*i.e.,* $A(x,y) = 1$) in time $O(n^c)$, and (b) for all $x \notin L$, every certificate y is rejected by A (*i.e.,* $A(x,y) = 0$) in time $O(n^c)$. The complexity class NP contains all languages that are verifiable in polynomial-time:

$$\text{NP} = \{L \mid L \text{ is verifiable in polynomial-time}\} \tag{2.16}$$

The complement $\overline{L}$ of a language L contains all strings or problem instances not in L. The complexity class co-NP contains all languages whose complements are in NP:

$$\text{co-NP} = \{L \mid \overline{L} \in \text{NP}\} \tag{2.17}$$

In other terms, co-NP is the class of problems for which instances *not* in the language can be verified in polynomial-time.

Example 2.6 *Consider the CIRCUIT-SAT language. An instance (x) of CIRCUIT-SAT is an encoding of a circuit $\langle \mathcal{C} \rangle$. A certificate ($y$) for CIRCUIT-SAT consists of (an encoding of) a truth assignment π to the primary inputs of $\mathcal{C}$. The verification algorithm $A(\langle \mathcal{C} \rangle, \langle \pi \rangle)$ simply performs circuit simulation under π and returns the assignment to the circuit primary output.*

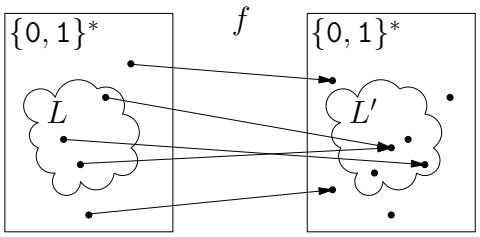


Figure 2.4: A reduction f from L to L'

*If $\langle \mathcal{C} \rangle \in$ CIRCUIT-SAT, then clearly there exists a certificate (*i.e., *a truth assignment to the primary inputs) $\langle \pi \rangle$, such that simulating it makes the output $\mathtt{1}$, and hence A accepts. On the other hand, if $\langle \mathcal{C} \rangle \notin$ CIRCUIT-SAT, then for all certificates (*i.e., *truth assignments to the primary inputs) $\langle \pi \rangle$ simulating them makes the output $\mathtt{0}$, and hence A rejects. Finally, circuit simulation takes polynomial-time in the circuit size and $\langle \pi \rangle$ is of polynomial size with respect to $\langle \mathcal{C} \rangle$. Therefore, CIRCUIT-SAT $\in$ NP.*

Definition 2.3 *[33] A language L is* polynomial-time reducible *to another language L', written as $L \leq_p L'$, if there exists a polynomial-time computable function $f : \{\mathtt{0},\mathtt{1}\}^* \to \{\mathtt{0},\mathtt{1}\}^*$, such that for all $x \in \{\mathtt{0},\mathtt{1}\}^*$:*

$$x \in L \iff f(x) \in L'. \tag{2.18}$$

We call the function f the reduction function.

Intuitively, $L \leq_p L'$ means that instances of L can be translated into instances of L' in polynomial-time and therefore solving L is not more than a polynomial factor harder than solving L'. It is also said that L is efficiently reducible to L'. Figure 2.4 illustrates a reduction function f from L to L'. By Definition 2.3, f must reduce *every* instance $x \in L$ into an instance of L', and *every* instance $x \notin L$ into a string outside L'. Note that the mapping f does not have to be surjective (*i.e.,* onto), which means that there may be instances in $\{\mathtt{0},\mathtt{1}\}^*$ that f does not map to, as shown in Figure 2.4. Also f does not have to be injective (*i.e.,* one-to-one), which means that it can map different strings to the same string, as shown in Figure 2.4.

Definition 2.4 *[33] A language L is NP-complete if:*

1. $L \in NP$, *and*

2. $L' \leq_p L$, *for all* $L' \in NP$.

A language that satisfies the second property in Definition 2.4, but not necessarily the first, is said to be NP-hard. In other words, a problem is NP-hard if every language in NP is polynomial-time reducible to it. NP-complete problems are those NP-hard problems that are also themselves in NP. They can be thought of as the hardest problems in NP. Note that every NP-complete problem is polynomial-time reducible to every other NP-complete problem. The class of all NP-complete languages is denoted by NPC. If a polynomial-time algorithm is found to solve any given NP-complete problem, then P = NP. No polynomial-time algorithms are currently known for any NP-complete problems.

Using the transitivity of polynomial-time reducibility, reducing a known NP-complete problem to L in polynomial-time is enough to prove that L is NP-hard, as shown in the following definition.

Definition 2.5 *A language L is NP-complete if:*

1. $L \in NP$, *and*

2. $L' \leq_p L$, *for some* L' *that is NP-complete.*

The class of co-NP-complete problems is defined as follows.

Definition 2.6 *A language L is co-NP-complete if:*

1. $L \in$ *co-NP, and*

2. $L' \leq_p L$, *for all* $L' \in$ *co-NP.*

We say that a language (or decision problem) is decidable (solvable) in *polynomial-space* if there exists an algorithm A and a constant c such that for any length-n input string $x \in \{0,1\}^*$, A determines whether or not x belongs to L using $O(n^c)$ memory bits. Of course, unlike the

time resource, space is reusable. The complexity class PSPACE contains all languages that are decidable in polynomial-space:

$$\text{PSPACE} = \{L \mid L \text{ in decidable in polynomial-space}\} \tag{2.19}$$

Definition 2.7 *A language L is said to be PSPACE-complete if:*

1. *$L \in$ PSPACE, and*
2. *$L' \leq_p L$, for all $L' \in$ PSPACE.*

PSPACE-complete languages are the hardest problems in PSPACE, in the sense that all languages in PSPACE are polynomial-time reducible to any PSPACE-complete language. As such, analogously to NP-complete problems, PSPACE-complete problems are polynomial-time inter-reducible. It is a widely accepted conjecture that PSPACE-complete languages are outside NP, although this has not been proven. The language of all true QBFs is considered the canonical PSPACE-complete problem, which makes it a more powerful encoding formalism than SAT, assuming that NP $\subset$ PSPACE.

Chapter 3

The Computational Complexity of Design Debugging

3.1 Introduction

One of the main reasons for the discrepancy in difficulty between computer-aided verification and design, a trend called the *verification gap*, is the computational difficulty of most functional verification tasks. For instance, combinational equivalence checking (CEC), arguably the most basic form of formal verification which is used to prove or disprove the functional equivalence of two combinational circuits, is co-NP-complete [49]. Sequential equivalence checking (SEC) is even harder; it is PSPACE-complete [12]. Model checking (MC), which aims to verify user-specified temporal properties of sequential designs, is also PSPACE-complete given properties in linear temporal logic (LTL) [108], such as SystemVerilog assertions (SVA).

Although several methodologies have been proposed over the years to automate the design debugging process, with many of them using formal encodings such as SAT and QBF [3, 55, 79, 110], the theoretical computational complexity of debugging has not yet been studied. This chapter presents several theoretical results on the complexity of debugging. We first illustrate that the problem of (a) combinational, (b) gate-level debugging, where (c) no primary input or initial-state variable is unassigned in the counter-example, and with (d) single error cardinality, is solvable in polynomial-time. Next, we provide four proofs showing that relaxing *any one* of

the above assumptions (a) to (d) makes the debugging problem NP-complete. This exposes the line where the design debugging problem moves from the complexity class P to that of NP-completeness, and establishes that the general debugging problem is NP-complete.

This chapter is organized as follows. Section 3.2 provides a detailed background on design debugging, with a focus on SAT-based automated debugging. Section 3.3 presents our results on the complexity of design debugging, and Section 3.4 summarizes the chapter.

3.2 Design Debugging

A *buggy* design implementation is one that exhibits a functional mismatch compared to its specification under at least one sequence of input stimuli. A *counter-example* consists of such a sequence of primary inputs, starting from a given initial-state, leading to a discrepancy between the actual and expected responses of a design implementation and its specification, respectively. A *suspect* is a component in the buggy circuit or RTL (*e.g.,* a logic gate, a specific RTL line, a module definition, an *always* block, and so on) that is not guaranteed to be bug-free. Given a buggy design and a counter-example, the output of an automated design debugger is a set of potential bug locations, which we refer to as *solutions*. Each solution denotes a set of suspects, where simultaneous modifications, called *corrections*, can rectify the erroneous behavior exhibited in the given counter-example.

There may be more than one set of RTL components that can be modified to separately correct the counter-example. An automated debugger must return *all* such solutions. The engineer is given the final task of going through the list of solutions in order to identify the actual bug and fix it. If several solutions are deemed appropriate to fix the bug, the engineer may prefer one solution over another in order to perform a correction. This preference is affected by the simplicity of the correction, as well as the effect of the change on circuit performance such as timing, power consumption, and so on [99].

In gate-level debugging, each logic gate is a separate suspect and therefore each solution is a set of gates [110]. In hierarchical debugging, a suspect can be a multiple-output module and as such, each solution is a set of modules [6]. The *error cardinality* of a solution, denoted

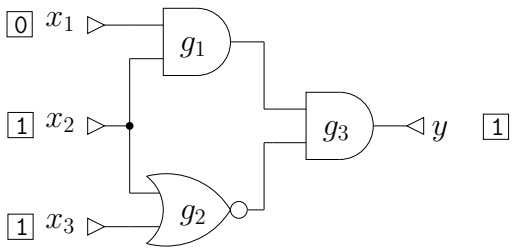


Figure 3.1: An erroneous circuit

by N, is the number of suspects it contains, where simultaneous modifications rectify the counter-example. Automated debugging tools normally give a preference to solutions with fewer components [110] because they usually require less modifications. Furthermore, counter-examples usually stop at the first clock-cycle where there is an output discrepancy, making it uncommon for multiple bugs to simultaneously propagate to a primary output during that clock-cycle [61].

Example 3.1 *Consider the combinational circuit given in Figure 3.1 to be a buggy implementation that does not pass verification. The Boolean values in the boxes shown next to the primary inputs and the primary output of the circuit respectively represent the input stimuli* $\langle x_1, x_2, x_3 \rangle = \langle 0, 1, 1 \rangle$ *of a counter-example and the corresponding* expected *output response* $y = 1$*. The circuit in Figure 3.1 yields the output* $y = 0$ *for that input combination, demonstrating a mismatch compared to its expected behavior.*

Assuming that each internal gate is a separate suspect, one gate-level debugging solution is the `AND` *gate* g_3*, since changing it to a* `NAND` *gate would correct the counter-example by making* $y = 1$*, as required. The set of gates* $\{g_1, g_2\}$ *is another solution, of error cardinality* $N = 2$*, since changing both of them to* `OR` *gates also fixes the counter-example. The maximum error cardinality is normally given to the automated debugger by the user. The debugger starts by finding all solutions with* $N = 1$*, then all solutions with* $N = 2$*, and so on, until the user-specified limit.*

Debugging techniques can be classified into BDD-based, simulation-based and satisfiability-based methods. BDD-based approaches [55, 56, 70] build and solve a so-called *error equation*,

which states whether a circuit can be modified at a certain location to satisfy its specification. This method has poor scalability due to the use of BDDs. Furthermore, its applicability to multiple errors has been shown to be limited.

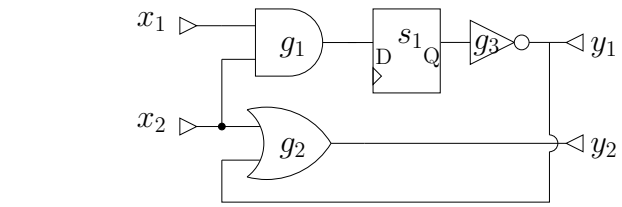
Simulation-based approaches [56, 73] use algorithms that trace backwards from the primary output values in the counter-example to prune suspect nodes using different criteria. Then simulations are performed to verify that each suspect line can correct the design. Such methods are only applicable to gate-level debugging and even then are not able to prune out large parts of sequential designs given long counter-examples.

3.2.1 SAT-based Automated Design Debugging

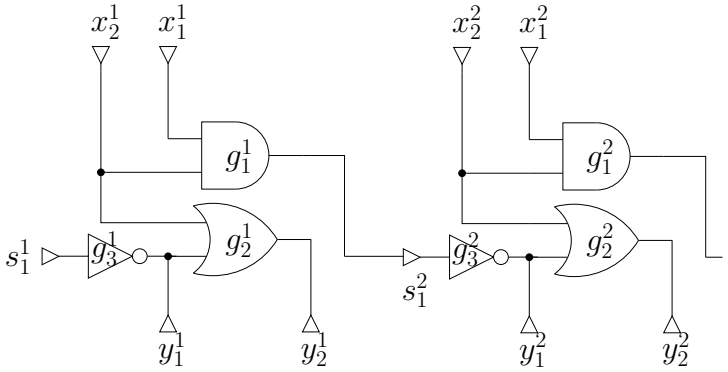
In 2004, capitalizing on the major advances in SAT solvers, a SAT-based automated debugging technique was proposed by [111]. It encodes combinational gate-level debugging as a SAT problem, where each satisfying assignment corresponds to a debugging solution. In recent years, extensions and improved debugging formulations using a variety of formal techniques have been proposed, building on the initial work of [111]. Sequential circuits are handled in [41, 110]. The work in [6] extends the formulation to deal with multiple-output circuit blocks in a hierarchical design, which makes it possible to debug RTL code. Several works improve the scalability of formal debugging techniques, such as QBF-based formulations [79], bounded model debugging [61], abstraction and refinement [100] and debugging using interpolants [63]. Extensive experiments show that satisfiability-based debugging techniques outperform traditional approaches by orders of magnitude [117].

SAT-based RTL debug tools start by synthesizing the erroneous RTL design into a gate-level sequential circuit $\mathcal{C}$. The symbols $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{s}$ respectively refer to the set of primary inputs, primary outputs and state elements (flip-flops) of $\mathcal{C}$. For each $\mathbf{z} \in \{\mathbf{x}, \mathbf{y}, \mathbf{s}, \ldots\}$, the Boolean variable z_i denotes the ith element in the set $\mathbf{z}$. In general, bold ($\mathbf{z}$) versus regular (z) symbols differentiate sets from single variables. We use the term *node* to refer to any gate, primary input or state element (primary outputs are simply labels, not separate nodes).

Given a single clock domain, *time-frame expansion* for k clock-cycles is the process of replicating, or *unrolling*, the combinational component of $\mathcal{C}$ k times, such that the next-state of



(a) $\mathcal{C}$



(b) Two time-frame ILA of $\mathcal{C}$

Figure 3.2: Time-frame expansion

each clock-cycle, or *time-frame*, is connected to the current-state of the next time-frame, thus modeling the sequential behavior of $\mathcal{C}$. For any variable (or set of variables) z_i (or $\mathbf{z}$), symbol z_i^t (or $\mathbf{z}^t$) denotes the corresponding variable (or set of variables) in time-frame t of the unrolled circuit. An unrolled circuit is also referred to as an *iterative logic array* (ILA). Figure 3.2 shows a sequential circuit and its corresponding two time-frame ILA. The behavior of $\mathcal{C}$ during the clock-cycle t is formalized using the *transition relation* predicate $T(\mathbf{s}^t, \mathbf{s}^{t+1}, \mathbf{x}^t, \mathbf{y}^t)$, which describes the dependence of the primary outputs $\mathbf{y}^t$ and next-state $\mathbf{s}^{t+1}$ on the primary inputs $\mathbf{x}^t$ and current-state $\mathbf{s}^t$. The transition relation T for time-frame t is simply the CNF formula representing the combinational component of $\mathcal{C}$, where the current state variables are $\mathbf{s}^t$ and the next-state variables are $\mathbf{s}^{t+1}$.

In hierarchical debugging, each RTL block (*e.g.,* a line of RTL code, a Verilog *always* block, an *if* statement, a module instantiation, etc) is a separate suspect that must be examined. For each RTL block, its corresponding set of synthesized gates in $\mathcal{C}$ is referred to as a circuit block, or simply *block*. This one-to-one correspondence between RTL blocks and circuit blocks

is maintained, and when a given circuit block is found as a solution by the SAT solver, its corresponding RTL block is returned to the user. Let $\mathcal{B} = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_{|\mathcal{B}|}\}$ denote the set of all suspect (circuit) blocks in $\mathcal{C}$, where each $\mathbf{b}_i$ is a collection of nodes. Note that the same node can belong to more than one block in $\mathcal{B}$ because of the hierarchical nature of RTL. The set $out(\mathbf{b}_i)$ denotes the output nodes of block $\mathbf{b}_i$. In the unrolled circuit, the set $\mathbf{b}_i^t$ (respectively, $out(\mathbf{b}_i^t)$) contains the nodes (respectively, the output nodes) of block $\mathbf{b}_i$ in time-frame t. In the circuit in Figure 3.3, the blocks $\mathbf{b}_1$, $\mathbf{b}_2$, $\mathbf{b}_3$, $\mathbf{b}_4$, and $\mathbf{b}_5$ are shown in dotted boxes. For example, we have $\mathbf{b}_3 = \{g_1, g_2\}$, $\mathbf{b}_5 = \{g_1, s_1\}$, $out(\mathbf{b}_1) = \{x_1\}$, $out(\mathbf{b}_3) = \{g_1, g_2\}$ and $out(\mathbf{b}_4) = \{g_3\}$.

Given an erroneous design $\mathcal{C}$, a set of suspect blocks $\mathcal{B}$, a counter-example of length k (along with its expected outputs) and an error cardinality N, the task of an automated design debugger is to find all sets of N blocks that can be responsible for the counter-example. More precisely, each returned set of N blocks $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_N}\}$, where $\{i_1, \ldots, i_N\} \subseteq \{1, \ldots, |\mathcal{B}|\}$, can be modified to rectify the erroneous behavior exhibited in the counter-example. We refer to each such set of N blocks as a solution of cardinality N. SAT-based automated design debugging [6, 110] encodes the debugging problem as a CNF formula whose satisfying assignments correspond to debugging solutions. The following are the steps to translate design debugging into a SAT problem. We use $\mathcal{C}$ and $\mathcal{B}$ given in Figure 3.3 as an example for illustrating the encoding process.

First, a set of *error-select* variables $\mathbf{e} = \{e_1, \ldots, e_{|\mathcal{B}|}\}$ are created with the following purpose: Setting $e_i = \texttt{1}$ must disconnect the fanouts of the gates in $out(\mathbf{b}_i)$ from their fanins, making them free variables, whereas setting $e_i = \texttt{0}$ must not modify the circuit. There are many ways to achieve this. The simplest is by inserting special multiplexers with select line e_i at the

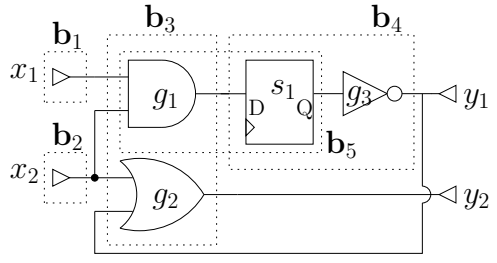


Figure 3.3: Blocks $\mathcal{B}$ in $\mathcal{C}$ of Figure 3.2(a)

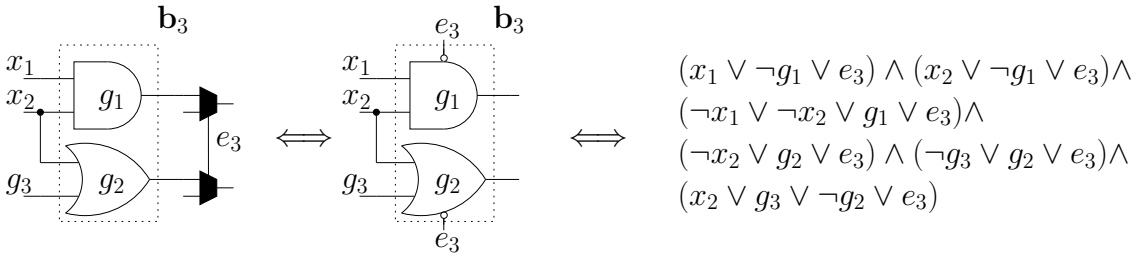


Figure 3.4: The usage of error-select variable e_3

output nodes of each block $\mathbf{b}_i$, such that if $e_i = \texttt{1}$, the outputs of $\mathbf{b}_i$ become unconstrained variables. The black `MUX`s at the outputs of block $\mathbf{b}_3$ shown in the leftmost circuitry in Figure 3.4 illustrate this. For simplicity, we represent this using the equivalent circuitry shown at the center of Figure 3.4. Furthermore, the same constraints can be achieved more efficiently by adding the literal e_3 directly in the clauses corresponding to these gates in the CNF of the transition relation T, as shown in Figure 3.4. Setting $e_3 = \texttt{1}$ would satisfy these clauses, and therefore the constraints of the gates in $out(\mathbf{b}_3) = \{g_1, g_2\}$, making the outputs of $\mathbf{b}_3$ free variables.

The CNF formula of the combinational component of this enhanced circuit is now denoted by the transition relation predicate $T_{en}(\mathbf{s}, \mathbf{s}', \mathbf{x}, \mathbf{y}, \mathbf{e})$, where $\mathbf{s}$ and $\mathbf{s}'$ denote the current-state and next-state variables. Next, T_{en} is replicated using time-frame expansion for the length of the counter-example k, and such that for all time-frames t, $out(\mathbf{b}_i^t)$ are controlled by the same error-select variable e_i. Figure 3.5 illustrates this, where each e_i is shown as an enable on the side of the gates in $out(\mathbf{b}_i^t)$, across all time-frames t. This allows the SAT solver to modify the outputs of block $\mathbf{b}_i$ *across all time-frames* by setting $e_i = \texttt{1}$ to fix any potential errors in $\mathbf{b}_i$. We let $\mathcal{U}$ refer to this enhanced ILA:

$$\mathcal{U} = \bigwedge_{t=1}^{k} T_{en}(\mathbf{s}^t, \mathbf{s}^{t+1}, \mathbf{x}^t, \mathbf{y}^t, \mathbf{e}) \tag{3.1}$$

Then, a set of constraints are applied to the initial state, the primary inputs and primary outputs of $\mathcal{U}$ in order to ensure that the counter-example is corrected. In detail, $\Phi_S(\mathbf{s}^1)$ (respectively, $\Phi_X(\mathbf{x}^1, \ldots, \mathbf{x}^k)$ and $\Phi_Y(\mathbf{y}^1, \ldots, \mathbf{y}^k)$) constrains the initial-state (respectively, primary inputs and primary outputs) of $\mathcal{U}$ to the initial-state (respectively, primary input sequence and *expected* primary output values) in the counter-example. Φ_S, Φ_X and Φ_Y are normally con-

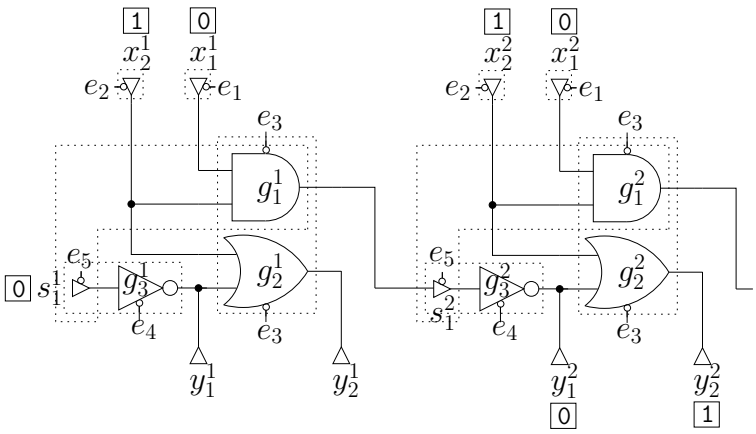


Figure 3.5: Design debugging formulation

junctions of unit clauses constraining the initial-state variables, primary inputs and primary outputs of the unrolled ILA. In general, Φ_Y can also express a set of properties that the design must satisfy [62].

Finally, an error cardinality constraint $\Phi_N(\mathbf{e})$ is conjuncted to the problem, forcing $\sum_{i=1}^{|\mathcal{B}|} e_i$ to a pre-specified constant N. This constraint can be enforced using a variety of techniques, such as an adder [110] or a bitonic sorter network [39]. The resulting propositional formula is given by:

$$Debug = \bigwedge_{t=1}^{k} T_{en}(\mathbf{s}^t, \mathbf{s}^{t+1}, \mathbf{x}^t, \mathbf{y}^t, \mathbf{e}) \wedge \Phi_S(\mathbf{s}^1) \wedge \Phi_X(\mathbf{x}^1, \ldots, \mathbf{x}^k) \wedge \Phi_Y(\mathbf{y}^1, \ldots, \mathbf{y}^k) \wedge \Phi_N(\mathbf{e}) \quad (3.2)$$

where $T_{en}(\mathbf{s}^t, \mathbf{s}^{t+1}, \mathbf{x}^t, \mathbf{y}^t, \mathbf{e})$ refers to the transition relation predicate of the enhanced circuit at time-frame t.

Each assignment to $\mathbf{e} = \{e_1, \ldots, e_{|\mathcal{B}|}\}$ satisfying *Debug* (3.2) corresponds to a debugging solution, and the SAT solver must find *all* such satisfying assignments to $\mathbf{e}$. This is normally done by iteratively blocking each satisfying assignment using a blocking clause and re-solving *Debug* until the problem becomes `UNSAT`. In a satisfying assignment where some $e_i = \mathtt{1}$, the values of $out(\mathbf{b}_i^t)$ across all time-frames t represent a sequence of *corrections*, which would correct the erroneous behavior in the counter-example. Note that *Debug* (3.2) allows these corrections to be non-deterministic functions of the applied primary inputs. In other terms, a correction can set a gate output to different values at different time-frames, given the same primary inputs.

Although it is possible to force deterministic corrections by adding extra constraints, this is not done usually because of the associated high cost and the rare occurrence of non-deterministic corrections in practice.

Example 3.2 *Consider the sequential circuit in Figure 3.3 to be a buggy implementation. We are also given a two-cycle counter-example with initial state* $s_1 = 0$*, inputs* $\langle x_1, x_2 \rangle = \langle \langle 0, 1 \rangle, \langle 0, 1 \rangle \rangle$ *and expected outputs in the second time-frame* $\langle y_1, y_2 \rangle = \langle 0, 1 \rangle$*, demonstrating a mismatch at primary output* y_1 *during the second clock-cycle.*

The corresponding design debugging formulation is illustrated in Figure 3.5. The constraints $\Phi_S = \neg s_1^1$*,* $\Phi_X = \neg x_1^1 \wedge x_2^1 \wedge \neg x_1^2 \wedge x_2^2$ *and* $\Phi_Y = \neg y_1^2 \wedge y_2^2$ *are shown in boxes next to the initial-state, primary inputs and outputs, while* Φ_N *is omitted for brevity. For* $N = 1$*, each of* $\{\mathbf{b}_1\}$*,* $\{\mathbf{b}_3\}$*,* $\{\mathbf{b}_4\}$ *and* $\{\mathbf{b}_5\}$ *will be returned by the solver as separate solutions. Corrections for solution* $\{\mathbf{b}_1\}$ *(respectively* $\{\mathbf{b}_3\}$*,* $\{\mathbf{b}_4\}$*,* $\{\mathbf{b}_5\}$*) consist of the satisfying assignments to* $\{x_1\}$ *(respectively* $\{g_1, g_2\}$*,* $\{g_3\}$*,* $\{s_1\}$*) during the two time-frames. For instance, in any correction for solution* $\{\mathbf{b}_1\}$*,* x_1^1 *must be set to* 1*, whereas* x_1^2 *is a don't-care.*

Typically, SAT-based RTL debug tools return all solutions for the smallest N for which *Debug* is SAT. This is done by first solving *Debug* with $N = 1$, then $N = 2$, and so on, until a user-specified maximum error cardinality $N = N_{max}$. If *Debug* is SAT for a given N, all other solutions of the same error cardinality are returned and the process stops. The debugger is sometimes asked to keep returning all solutions of up to $N = N_{max}$. In this case, blocking clauses that are added for solutions of a given error cardinality must remain in *Debug* for higher error cardinalities. Otherwise, if $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_N}\}$ is a solution of cardinality N, for all $\mathbf{b}_j \in \mathcal{B} - \{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_N}\}$, the set of blocks $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_N}, \mathbf{b}_j\}$ is a solution of cardinality $N + 1$, which is clearly unwanted.

Hybrid strategies are often used to guide the solver towards certain areas of the RTL or to return solutions with varying levels of granularity. For example, if a large RTL block $\mathbf{b}_i$ is found as an $N = 1$ solution of *Debug*, but none of its contained blocks are $N = 1$ solutions, this can be an indication of multiple errors in $\mathbf{b}_i$. As such, the set of considered suspect blocks $\mathcal{B}$ can be refined to include only those sub-blocks whose gates are contained in block $\mathbf{b}_i$, and N can

be subsequently increased to 2 (or more) in an effort to find higher error cardinality solutions of finer granularity contained in $\mathbf{b}_i$.

3.3 On the Complexity of RTL Debug

In order to prove complexity results related to design debugging, we must consider its corresponding decision problem that asks whether there exists a solution that can fix the counter-example. This problem is no harder than the original debugging problem which must return all such solutions. This works in our favor when we are proving hardness: If the decision version is hard, then all-solution design debugging is at least as hard.

Let $\Phi = \Phi_S \wedge \Phi_X \wedge \Phi_Y$ denote the counter-example consisting of initial state, primary input and expected primary output constraints. Also, let k refer to the number of cycles, or *length*, of the counter-example. Here, Φ_S (respectively, Φ_X and Φ_Y) is a conjunction of unit clauses on the initial-state variables $\mathbf{s}^1$ (respectively, the sequence of primary inputs $\mathbf{x}^1, \ldots, \mathbf{x}^k$, and the sequence of primary outputs $\mathbf{y}^1, \ldots, \mathbf{y}^k$). We will use the notation $s_i^1 \in \Phi_S$ to indicate that Φ_S constrains bit s_i of the initial state. In other terms, if Φ_S contains one of (respectively, does not contain either of) the unit clauses (s_i^1) or $(\neg s_i^1)$, then $s_i^1 \in \Phi_S$ (respectively, $s_i^1 \notin \Phi_S$). Similarly, $x_i^t \in \Phi_X$ (respectively, $y_i^t \in \Phi_Y$) if and only if Φ_X (respectively, Φ_Y) constrains the ith primary input (respectively, output) during the tth time-frame.

Now we can define the DEBUG language as follows:

$$\begin{aligned}\text{DEBUG} = \{\langle \mathcal{C}, \Phi, N, \mathcal{B} \rangle \mid & \text{ Given the freedom to simultaneously and arbitrarily modify the outputs of } N \text{ suspect blocks in } \mathcal{B}, \text{ across } k \text{ time-frames in the execution} \\ & \text{of circuit } \mathcal{C} \text{ under the counter-example initial state } \Phi_S \text{ and primary input sequence } \Phi_X, \text{ it is possible to get the expected primary output values given by } \Phi_Y\}\end{aligned} \tag{3.3}$$

We can equivalently describe DEBUG by referring to its SAT formulation as follows:

$$\text{DEBUG} = \{\langle \mathcal{C}, \Phi, N, \mathcal{B} \rangle \mid \text{Given circuit } \mathcal{C}\text{, counter-example } \Phi\text{, error cardinality } N \text{ and suspect blocks } \mathcal{B}\text{, } \textit{Debug} \text{ given by Equation (3.2) is } \texttt{SAT}\} \tag{3.4}$$

A problem instance of DEBUG is of the form $\langle \mathcal{C}, \Phi, N, \mathcal{B} \rangle$. The size of a debug instance $|\langle \mathcal{C}, \Phi, N, \mathcal{B} \rangle|$ is therefore the sum of (1) $|\langle \mathcal{C} \rangle|$ (*i.e.,* the size of encoding $\mathcal{C}$), which is polynomial in its number of nodes, (2) $|\langle \Phi \rangle| = |\langle \Phi_S \rangle| + |\langle \Phi_X \rangle| + |\langle \Phi_Y \rangle| = O(|\mathbf{s}| + k \cdot (|\mathbf{x}| + |\mathbf{y}|))$,[1] (3) $|\langle N \rangle| = O(\log N)$, and (4) $|\langle \mathcal{B} \rangle|$.

Instead of simply proving that the DEBUG language given by (3.4) is NP-complete, we will establish the line where debugging moves from the complexity class P to that of NP-completeness by introducing four independent restrictions or assumptions on the types of instances $\langle \mathcal{C}, \Phi, N, \mathcal{B} \rangle$ allowed in DEBUG, as follows:

(a) $\mathcal{C}$ is combinational:

$|\mathbf{s}| = 0$.

(b) Every suspect block in $\mathcal{B}$ has exactly one output:

$\forall \mathbf{b}_i \in \mathcal{B}\ .\ (|out(\mathbf{b}_i)| = 1)$.

(c) No primary inputs can be unassigned throughout the counter-example, neither can any initial-state variable be unassigned in the first time-frame:

$[\forall s \in \mathbf{s}^1.(s \in \Phi_S)] \wedge [\forall x \in \mathbf{x}^1 \cup \cdots \cup \mathbf{x}^k.(x \in \Phi_X)]$.

(d) Single error cardinality:

$N = 1$.

By including or excluding any of these four restrictions on debugging instances, we obtain $2^4 = 16$ variations of the DEBUG problem. We let DEBUG[a,b,c,d] with $a, b, c, d \in \{\texttt{0}, \texttt{1}\}$

[1] In a naïve encoding, each state element (respectively, primary input and primary output) constraint in Φ can take $O(\log |\mathbf{s}|)$ (respectively, $O(\log |\mathbf{x}|)$ and $O(\log |\mathbf{y}|)$) space. The reason is that for each unit clause in, for example Φ_S, $O(\log |\mathbf{s}|)$ bits are needed to identify or index the particular state element in $\mathbf{s}$ that the unit clause is assigning, and analogously for Φ_X and Φ_Y. This would yield a total counter-example encoding size of $|\langle \Phi \rangle| = O(|\mathbf{s}| \cdot \log |\mathbf{s}| + k \cdot (|\mathbf{x}| \cdot \log |\mathbf{x}| + |\mathbf{y}| \cdot \log |\mathbf{y}|))$. However, it is possible to order the constraints in Φ according to the order in which the state elements, primary inputs and primary outputs appear in $\langle \mathcal{C} \rangle$. Doing this would eliminate the need to identify each variable in the encoding of Φ, resulting in $|\langle \Phi \rangle| = O(|\mathbf{s}| + k \cdot (|\mathbf{x}| + |\mathbf{y}|))$. Logarithmic factors such as the difference between the former and latter encodings do not affect our proofs.

denote these DEBUG variations, where setting $a = \texttt{1}$ (respectively, $a = \texttt{0}$) indicates that restriction (a) applies (respectively, does not have to apply), and so on. For instance, the language DEBUG[1,0,0,1] refers to the debugging language restricted to (a) combinational circuits, and (d) single error cardinality solutions. Also DEBUG[0,0,0,0] = DEBUG given by (3.4), and DEBUG[0,1,0,0] is essentially gate-level debugging.[2] We will prove that DEBUG[1,1,1,1] $\in$ P, and that $\forall\langle a,b,c,d\rangle \in \{\texttt{0},\texttt{1}\}^4 - \{\langle \texttt{1},\texttt{1},\texttt{1},\texttt{1}\rangle\}$, DEBUG[$a$,$b$,$c$,$d$] $\in$ NPC. Note that the fact that a given variation of DEBUG can be reduced to a SAT problem does not prove that it is NP-complete. We need to reduce a known NP-complete problem to the particular variation of DEBUG instead.

Theorem 3.1 *Consider the following language:*

$$\textit{DEBUG[1,1,1,1]} = \{\langle \mathcal{C}, \Phi, 1, \mathcal{B}\rangle \mid \textit{Given combinational circuit } \mathcal{C}\textit{, one time-frame counter-example } \Phi \textit{ assigning all primary inputs, error cardinality } N = 1 \textit{ and single-output suspect blocks } \mathcal{B}\textit{, Debug (3.2) is } \textit{SAT}\} \tag{3.5}$$

DEBUG[1,1,1,1] $\in$ *P.*

Proof:

We devise an algorithm that decides DEBUG[1,1,1,1] in polynomial-time in $|\langle \mathcal{C}, \Phi, 1, \mathcal{B}\rangle|$. Since Φ_X constrains all primary inputs, the circuit $\mathcal{C}$ can be fully assigned by propagating Φ_X. Given that each suspect is a single-output block, the only way to change the functionality of a suspect is to flip its output bit. Since $N = 1$, one output bit can be flipped at a time. Flipping a bit, propagating it in the circuit and checking whether Φ_Y has been satisfied takes polynomial-time in $|\langle \mathcal{C}\rangle| + |\langle \Phi\rangle|$. Our algorithm does this separately for $\mathcal{B}$ blocks, one at a time, which takes polynomial-time in $|\langle \mathcal{C}\rangle| + |\langle \Phi\rangle| + |\langle \mathcal{B}\rangle|$. The algorithm accepts if and only if at least one of these bit-flips can yield the expected primary output values given by Φ_Y. This

[2] The only difference between them is that in DEBUG[0,1,0,0], a suspect block can be an arbitrary single-output function, and not necessarily a primitive logic gate, such as `AND` or `OR`. However, this difference is irrelevant to the computational complexity of both problems because the design debugging problem only examines the outputs of a suspect block and disregards its inputs.

takes polynomial-time in the input $|\langle \mathcal{C}, \Phi, 1, \mathcal{B} \rangle|$, and hence DEBUG[1,1,1,1] $\in$ P.

■

Lemma 3.2 $\forall \langle a, b, c, d \rangle \in \{0, 1\}^4$, *DEBUG[a,b,c,d]* $\in$ *NP.*

Proof: For all variations of DEBUG, a certificate consists of (1) the truth assignment to all the output variables of a given set of N blocks across the k time-frames of the counter-example, and (2) assignments to the unconstrained state elements in Φ_S (if any exist) and primary inputs in Φ_X (if any exist). This certificate is clearly of polynomial-size in $|\langle \mathcal{C}, \Phi, N, \mathcal{B} \rangle|$. The verification algorithm A must simply circuit-simulate the counter-example constraints in Φ_S and Φ_X, along with (1) and (2) in the certificate, which together assign all free variables in the k-time-frame unrolled circuit, then verify whether the resulting primary output values conform to Φ_Y. Algorithm A clearly takes polynomial-time in the input $|\langle \mathcal{C}, \Phi, N, \mathcal{B} \rangle|$.

If a given problem instance $\langle \mathcal{C}, \Phi, N, \mathcal{B} \rangle$ belongs to the considered DEBUG variation, the corresponding debugging SAT formulation (3.2) is SAT by definition, and therefore it is easy to see that there will exist such a certificate (it can be extracted from the satisfying assignment to (3.2)). On the other hand, no such certificate exists for debugging instances that are UNSAT. As such, $\forall \langle a, b, c, d \rangle \in \{0, 1\}^4$, DEBUG[$a$,$b$,$c$,$d$] $\in$ NP.

■

Theorem 3.3 *Consider the following language:*

$$\textit{DEBUG[0,1,1,1]} = \{\langle \mathcal{C}, \Phi, 1, \mathcal{B} \rangle \mid \textit{Given circuit } \mathcal{C} \textit{ (sequential or combinational), counter-example } \Phi \textit{ assigning all input stimuli, error cardinality } N = 1 \textit{ and single-output suspect blocks } \mathcal{B}\textit{, Debug (3.2) is SAT}\} \tag{3.6}$$

DEBUG[0,1,1,1] $\in$ *NPC.*

Proof:

We will show that CIRCUIT-SAT $\leq_p$ DEBUG[0,1,1,1] by giving a polynomial-time reduction f that maps instances of CIRCUIT-SAT into instances of DEBUG[0,1,1,1]. By Definition 2.3, for all $\langle \mathcal{C} \rangle \in \{0,1\}^*$, the reduction function $f : \langle \mathcal{C} \rangle \rightarrow \langle \mathcal{C}', \Phi, 1, \mathcal{B} \rangle$ must satisfy:

$$\langle \mathcal{C} \rangle \in \text{ CIRCUIT-SAT } \iff f(\langle \mathcal{C} \rangle) = \langle \mathcal{C}', \Phi, 1, \mathcal{B} \rangle \in \text{ DEBUG[0,1,1,1]}. \tag{3.7}$$

Given an instance of CIRCUIT-SAT, which is a single-output combinational circuit $\langle \mathcal{C} \rangle$, Figure 3.6 shows how we build the circuit $\langle \mathcal{C}' \rangle$ in the corresponding DEBUG[0,1,1,1] instance. In $\mathcal{C}'$, x_1 is the only primary input. The other primary inputs of $\mathcal{C}$, namely $x_2, \ldots, x_n$ are turned into state elements $s_1, \ldots, s_{n-1}$ in $\mathcal{C}'$, respectively, where s_1 is a shifted version of x_1 by one cycle (using a flip-flop), s_2 is a shifted version of s_1 by one cycle, s_3 is a shifted version of s_2 by one cycle, and so on. The set of suspect blocks $\mathcal{B}$ in the reduction consists of one block, $\mathcal{B} = \{\mathbf{b}_1\}$, where $\mathbf{b}_1 = \{x_1\}$, as shown in Figure 3.6.

Finally, the counter-example in the reduced instance consists of $k = n$ cycles, where n is the number of primary inputs of $\mathcal{C}$, and its constraints Φ are given by:

$$\Phi_S = s_1^1 \wedge s_2^1 \wedge \cdots \wedge s_{n-1}^1$$

$$\Phi_X = x_1^1 \wedge x_1^2 \wedge \cdots \wedge x_1^n$$

$$\Phi_Y = y_1^n$$

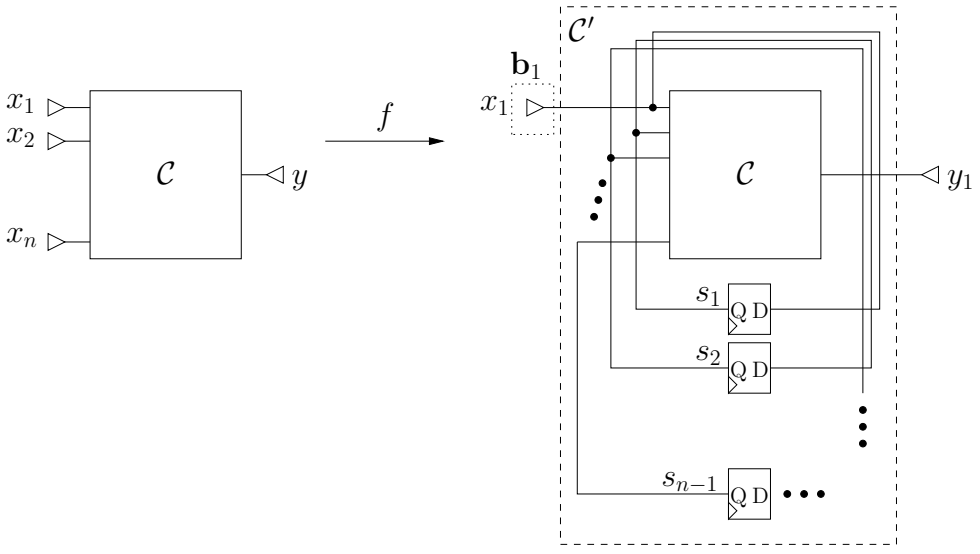


Figure 3.6: Constructing $\mathcal{C}'$ and $\mathcal{B}$ for DEBUG[0,1,1,1]

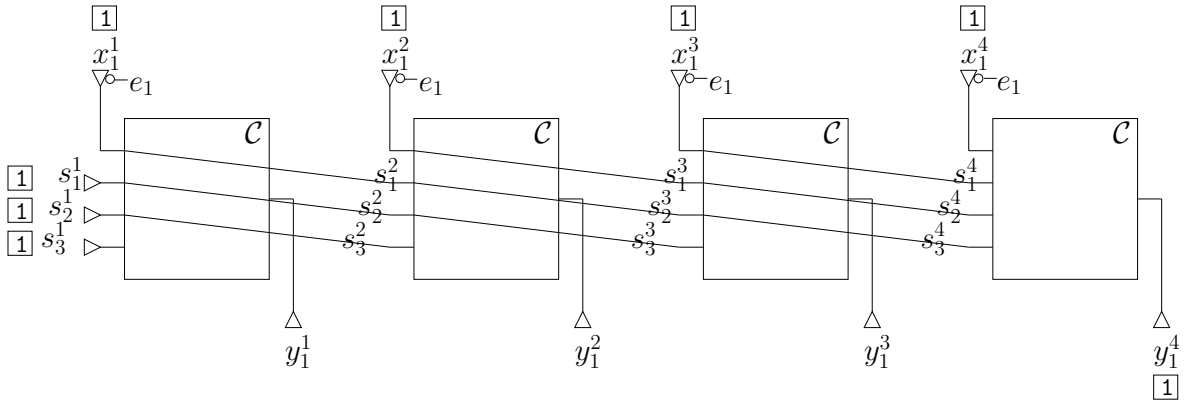


Figure 3.7: SAT-based formulation given $\langle \mathcal{C}', \Phi, 1, \mathcal{B} \rangle$ for DEBUG[0,1,1,1]

In reality, we will see that Φ_S and Φ_X can be arbitrary, as long as Φ_Y forces the output of the last time-frame of $\mathcal{C}'$ to 1.

Now, given any $\langle \mathcal{C} \rangle$ and its corresponding reduction $\langle \mathcal{C}', \Phi, 1, \mathcal{B} \rangle$ as described above, we need to show that (3.7) holds. That is, we must show that $\mathcal{C}$ is SAT if and only if *Debug* given by (3.2) is SAT using our constructed debugging instance $\langle \mathcal{C}', \Phi, 1, \mathcal{B} \rangle$. We illustrate that this is true using an example.

Consider an arbitrary circuit $\mathcal{C}$ with four primary inputs x_1, x_2, x_3 and x_4. Then, the SAT-based debugging formulation using $f(\langle \mathcal{C} \rangle) = \langle \mathcal{C}', \Phi, 1, \mathcal{B} \rangle$ is shown in Figure 3.7. As in Figure 3.5, the error cardinality constraints Φ_N, in this case forcing the only error-select line $e_1 = 1$, is not shown in Figure 3.7. Setting $e_1 = 1$ makes x_1^1, x_1^2, x_1^3 and x_1^4 free variables. Due to the flip-flops, the inputs of $\mathcal{C}$ during the fourth time-frame are connected to these free variables. Since the only output constraint in Φ_Y is on the primary output of this fourth time-frame, the enhanced ILA in Figure 3.7 will only be SAT if there exists a way to assign x_1^1, x_1^2, x_1^3 and x_1^4 to make $y_1 = 1$, which is true exactly when $\mathcal{C}$ is SAT. Hence, (3.7) holds. The same argument applies when $\mathcal{C}$ has n inputs instead of four.

The mapping f clearly takes polynomial-time in $|\langle \mathcal{C} \rangle|$. Therefore, CIRCUIT-SAT $\leq_p$ DEBUG[0,1,1,1]. Since CIRCUIT-SAT is known to be NP-complete, this implies that DEBUG[0,1,1,1] is NP-hard. But DEBUG[0,1,1,1] $\in$ NP by Lemma 3.2, hence it is NP-complete.

■

Corollary 3.4 $\forall\langle b,c,d\rangle \in \{0,1\}^3$, *DEBUG[0,b,c,d]* $\in$ *NPC.*

Proof: If we relax or exclude any of conditions (b), (c) and (d) from DEBUG[0,1,1,1], we can still use the same mapping f given in the proof of Theorem 3.3 that reduces instances of CIRCUIT-SAT. Note that, for example, such a mapping would not yield instances of DEBUG[0,1,1,0] with $N > 1$, but this is irrelevant because f does not have to be an onto function. Therefore, $\forall\langle b,c,d\rangle \in \{0,1\}^3$, DEBUG[0,$b$,$c$,$d$] $\in$ NPC.

■

Theorem 3.5 *Consider the following language:*

$$\textit{DEBUG[1,0,1,1]} = \{\langle \mathcal{C}, \Phi, 1, \mathcal{B}\rangle \mid \textit{Given combinational circuit } \mathcal{C}\textit{, one time-frame counter-example } \Phi \textit{ assigning all primary inputs, error cardinality } N = 1 \textit{ and arbitrary suspect blocks } \mathcal{B}\textit{, Debug (3.2) is SAT}\} \tag{3.8}$$

DEBUG[1,0,1,1] $\in$ *NP.*

Proof:

We will show that CIRCUIT-SAT $\leq_p$ DEBUG[1,0,1,1] by giving a polynomial-time reduction f that maps instances of CIRCUIT-SAT into instances of DEBUG[1,0,1,1] . Again, for all $\langle\mathcal{C}\rangle \in \{0,1\}^*$, $f : \langle\mathcal{C}\rangle \rightarrow \langle\mathcal{C}', \Phi, 1, \mathcal{B}\rangle$ must satisfy:

$$\langle\mathcal{C}\rangle \in \text{ CIRCUIT-SAT } \Longleftrightarrow f(\langle\mathcal{C}\rangle) = \langle\mathcal{C}', \Phi, 1, \mathcal{B}\rangle \in \text{ DEBUG[1,0,1,1] .} \tag{3.9}$$

Given an instance $\langle\mathcal{C}\rangle$ of CIRCUIT-SAT, the circuit $\langle\mathcal{C}'\rangle$ in our reduction is simply equal

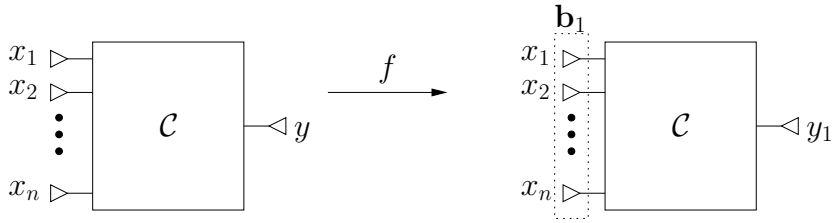


Figure 3.8: Constructing $\mathcal{C}'$ and $\mathcal{B}$ for DEBUG[1,0,1,1]

to $\langle \mathcal{C} \rangle$. The set of suspect blocks $\mathcal{B}$ consists of one block, $\{\mathbf{b}_1\}$, where $\mathbf{b}_1 = \{x_1, x_2, \ldots, x_n\}$, as shown in Figure 3.8.

Finally, the one-cycle counter-example consists of the following constraints Φ:

$$\Phi_S = \mathtt{1} \text{ (\textit{i.e.}, no initial-state constraints)}$$
$$\Phi_X = x_1^1 \wedge x_2^1 \wedge \cdots \wedge x_n^1$$
$$\Phi_Y = y_1^1$$

Again, Φ_X can be arbitrary, as long as Φ_Y forces the output of $\mathcal{C}'$ to 1.

If $\langle \mathcal{C} \rangle \in$ CIRCUIT-SAT, then by definition there exists a truth assignment π to its primary inputs making the output of $\mathcal{C}$ 1. In the mapped instance $\langle \mathcal{C}, \Phi, 1, \mathcal{B} \rangle$, the outputs of the only block $\mathbf{b}_1$ are the inputs to the same circuit $\mathcal{C}$ whose output is constrained to 1 by Φ_Y. As such, $\mathbf{b}_1$ is clearly a solution, since we know that assigning $out(\mathbf{b}_1)$ to π satisfies $\mathcal{C}$. And conversely, if $\langle \mathcal{C} \rangle \notin$ CIRCUIT-SAT, no such π exists, and $\langle \mathcal{C}, \Phi, 1, \mathcal{B} \rangle \notin$ DEBUG[1,0,1,1], as required. Hence, (3.9) holds.

The mapping f clearly takes polynomial-time in $|\langle \mathcal{C} \rangle|$. Therefore, CIRCUIT-SAT $\leq_p$ DEBUG[1,0,1,1]. Since CIRCUIT-SAT is known to be NP-complete, and along with Lemma 3.2, DEBUG[1,0,1,1] is also NP-complete.

■

Corollary 3.6 *$\forall \langle a, c, d \rangle \in \{0, 1\}^3$, DEBUG[a,0,c,d] $\in$ NPC.*

Proof: If we relax any of conditions (a), (c) and (d) from DEBUG[1,0,1,1], we can still use the same mapping f given in the proof of Theorem 3.5 to reduce instances of CIRCUIT-SAT.

■

Theorem 3.7 *Consider the following language:*

$$DEBUG[1,1,0,1] = \{\langle \mathcal{C}, \Phi, 1, \mathcal{B} \rangle \mid \textit{Given combinational circuit } \mathcal{C}\textit{, one time-frame counter-example } \Phi \textit{ where not all primary inputs need to be constrained, error cardinality } N = 1 \textit{ and single-output suspect blocks } \mathcal{B}\textit{, Debug (3.2) is } \mathtt{SAT}\} \tag{3.10}$$

DEBUG[1,1,0,1] $\in$ *NPC.*

Proof:

We will show that CIRCUIT-SAT $\leq_p$ DEBUG[1,1,0,1] by giving a polynomial-time reduction f that maps instances of CIRCUIT-SAT into instances of DEBUG[1,1,0,1]. For all $\langle \mathcal{C} \rangle \in \{0,1\}^*$, $f : \langle \mathcal{C} \rangle \rightarrow \langle \mathcal{C}', \Phi, 1, \mathcal{B} \rangle$ must satisfy:

$$\langle \mathcal{C} \rangle \in \text{ CIRCUIT-SAT } \iff f(\langle \mathcal{C} \rangle) = \langle \mathcal{C}', \Phi, 1, \mathcal{B} \rangle \in \text{ DEBUG[1,1,0,1]}. \tag{3.11}$$

Given an instance $\langle \mathcal{C} \rangle$ of CIRCUIT-SAT, the circuit $\mathcal{C}'$ in our reduction is shown in Figure 3.9. $\mathcal{C}'$ consists of $\mathcal{C}$ along with an extra BUFFER gate g_1 connected to a new primary input x_{n+1} and a new primary output y_2. The set of suspect blocks $\mathcal{B}$ consists of a single block $\mathbf{b}_1 = \{g_1\}$, as shown in Figure 3.9. Finally the one-cycle counter-example consists of the

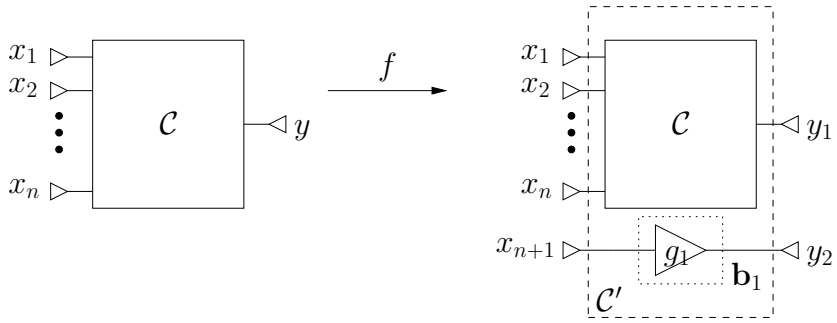


Figure 3.9: Constructing $\mathcal{C}'$ and $\mathcal{B}$ for DEBUG[1,1,0,1]

following constraints Φ:

$$\Phi_S = \mathtt{1} \text{ (\textit{i.e.}, no initial-state constraints)}$$
$$\Phi_X = \neg x_{n+1}^1$$
$$\Phi_Y = y_1^1 \wedge y_2^1$$

Notice that Φ_X does not constrain primary input $x_1, \ldots, x_n$ and only constrains x_{n+1} to 0. On the other hand, Φ_Y forces both outputs to 1.

If $\langle \mathcal{C} \rangle \in$ CIRCUIT-SAT, then there exists a truth assignment π to $x_1, \ldots, x_n$ evaluating the output of $\mathcal{C}$ to 1. In the mapped instance $\langle \mathcal{C}', \Phi, 1, \mathcal{B} \rangle$, $x_1, \ldots, x_n$ can be assigned to the same π, satisfying $y_1^1 = \mathtt{1}$. Also, making $e_1 = \mathtt{1}$ in Debug (3.2) allows the output of block $\mathbf{b}_1$, gate g_1, to be a free variable which can be assigned to 1 to trivially satisfy $y_2^1 = \mathtt{1}$. As such, the constructed problem instance $\langle \mathcal{C}', \Phi, 1, \mathcal{B} \rangle \in$ DEBUG[1,1,0,1]. Conversely, if $\langle \mathcal{C} \rangle \notin$ CIRCUIT-SAT, no such π exists, and it is not possible to satisfy the $y_1^1 = \mathtt{1}$ constraint in Φ_Y. Therefore, $\langle \mathcal{C}, \Phi, n, \mathcal{B} \rangle \notin$ DEBUG[1,1,0,1], as required. Hence, (3.11) holds.

The described mapping f clearly takes polynomial-time in $|\langle \mathcal{C} \rangle|$. Therefore, CIRCUIT-SAT $\leq_p$ DEBUG[1,1,0,1]. Since CIRCUIT-SAT is known to be NP-complete, and along with Lemma 3.2, DEBUG[1,1,0,1] is also NP-complete.

■

Corollary 3.8 *$\forall \langle a, b, d \rangle \in \{0, 1\}^3$, DEBUG[a,b,0,d] $\in$ NP-C.*

Proof: If we relax any of conditions (a), (b) and (d) from DEBUG[1,1,0,1], we can still use the same mapping given in the proof of Theorem 3.7 to reduce instances of CIRCUIT-SAT.

■

Theorem 3.9 *Consider the following language:*

$$\textit{DEBUG[1,1,1,0]} = \{\langle \mathcal{C}, \Phi, N, \mathcal{B} \rangle \mid \textit{Given combinational circuit } \mathcal{C}\textit{, one time-frame counter-example } \Phi \textit{ assigning all primary inputs, error cardinality } N \textit{ and single-output suspect blocks } \mathcal{B}\textit{, Debug (3.2) is } \mathtt{SAT}\} \tag{3.12}$$

DEBUG[1,1,1,0] ∈ NPC.

Proof:

We will show that CIRCUIT-SAT $\leq_p$ DEBUG[1,1,1,0] by giving a polynomial-time reduction f that maps instances of CIRCUIT-SAT into instances of DEBUG[1,1,1,0]. For all $\langle \mathcal{C} \rangle \in \{0,1\}^*$, $f : \langle \mathcal{C} \rangle \rightarrow \langle \mathcal{C}', \Phi, N, \mathcal{B} \rangle$ must satisfy:

$$\langle \mathcal{C} \rangle \in \text{ CIRCUIT-SAT } \iff f(\langle \mathcal{C} \rangle) = \langle \mathcal{C}', \Phi, N, \mathcal{B} \rangle \in \text{ DEBUG[1,1,1,0]}. \tag{3.13}$$

Given an instance $\langle \mathcal{C} \rangle$ of CIRCUIT-SAT, the circuit $\langle \mathcal{C}' \rangle$ in our reduction is the same circuit $\langle \mathcal{C} \rangle$. The set of suspect blocks $\mathcal{B}$ consists of n blocks, $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$, where $\mathbf{b}_1 = \{x_1\}, \ldots, \mathbf{b}_n = \{x_n\}$, as shown in Figure 3.10. Our reduction function f sets the error cardinality N to n. Finally the one-cycle counter-example consists of the following constraints Φ:

$$\Phi_S = 1 \text{ (\textit{i.e.,} no initial-state constraints)}$$
$$\Phi_X = x_1^1 \wedge x_2^1 \wedge \cdots \wedge x_n^1$$
$$\Phi_Y = y_1^1$$

Here, Φ_X can be arbitrary, as long as Φ_Y forces the output of $\mathcal{C}'$ to 1.

If $\langle \mathcal{C} \rangle \in$ CIRCUIT-SAT, then there exists a truth assignment π to its primary inputs making the output of $\mathcal{C}$ 1. In the mapped instance $\langle \mathcal{C}, \Phi, n, \mathcal{B} \rangle$, the outputs of the blocks $\mathbf{b}_1, \ldots, \mathbf{b}_n$ are the inputs to the same circuit $\mathcal{C}$ whose output is constrained to 1 by Φ_Y. As such, $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ is a cardinality-n solution, since we can change $N = n$ block outputs, and we know that assigning $x_1^1, \ldots, x_1^n$ to π satisfies Φ_Y. Conversely, if $\langle \mathcal{C} \rangle \notin$ CIRCUIT-SAT, no such π exists, and $\langle \mathcal{C}, \Phi, n, \mathcal{B} \rangle \notin$ DEBUG[1,1,1,0], as required. Hence, (3.13) holds.

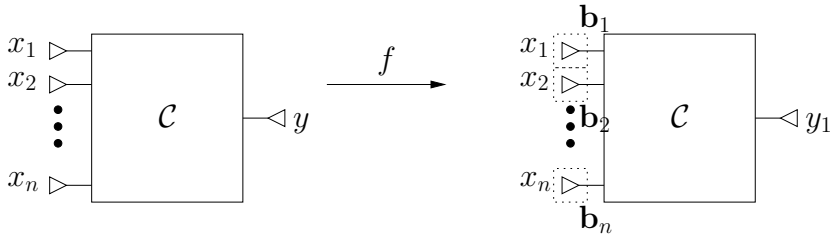


Figure 3.10: Constructing $\mathcal{C}'$ and $\mathcal{B}$ for DEBUG[1,1,1,0]

The described mapping f takes polynomial-time in $|\langle \mathcal{C} \rangle|$. Therefore, CIRCUIT-SAT $\leq_p$ DEBUG[1,1,1,0]. Since CIRCUIT-SAT is known to be NP-complete, and along with Lemma 3.2, DEBUG[1,1,1,0] is also NP-complete.

■

Corollary 3.10 $\forall \langle a, b, c \rangle \in \{0, 1\}^3$, *DEBUG[a,b,c,0]* $\in$ *NPC.*

Proof: If we relax any of conditions (a), (b) and (c) from DEBUG[1,1,1,0], we can still use the same mapping f given in the proof of Theorem 3.9 to reduce instances of CIRCUIT-SAT.

■

Therefore, every considered DEBUG variant, except the most restricted variation DEBUG[1,1,1,1] is NP-complete.

Corollary 3.11 $\forall \langle a, b, c, d \rangle \in \{0, 1\}^4 - \{\langle 1, 1, 1, 1 \rangle\}$, *DEBUG[a,b,c,d]* $\in$ *NPC.*

Proof: This is a direct implication of Corollaries 3.4, 3.6, 3.8 and 3.10.

■

3.4 Summary

This chapter proves several new results on the complexity of design debugging. Four factors affecting its theoretical computational complexity are identified. It is shown that, except in the most restricted scenario where the circuit is combinational, suspects are single-output blocks, no primary input or initial-state variable is unassigned in the counter-example, and solutions have single error cardinality, RTL debug is NP-complete.

Chapter 4

Debugging using Dominance

4.1 Introduction

An automated RTL debugger must return all potential bug locations, called *solutions*, which consist of RTL lines or blocks where *corrections* can rectify the erroneous behavior in the given counter-example. The engineer is given the final task of examining each solution and its correction in order to identify the real bug and fix it.

Modern debuggers make heavy use of formal tools, such as SAT [110], QBF [79] and maximum satisfiability [27]. In all these techniques, finding each solution requires a separate call to the formal engine. With typical design sizes exceeding the half-million synthesized gates mark, discovering up to hundreds of solutions [61] one-by-one is computationally expensive and limits the effectiveness of automated debuggers. This chapter addresses this issue by generating on-the-fly *implied* solutions, thus reducing the number of iterations for returning all solutions. This is done by using structural dominance relationships between circuit components.

A node u is said to be a (structural) *single-vertex dominator* of another node v if every path from v to a primary output passes through u. Single-vertex dominators can be found in linear-time [9, 45] and have been used for optimizing various CAD tasks, *e.g.*, test pattern generation [64, 89]. More recently, they have been leveraged in the gate-level debugger in [110], which performs an initial debugging pass on selected dominator gates. However, state-of-the-art automated design debuggers operate at the RTL-level [6, 100], where bugs occur in *multiple-*

vertex, multiple-output blocks in the circuit. As such, it is difficult to make use of single-vertex dominators at the RTL-level. A multiple-vertex block **a** dominates another multiple-vertex block **b** if every path from every node in **b** to a primary output passes through a node in **a**. Unlike existing approaches for finding multiple-vertex dominators, where block boundaries are not specified in advance [8, 54, 65], we are interested in establishing dominance relationships among a fixed set of blocks, naturally provided in a hierarchical RTL design.

The initial contribution of this work [82] is an algorithm that iteratively calculates dominance relationships between a predefined set of multiple-vertex blocks in a design. Next, it is proven that for each (set of) block(s) returned as a solution by the automated design debugger, every corresponding (set of) dominator(s) is a separate implied solution. As such, applying our algorithm as a preprocessing step, the number of design debugging iterations for finding all solutions can be significantly reduced. Furthermore, we prove that corrections for implied solutions can be automatically generated without explicitly analyzing these solutions. It is shown that dominator-based solution implications are guaranteed to be valid given any error cardinality.

The proposed method is conveniently presented and implemented on top of a SAT-based automated design debugging framework [6, 110]. However, it is also applicable to simulation-based and other formal diagnosis techniques. An extensive set of experiments on real industrial designs obtained by our partners demonstrates the consistent benefits of the presented framework. It is shown that 66% of solutions are discovered early due to dominator implications. This results in a 1.64x overall speed-up in solving time in an industrial environment, demonstrating the robustness of the proposed approach.

This chapter is organized as follows. Section 4.2 gives notation and preliminaries on dominators. Section 4.3 presents the iterative algorithm for computing dominance relationships between blocks. Section 4.4 shows how to leverage block dominators for early solution implications in design debugging. Section 4.5 gives experimental results and Section 4.6 summarizes the chapter.

4.2 Preliminaries

Subsection 3.2.1 gives a detailed illustration of the design debugging formulation as a SAT problem whose satisfying assignments correspond to debugging solutions. We refer the reader to that subsection for details on design debugging.

The following summarizes some of the notation in debugging used in Subsection 3.2.1 and presents some new notation. Given a sequential circuit $\mathcal{C}$, the symbol $\mathbf{n}$ denotes the set of all nodes in $\mathcal{C}$. The symbols $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{s}$ label (possibly overlapping) subsets of $\mathbf{n}$, respectively referring to the sets of primary inputs, primary outputs and state elements (flip-flops) of $\mathcal{C}$. For each $\mathbf{z} \in \{\mathbf{x}, \mathbf{y}, \mathbf{s}, \mathbf{n}\}$, the Boolean variable z_i denotes the ith element in the set $\mathbf{z}$.

We consider designs with single clock domains, although the described theory is applicable to multiple clock domains using the reduction from multiple clock domains to single clock domains in [44]. For any variable (or set of variables) z_i (or $\mathbf{z}$), symbol z_i^t (or $\mathbf{z}^t$) denotes the corresponding variable (or set of variables) in time-frame t of the unrolled circuit. The transition relation of $\mathcal{C}$ during time-frame t is given by $T(\mathbf{s}^t, \mathbf{s}^{t+1}, \mathbf{x}^t, \mathbf{y}^t)$.

The sequential circuit $\mathcal{C}$ can also be represented as a directed graph. For convenience, we add an artificial sink node r to this graph, such that the set of nodes $V = \mathbf{n} \cup \{r\}$ and the set of edges $E = \{(n_i, n_j) | n_i \text{ is a fanin of } n_j \text{ in } \mathcal{C}\} \cup \{(y_i, r) | \forall y_i \in y\}$. We reserve the letters u and v to refer to nodes in V. Let $fanout(v) = \{u \in V | (v, u) \in E\}$ and $fanin(v) = \{u \in V | (u, v) \in E\}$. Furthermore, the nodes $\mathbf{n}$ of $\mathcal{C}$ are grouped into (possibly overlapping) *blocks*. Each block consists of the synthesized gates of a given block of RTL code, such as an *always* block in Verilog. Let $\mathcal{B} = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_{|\mathcal{B}|}\}$ denote the set of all blocks, where each $\mathbf{b}_i \subseteq \mathbf{n}$ is a collection of nodes. The set $out(\mathbf{b}_i)$ denotes the outputs of block $\mathbf{b}_i$. Finally, for each node v, we let $out^{-1}(v) = \{\mathbf{b}_j | v \in out(\mathbf{b}_j)\}$ denote the set of blocks in which v is an output.

In the sequential circuit in Figure 4.1(a), the blocks are shown in dotted boxes. We have $out^{-1}(g_3) = \{\mathbf{b}_4\}$, $out^{-1}(s_1) = \{\mathbf{b}_5\}$, $out^{-1}(g_1) = \{\mathbf{b}_3\}$ and $out^{-1}(r) = \emptyset$. Note that y_1 and y_2 are primary output labels for g_3 and g_2, respectively, and do not represent separate nodes. Figure 4.1(b) presents the corresponding directed graph, including the artificial sink r.

We refer the reader to Example 3.2 and Figure 3.5, which give a SAT encoding for debugging

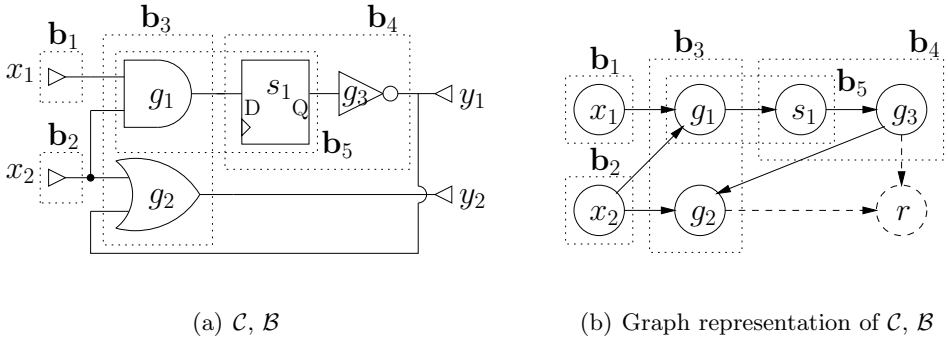
the circuit in Figure 4.1(a).

4.2.1 Single-Vertex Dominators

In a directed graph $\mathcal{C} = (V, E, r)$ with a single output sink $r \in V$, a node $u \in V$ is said to be a structural single-vertex post-dominator, or simply *dominator*, of a node $v \in V$, if every path from v to the sink r passes through u. The set $dom(v) = \{u \in V | u \text{ dominates } v\}$ consists of nodes that dominate v. As a convention, we consider that a node dominates itself. Furthermore, to ease the presentation, we assume that every node has a path to r (*i.e.*, all dangling logic has been removed).

The *immediate dominator* of a node v ($v \neq r$), denoted by $idom(v)$, is a provably unique node u ($u \neq v$) that dominates v and is dominated by all the nodes in $dom(v) - \{v\}$. It can be shown that for all $v \in V - \{r\}$, $dom(v) = \{v\} \cup idom(v) \cup idom(idom(v)) \cup \cdots \cup \{r\}$ [31]. Therefore it is sufficient to compute all immediate dominators, which can be done in $O(|E|+|V|)$ time [9, 45]. A nearly linear algorithm given in [71] for computing immediate dominators is often used in practice. In the directed graph shown in Figure 4.1(b), $dom(x_1) = \{x_1, g_1, s_1, g_3, r\}$, $dom(x_2) = \{x_2, r\}$, $idom(x_1) = \{g_1\}$, $idom(x_2) = \{r\}$.

In this chapter, we are interested in finding dominance relationships between blocks in $\mathcal{B}$, rather than between nodes in V. Section 4.3 outlines our approach, and discusses why methods for computing single-vertex dominators, as well as existing techniques for computing multiple-vertex dominators are not applicable in a design debugging setting.



(a) $\mathcal{C}$, $\mathcal{B}$ (b) Graph representation of $\mathcal{C}$, $\mathcal{B}$

Figure 4.1: A sequential circuit with blocks

4.3 Dominance between Blocks

In this section, an iterative algorithm is presented for finding all dominance relationships among a fixed set of multiple-vertex blocks, which are naturally defined in a hierarchical RTL design.

Definition 4.1 *A block* $\mathbf{b}_j$ *dominates another block* $\mathbf{b}_i$*, denoted as* $\mathbf{b}_j \mathsf{D} \mathbf{b}_i$*, if and only if every path from every node in* $\mathbf{b}_i$ *to a primary output in* y *passes through a node in* $\mathbf{b}_j$*.*

Assuming that internal (non-output) block nodes cannot be primary outputs, any path to a primary output exiting a block must pass through one of its outputs. Furthermore all primary outputs are connected to the artificial sink r. As such, the block dominator relation $\mathsf{D} \subseteq \mathcal{B} \times \mathcal{B}$ can be formalized using *restricted quantifier* notation [14] as follows:

$$\mathbf{b}_j \mathsf{D} \mathbf{b}_i \Leftrightarrow \forall v[v \in out(\mathbf{b}_i)].\forall p[v \overset{p}{\rightsquigarrow} r].\exists u[u \in p].(u \in out(\mathbf{b}_j)) \tag{4.1}$$

where a path $p : v \overset{p}{\rightsquigarrow} r$ is a sequence of nodes starting at v and ending at r. The right-hand-side of Equation 4.1 reads *"for all vertices* v *in* $out(\mathbf{b}_i)$*, and for all paths* p *from* v *to* r*, there exists a vertex* u *in* p*, such that* $u \in out(\mathbf{b}_j)$*"*.

We let the set $\mathsf{D}(\mathbf{b}_i) = \{\mathbf{b}_j | \mathbf{b}_j \mathsf{D} \mathbf{b}_i\}$ consist of blocks that dominate $\mathbf{b}_i$. Note that each block dominates itself (*i.e.,* $\mathbf{b}_i \mathsf{D} \mathbf{b}_i$) according to (4.1). Consider the sequential circuit given in Figure 4.1(a). Although x_2 is not dominated by g_1 or g_2 separately, block $\mathbf{b}_2 = \{x_2\}$ is dominated by block $\mathbf{b}_3 = \{g_1, g_2\}$.

The relation D on the blocks $\mathcal{B}$ of $\mathcal{C}$ in Figure 4.1(b) is illustrated in Figure 4.2. Unlike single-vertex dominators, a block does not necessarily have a unique immediate dominator

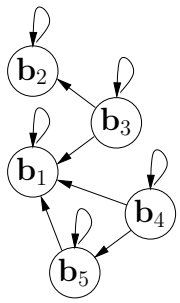


Figure 4.2: Block dominator relation D of $\mathcal{C}$

block. This is because given blocks b_i, b_j, b_m:

$$b_i \mathsf{D} b_m \wedge b_j \mathsf{D} b_m \not\Rightarrow b_i \mathsf{D} b_j \vee b_j \mathsf{D} b_i.$$

This can be seen for blocks $\mathbf{b}_3$, $\mathbf{b}_4$ and $\mathbf{b}_1$ in Figure 4.2. Here both $\mathbf{b}_3$ and $\mathbf{b}_4$ dominate $\mathbf{b}_1$, however there is no dominance relationship between the two. As such, algorithms for calculating single-vertex immediate dominators cannot be used for computing block dominators. On the other hand, in existing approaches for computing so-called *generalized* or multiple-vertex dominators [8, 54, 65], block boundaries are not defined in advance. Instead, nodes are assembled into multiple-vertex dominators on-the-fly according to certain conventions, *e.g.*, the smallest subset of $fanout(v)$ collectively dominating a node v [8, 54]. This is not applicable in a design debugging setting, where circuit blocks are defined in advance by the hierarchical RTL design.

In this work, the block dominator relation D on the set of blocks $\mathcal{B}$ is computed in two steps. First, the block dominators of each node $v \in V$ are computed. Then, these block-to-node dominators are used to compute the block-to-block dominator relation D.

Definition 4.2 *A block $\mathbf{b}_j$ dominates a node v, denoted as $\mathbf{b}_j \mathsf{d} v$, if and only if every path from v to a primary output in y passes through a node in $\mathbf{b}_j$.*

The block-to-node dominator relation $\mathsf{d} \subseteq \mathcal{B} \times V$ can be formalized as :

$$\mathbf{b}_j \mathsf{d} v \Leftrightarrow \forall p[v \overset{p}{\rightsquigarrow} r].\exists u[u \in p].(u \in out(\mathbf{b}_j)) \tag{4.2}$$

We let the set $\mathsf{d}(v) = \{\mathbf{b}_j | \mathbf{b}_j \mathsf{d} v\}$ consist of blocks that dominate node v. For instance, in Figure 4.1(b), $\mathsf{d}(x_1) = \{\mathbf{b}_1, \mathbf{b}_3, \mathbf{b}_4, \mathbf{b}_5\}$, $\mathsf{d}(x_2) = \{\mathbf{b}_2, \mathbf{b}_3\}$ and $\mathsf{d}(s_1) = \{\mathbf{b}_4, \mathbf{b}_5\}$.

Algorithm 4.1 shows our pseudocode for computing the block dominator relation D. It first computes the sets $\mathsf{d}(v)$ for every $v \in V$ (lines 1 to 17). This is done using an iterative algorithm, where the set of block dominators of each node is initialized to all blocks $\mathcal{B}$ and iteratively refined until it converges to its actual block dominators. These block-to-node dominators are subsequently used on line 19 to compute $\mathsf{D}(\mathbf{b}_i)$ for every $\mathbf{b}_i \in \mathcal{B}$.

On line 1, $\mathcal{C}^T$ denotes the transpose of directed graph $\mathcal{C}$ (*i.e.,* $\mathcal{C}$ with edges reversed). The function REVERSEPOSTORDERING($\mathcal{C}^T, r$) performs a depth-first search (DFS) of $\mathcal{C}^T$ starting

Algorithm 4.1: Compute Block Dominators

input : Directed graph $\mathcal{C} = (V, E, r)$, blocks $\mathcal{B}$

output: Block dominator relation D

$V \leftarrow$ reversePostordering$(\mathcal{C}^T, r)$;

// For each node v, compute $out^{-1}[v]$: the set of blocks in which v is an output

foreach $v \in V$ **do** $out^{-1}[v] \leftarrow \emptyset$;

foreach $\mathbf{b}_i \in \mathbf{b}$ **do**

  foreach $v \in out[\mathbf{b}_i]$ **do** $out^{-1}[v] \leftarrow out^{-1}[v] \cup \mathbf{b}_i$;

// Compute block-to-node dominator relation d

$\mathsf{d}[r] \leftarrow \emptyset$;

foreach $v \in V - \{r\}$ **do** $\mathsf{d}[v] \leftarrow \mathcal{B}$;

$changed \leftarrow true$;

while *changed* **do**

  $changed \leftarrow false$;

  foreach $u \in V$ *in reverse postorder* **do**

    $blocks \leftarrow \bigcap_{\forall v \in fanout[u]} \left(\mathsf{d}[v] \cup out^{-1}[v]\right)$;

    if $blocks \neq \mathsf{d}[u]$ **then**

      $\mathsf{d}[u] \leftarrow blocks$;

      $changed \leftarrow true$;

foreach $v \in V$ **do** $\mathsf{d}[v] \leftarrow \mathsf{d}[v] \cup out^{-1}[v]$;

// Compute block dominator relation D

foreach $\mathbf{b}_i \in \mathcal{B}$ **do** $\mathsf{D}[\mathbf{b}_i] \leftarrow \bigcap_{\forall v \in out[\mathbf{b}_i]} \mathsf{d}[v]$;

from r, and sorts the nodes in *decreasing finishing times*. In general, a reverse postordering is not unique. For instance, for $\mathcal{C}$ given in Figure 4.1(b), reversePostordering$(\mathcal{C}^T, r)$ can return $\langle r, g_2, g_3, s_1, g_1, x_2, x_1 \rangle$. Traversing V in reverse postorder guarantees for each node $u \in V$ that at least one of $v \in fanout(u)$ is already visited by the time u is traversed. This will reduce the number of iterations needed for convergence when computing the sets $\mathsf{d}(v)$ later in the algorithm.

Lines 3 to 5 calculate the sets $out^{-1}(v)$ for each node v. The algorithm for computing the sets $\mathsf{d}(v)$ for all nodes v (lines 7 to 16) is based on the traditional data-flow analysis algorithm for finding single-vertex dominators [7, 31]. Lines 7 and 8 initialize each dominator set $\mathsf{d}(v)$ to all blocks $\mathcal{B}$ for $v \in V - \{r\}$, and to the empty set for $v = r$. In each iteration of the **while** loop, the nodes are traversed in reverse postorder (as calculated on line 1) and a refined set of dominator blocks is computed for each node on line 13. The computation of this refined set of dominator blocks of each node on line 13 is the main difference with the data-flow analysis algorithm for single-vertex dominators. The new set of dominator blocks of a node $u \in V$ is updated to be the intersection, over all $v \in fanout(u)$, of the dominator blocks of v as well as the blocks in which v is an output. If any of the sets $\mathsf{d}(v)$ are changed during an iteration (*i.e.*, the **if** condition on line 14 is true), the **while** loop is executed again. The **while** loop terminates after an iteration where all block-to-node dominator sets remain unchanged. Line 17 adds the blocks in which node v is an output, to the dominators of v. Finally, on line 19, the block dominators $\mathsf{D}(\mathbf{b}_i)$ of each block $\mathbf{b}_i$ are computed by intersecting the block dominators of each node in $out(\mathbf{b}_i)$.

Lemma 4.1 *The* **while** *loop in Algorithm 4.1 terminates and the block-to-node dominator relation* d *is correctly computed by the end of the* **foreach** *loop on line 17.*

Proof:

In [60], the authors describe a class of iterative *data-flow analysis* algorithms, which have a variety of applications (*e.g.,* in compiler optimization [5]) and are not restricted to calculating dominators. Their class of algorithms is presented using a general lattice theoretic framework. The authors in [60] analyze the computation and the conditions for the termination of this class

of data-flow analysis algorithms. We will use the conclusions of [60] to analyze the computation of the block-to-node dominator relation d in Algorithm 4.1.

In order to present the relevant results of [60], we need to introduce some concepts from lattice algebra. A *meet-semilattice* is an algebraic structure $\langle L, \wedge \rangle$ consisting of a set L with a binary operation $\wedge$, called *meet*, such that for all $x, y, z \in L$, the following identities hold:

- Idempotency: $x \wedge x = x$
- Commutativity: $x \wedge y = y \wedge x$
- Associativity: $x \wedge (y \wedge z) = (x \wedge y) \wedge z$

A partial order is induced on a meet-semilattice by setting $x \leq y$ if and only if $x \wedge y = x$. L is said to have a *zero* element $\mathbf{0}$ if for all $x \in L$, $\mathbf{0} \wedge x = \mathbf{0}$. L is said have a *one* element $\mathbf{1}$, if for all $x \in L$, $\mathbf{1} \wedge x = x$.

Algorithm 4.2 is algorithm "MK" from [60]. The input of this algorithm is a circuit $\mathcal{C}$ and a meet-semilattice $\langle L, \wedge \rangle$ with a zero element $\mathbf{0}$ and a one element $\mathbf{1}$. Algorithm 4.2 computes a mapping $A : V \to L$, which maps each $v \in V$ to an element in L. The mapping A does not have to be one-to-one or onto. Note that the meaning of $A(v)$ is not defined, since Algorithm 4.2 is just an algorithm template. Furthermore, for each $v \in V$, f_v on line 7 of Algorithm 4.2 is a general function of the form $f_v : L \to L$, that must satisfy certain conditions given below.

Let F refer to a *set of functions* of form $f : L \to L$. F is said to be an admissible set of functions for L if and only if the following four conditions are satisfied:

(a) All functions $f \in F$ are distributive over $\wedge$:

$\forall x, y[x, y \in L].\forall f[f \in F].(f(x \wedge y) = f(x) \wedge f(y))$

(b) F has an identity function e:

$\exists e[e \in F].\forall x[x \in L].(e(x) = x)$

(c) F is closed under composition:

$\forall f, g[f, g \in F].(f \circ g \in F)$

(d) For each $x \in L$, there exists a subset of functions $H \subseteq F$, such that $x = \bigwedge_{f \in H} f(\mathbf{0})$:

$\forall x[x \in L].\exists H[H \subseteq F].\left(x = \bigwedge_{f \in H} f(\mathbf{0})\right)$

Algorithm 4.2: Algorithm MK in [60]

input : Directed graph $\mathcal{C} = (V, E, r)$, meet-semilattice $\langle L, \wedge \rangle$

output: Map $A : V \to L$

```
1  A[r] ← 0;
2  foreach v ∈ V − {r} do A[v] ← 1;
3  changed ← true;
4  while changed do
5      changed ← false;
6      foreach u ∈ V in reverse postorder do
7          temp ← ⋀_{∀v∈fanout(u)} f_v(A[v]);
8          if temp ≠ A[u] then
9              A[u] ← temp;
10             changed ← true;
```

The authors of [60] show that if there exists a set of admissible functions F, such that $\forall v \in V, f_v \in F$, then Algorithm 4.2 terminates. Note that this mapping from each node v to a function $f_v \in F$ does not have to be one-to-one. *I.e.,* given two nodes $u, v \in V$, it is possible that they map to the same function $f_u = f_v \in F$. Also, the mapping does not have to be onto. *I.e.,* there may exist functions in F that no node maps to, or $\{f_v | \forall v \in V\} \subseteq F$.

Furthermore, they show that in such a scenario, at the completion of Algorithm 4.2, for each $v \in V$, we have:

$$A(v) = \bigwedge_{\substack{\forall \text{ paths } p=\langle v, v_1, v_2, \ldots, v_n, r\rangle \\ \text{from } v \text{ to } r}} f_{v_1}(f_{v_2}(\cdots f_{v_n}(f_r(\mathbf{0}))\cdots)) \tag{4.3}$$

In what follows, we will show that lines 7 to 16 of Algorithm 4.1 constitute a special case of Algorithm 4.2 for a specific meet-semilattice $\langle L, \wedge\rangle$, a specific set of functions $\{f_v | \forall v \in V\}$, and replacing each $A(v)$ by $\mathsf{d}(v)$, which at the end of line 17 will correctly store the block dominators of node v.

Let us consider the power set of $\mathcal{B}$, denoted as $\mathcal{P}(\mathcal{B})$, which is the set of all subsets of $\mathcal{B}$. We will use capital bold letters, such as $\mathbf{B}$, to refer to elements of $\mathcal{P}(\mathcal{B})$. *I.e.,* $\mathbf{B} \in \mathcal{P}(\mathcal{B})$ is some subset of the blocks in $\mathcal{B}$, or $\mathbf{B} \subseteq \mathcal{B}$. It is easy to show that $\langle \mathcal{P}(\mathcal{B}), \cap\rangle$ is a meet-semilattice, where $\cap$ (set intersection) is our meet-operator. In fact, $\forall \mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathcal{P}(\mathcal{B})$:

- $\mathbf{B} \cap \mathbf{B} = \mathbf{B}$. Hence, $\cap$ is idempotent.
- $\mathbf{A} \cap \mathbf{B} = \mathbf{B} \cap \mathbf{A}$. Hence, $\cap$ is commutative.
- $\mathbf{A} \cap (\mathbf{B} \cap \mathbf{C}) = (\mathbf{A} \cap \mathbf{B}) \cap \mathbf{C}$. Hence, $\cap$ is associative.

Furthermore, the elements of $\mathcal{P}(\mathcal{B})$ admit a partial order $\subseteq$ such that $\mathbf{A} \subseteq \mathbf{B}$ if and only if $\mathbf{A} \cap \mathbf{B} = \mathbf{A}$. Also, $\mathcal{P}(\mathcal{B})$ has a *zero* element, which is the empty set $\emptyset$, since $\forall \mathbf{B} \in \mathcal{P}(\mathcal{B})$, $\emptyset \cap \mathbf{B} = \emptyset$. And $\mathcal{P}(\mathcal{B})$ has a *one* element, which is the set of all blocks $\mathcal{B}$, since $\forall \mathbf{B} \in \mathcal{P}(\mathcal{B})$, $\mathcal{B} \cap \mathbf{B} = \mathbf{B}$.

Using the meet-semilattice $\langle \mathcal{P}(\mathcal{B}), \cap\rangle$, replacing $\mathbf{0}$ and $\mathbf{1}$ by $\emptyset$ and $\mathcal{B}$, respectively, as well as A by d, lines 1 and 2 of Algorithm 4.2 become the same as lines 7 and 8 of Algorithm 4.1. Also, since $A(v) \in L$, now $\mathsf{d}(v) \in \mathcal{P}(\mathcal{B})$, as expected. Furthermore, after replacing A by d and

$\wedge$ by $\cap$ in the remainder of Algorithm 4.2, the only remaining difference from lines 7 to 16 of Algorithm 4.1 is line 13. So in order to prove that lines 7 to 16 of Algorithm 4.1 are in fact a valid special case of Algorithm 4.2 (this would make it possible to prove the termination and correctness of our own algorithm), it only remains to show that the specific functions of the form $f_v(\mathbf{B}) = \mathbf{B} \cup out^{-1}(v)$ on line 13 of Algorithm 4.1 satisfy the conditions put forth in [60]. In order to do so, we must show that there exists an admissible set of functions F that the functions $f_v(\mathbf{B}) = \mathbf{B} \cup out^{-1}(v)$, for each $v \in V$, can be drawn from.

Given the meet-semilattice $\langle \mathcal{P}(\mathcal{B}), \cap \rangle$, let F refer to the following set of functions f, mapping sets of blocks to sets of blocks:

$$F = \{f(\mathbf{B}) = \mathbf{B} \cup \mathbf{A} \mid \forall \mathbf{A} \in \mathcal{P}(\mathcal{B})\}. \tag{4.4}$$

Let us first rewrite the four admissability conditions given above using our specific meet-semilattice $\langle \mathcal{P}(\mathcal{B}), \cap \rangle$.

(a) All functions in F are distributive over $\cap$:

$\forall \mathbf{A}, \mathbf{B}[\mathbf{A}, \mathbf{B} \in \mathcal{P}(\mathcal{B})].\forall f[f \in F].(f(\mathbf{A} \cap \mathbf{B}) = f(\mathbf{A}) \cap f(\mathbf{B}))$

(b) F has an identity function:

$\exists e[e \in F].\forall \mathbf{B}[\mathbf{B} \in \mathcal{P}(\mathcal{B})].(e(\mathbf{B}) = \mathbf{B})$

(c) F is closed under composition:

$\forall f, g[f, g \in F].(f \circ g \in F)$

(d) $\forall \mathbf{A}[\mathbf{A} \in \mathcal{P}(\mathcal{B})].\exists H[H \subseteq F].\left(\mathbf{A} = \bigcap_{f \in H} f(\emptyset)\right)$

We prove that F given in (4.4) is admissible:

(a) For any function $f \in F$, such that $f(\mathbf{B}) = \mathbf{B} \cup \mathbf{C}$, where $\mathbf{C} \in \mathcal{P}(\mathcal{B})$, and $\forall \mathbf{A}, \mathbf{B} \in \mathcal{P}(\mathcal{B})$, we have $f(\mathbf{A} \cap \mathbf{B}) = (\mathbf{A} \cap \mathbf{B}) \cup \mathbf{C} = (\mathbf{A} \cup \mathbf{C}) \cap (\mathbf{B} \cup \mathbf{C}) = f(\mathbf{A}) \cap f(\mathbf{B})$. Therefore All functions in F are distributive over $\cap$.

(b) By Equation (4.4), F contains the function $f(\mathbf{B}) = \mathbf{B}$, which is the identity function.

(c) For any two functions in F, $f(\mathbf{B}) = \mathbf{B} \cup \mathbf{A}$ and $g(\mathbf{B}) = \mathbf{B} \cup \mathbf{C}$, we have $(f \circ g)(\mathbf{B}) = f(g(\mathbf{B}) = f(\mathbf{B} \cup \mathbf{C}) = \mathbf{B} \cup (\mathbf{C} \cup \mathbf{A}) \in F$ by the definition of F in Equation (4.4), since $\mathbf{C} \cup \mathbf{A} \in \mathcal{P}(\mathcal{B})$.

(d) For any block $\mathbf{A} \in \mathcal{P}(\mathcal{B})$, let the set of functions H contain one function as follows: $H = \{f(\mathbf{B}) = \mathbf{B} \cup \mathbf{A}\}$. Then $\bigcap_{f \in H} f(\emptyset) = \mathbf{A}$, as required.

Now that we know that F in (4.4) is admissible, we must show that the functions $f_v(\mathbf{B}) = \mathbf{B} \cup out^{-1}(v)$ used on line 13 of Algorithm 4.1 are indeed drawn from this F. Since $\forall v \in V$, $out^{-1}(v) \in \mathcal{P}(\mathcal{B})$, clearly $\{f_v(\mathbf{B}) = \mathbf{B} \cup out^{-1}(v) \,|\, \forall v \in V\} \subseteq F$. Therefore, the **while** loop in Algorithm 4.1 terminates.

Now, using (4.3) in the context of our meet-semilattice $\langle \mathcal{P}(\mathcal{B}), \cap \rangle$, we know that at the completion of this **while** loop, we have:

$$\begin{aligned} \mathsf{d}(v) &= \bigcap_{\substack{\forall \text{ paths } p = \langle v, v_1, v_2, \ldots, v_n, r \rangle \\ \text{such that } v \overset{p}{\rightsquigarrow} r}} f_{v_1}(f_{v_2}(\cdots f_{v_n}(f_r(\emptyset)) \cdots)) \\ &= \bigcap_{\forall p [v \overset{p}{\rightsquigarrow} r]} \left(\bigcup_{\forall u \in p - \{v\}} out^{-1}(u) \right) \end{aligned} \tag{4.5}$$

Finally, on line 17 of Algorithm 4.1, $out^{-1}(v)$ is added to each $\mathsf{d}(v)$. As such, by the end of the **foreach** loop on line 17, we have:

$$\begin{aligned} \mathsf{d}(v) &= \left(\bigcap_{\forall p [v \overset{p}{\rightsquigarrow} r]} \left(\bigcup_{\forall u \in p - \{v\}} out^{-1}(u) \right) \right) \cup out^{-1}(v) \\ &= \bigcap_{\forall p [v \overset{p}{\rightsquigarrow} r]} \left(\bigcup_{\forall u \in p} out^{-1}(u) \right) = \bigcap_{\forall p [v \overset{p}{\rightsquigarrow} r]} \left(\bigcup_{\forall u \in p} \{\mathbf{b}_j | u \in out(\mathbf{b}_j)\} \right) \\ &= \bigcap_{\forall p [v \overset{p}{\rightsquigarrow} r]} \{\mathbf{b}_j | \exists u [u \in p].(u \in out(\mathbf{b}_j))\} \\ &= \{\mathbf{b}_j | \forall p [v \overset{p}{\rightsquigarrow} r].\exists u [u \in p].(u \in out(\mathbf{b}_j))\} \end{aligned}$$

Therefore, the computed sets $\mathsf{d}(v)$ satisfy the definition of the block-to-node dominator relation d given in (4.2).

■

Theorem 4.2 *Algorithm 4.1 correctly computes the block dominator relation* D.

Proof: $\mathsf{D}(\mathbf{b}_i)$ is computed on line 19 as $\bigcap_{\forall v \in out(\mathbf{b}_i)} \mathsf{d}(v)$. Using Lemma 4.1, we get:

$$\begin{aligned}\mathsf{D}(\mathbf{b}_i) &= \bigcap_{\forall v \in out(\mathbf{b}_i)} \{\mathbf{b}_j | \forall p[v \overset{p}{\rightsquigarrow} r].\exists u[u \in p].(u \in out(\mathbf{b}_j))\} \\ &= \{\mathbf{b}_j | \forall v[v \in out(\mathbf{b}_i)].\forall p[v \overset{p}{\rightsquigarrow} r].\exists u[u \in p].(u \in out(\mathbf{b}_j))\}\end{aligned}$$

which satisfies the definition of the block dominator relation D given in (4.1).

■

The overall run-time of Algorithm 4.1 is normally dictated by the run-time of the **while** loop from line 10 to 16. Furthermore, during each iteration of the **while** loop, line 13 clearly dominates computation time. We assume that all dangling logic has been removed during preprocessing (*i.e.,* every node has a path to r), and as such $|V| = O(|E|)$. Using an aggregate analysis of all executions of line 13 during a single iteration of the **while** loop, it can be seen that line 13 performs a total of $O(|E|)$ intersections and unions between two sets of size at most $|\mathcal{B}|$ (since $\mathsf{d}(v), out^{-1}(v) \subseteq \mathcal{B}$). We assume that all sets are implemented using ordered lists and therefore intersections and unions can be done in linear time. As such, in a single iteration of the **while** loop, line 13 takes $O(|\mathcal{B}| \cdot |E|)$ time.

Let c denote the so-called *loop-connectedness* of the directed graph $\mathcal{C} = (V, E, r)$, which refers to the maximum number of *back edges* in any cycle-free path in $\mathcal{C}$. The back edges are defined according to the DFS performed in REVERSEPOSTORDERING$(\mathcal{C}^T, r)$ on line 1.[1] It is proven in [60] that the number of iterations of the **while** loop for the general class of algorithms given by Algorithm 4.2 is bounded by $c + 2$, if and only if the following condition holds:

$$\forall f, g[f, g \in F].\, ((f \circ g)(\mathbf{0}) \geq g(\mathbf{0}) \wedge f(\mathbf{1}))\,, \tag{4.6}$$

where F is a set of admissible functions for meet-semilattice $\langle L, \wedge \rangle$, as described in the proof of Lemma 4.1. Using the set of admissible functions F for meet-semilattice $\langle \mathcal{P}(\mathcal{B}), \cap \rangle$ given in (4.4), which we use for our own Algorithm 4.1, this condition becomes:

$$\forall f, g[f, g \in F].\, ((f \circ g)(\emptyset) \supseteq g(\emptyset) \cap f(\mathcal{B}))\,. \tag{4.7}$$

[1] Back edges in a DFS traversal are the edges that lead to a *discovered but unfinished* node during the traversal (referred to as a *gray* node in [33]). The reader is referred to [33] for a reminder on back edges in DFS.

Consider any two functions $f, g \in F$ such that $f(\mathbf{B}) = \mathbf{B} \cup \mathbf{A}$ and $g(\mathbf{B}) = \mathbf{B} \cup \mathbf{C}$, where $\mathbf{A}, \mathbf{C} \in \mathcal{P}(\mathcal{B})$ are arbitrary sets of blocks. We have:

$$(f \circ g)(\emptyset) = f(g(\emptyset)) = f(\mathbf{C}) = \mathbf{A} \cup \mathbf{C}$$

and

$$g(\emptyset) \cap f(\mathcal{B}) = \mathbf{C} \cap \mathcal{B} = \mathbf{C},$$

clearly satisfying (4.7). Therefore, our algorithm takes $O(c \cdot |\mathcal{B}| \cdot |E|)$ time.

4.4 Leveraging Block Dominance in Design Debugging

In this section, we show how to leverage the relation D to imply solutions early in the design debugging iterations. In effect, given a solution consisting of a set of blocks, we show that we can replace each block by any of its dominator blocks to get another solution. Formally, it is proven that for each known solution of *Debug* (3.2) of the form $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_N}\}$, every set of the form $\{\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_N}\}$ such that $\langle \mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_N} \rangle \in \mathsf{D}(\mathbf{b}_{i_1}) \times \cdots \times \mathsf{D}(\mathbf{b}_{i_N})$ is also a solution of *Debug* (3.2). Furthermore, it is shown that corrections for each implied solution can also be obtained automatically from the satisfying assignment of the original solution.

First, due to the fixed length of a given counter-example, we must define the following, slightly modified concept of domination.

Definition 4.3 *We say that a block* $\mathbf{b}_j$ *dominates another block* $\mathbf{b}_i$ within k cycles*, denoted as* $\mathbf{b}_j \mathsf{D}_k \mathbf{b}_i$*, if and only if every path containing at most* k *state elements, starting from every node in* $out(\mathbf{b}_i)$ *to* r *passes through a node in* $out(\mathbf{b}_j)$*.*

The following three Lemmas are used in the proof of Theorem 4.6. Note that in Lemma 4.3, $\bigcup_{n=1}^{N} \mathbf{b}_{i_n}$ (respectively $\bigcup_{n=1}^{N} \mathbf{b}_{j_n}$) denotes a *super-block* consisting of all nodes in blocks $\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_N}$ (respectively $\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_N}$).

Lemma 4.3 *Given blocks* $\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_N}$ *and* $\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_N}$*,*

$$\bigwedge_{n=1}^{N} (\mathbf{b}_{j_n} \mathsf{D} \mathbf{b}_{i_n}) \Rightarrow \left(\bigcup_{n=1}^{N} \mathbf{b}_{j_n} \right) \mathsf{D} \left(\bigcup_{n=1}^{N} \mathbf{b}_{i_n} \right).$$

In other terms, if $\forall n[1 \leq n \leq N]$, $\mathbf{b}_{j_n}$ *dominates* $\mathbf{b}_{i_n}$, *then the super-block formed by all nodes in* $\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_N}$, *dominates the super-block formed by all nodes in* $\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_N}$.

Proof: If $\forall n[1 \leq n \leq N]$, any path from any node in $out(\mathbf{b}_{i_n})$ to a primary output passes through a node in $out(\mathbf{b}_{j_n})$, then clearly any path from any node in one of $out(\mathbf{b}_{i_1}), \ldots, out(\mathbf{b}_{i_N})$ to a primary output passes through a node in one of $out(\mathbf{b}_{j_1}), \ldots, out(\mathbf{b}_{j_N})$.

■

Lemma 4.4 *Given blocks* $\mathbf{b}_i$ *and* $\mathbf{b}_j$,

$$\mathbf{b}_j \mathsf{D} \mathbf{b}_i \Rightarrow \mathbf{b}_j \mathsf{D}_\mathsf{k} \mathbf{b}_i$$

Proof: If $\mathbf{b}_j \mathsf{D} \mathbf{b}_i$ then every path from $\mathbf{b}_i$ to a primary output passes through $\mathbf{b}_j$. In particular, all paths to a primary output with at most k state elements also pass through $\mathbf{b}_j$.

■

Lemma 4.5 *If* $\mathbf{b}_j \mathsf{D}_\mathsf{k} \mathbf{b}_i$ *in* $\mathcal{C}$, *then in the k-cycle time-frame expansion of* $\mathcal{C}$, *every path from every node in* $out(\mathbf{b}_i^t)$ *(*$\forall t[1 \leq t \leq k]$*) to any primary output in* $\{y^t, \ldots, y^k\}$ *passes through a node in* $out(\mathbf{b}_j^{t'})$ *(for some* $t'[t \leq t' \leq k]$*).*

Proof: True by construction.

■

Recall from Subsection 3.2.1 that modern automated debuggers start by solving *Debug* (3.2) for $N = 1$, then $N = 2$, and so on, until a user-specified maximum error cardinality $N = N_{max}$. An RTL debug tool either stops after finding all solutions for the smallest N for which *Debug* is `SAT`, or alternatively returns all solutions of every error cardinality up to $N = N_{max}$. In both cases, given a solution of cardinality N, either no solutions of smaller cardinalities exist or all such solutions have already been blocked in *Debug* using blocking clauses, which is why the following theorem only considers implied solutions of the same cardinality N as the original solution.

Theorem 4.6 *Given an erroneous design $\mathcal{C}$, a counter-example of length k along with the corresponding expected outputs and an error cardinality N, if $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_N}\}$ is a solution of Debug (3.2) and $\bigwedge_{n=1}^{N}(\mathbf{b}_{j_n} \mathsf{D} \mathbf{b}_{i_n})$, then $\{\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_N}\}$ is an implied solution of Debug (3.2).*

Proof: The theorem can be formalized as:

$$Debug \wedge \bigwedge_{n=1}^{N} e_{i_n} \text{ is } \texttt{SAT} \;\Rightarrow Debug \wedge \bigwedge_{n=1}^{N} e_{j_n} \text{ is } \texttt{SAT} \tag{4.8}$$

where we refer to the left-hand-side (respectively, right-hand-side) formula of the implication as the LHS (respectively, RHS).

Recall that $\mathcal{U}$ in (3.1) refers to the k-time-frame enhanced ILA obtained from $\mathcal{C}$ as described in Subsection 3.2.1. Let $I = \{\mathbf{b}_{i_n}^t | 1 \leq n \leq N, 1 \leq t \leq k\}$ (respectively $J = \{\mathbf{b}_{j_n}^t | 1 \leq n \leq N, 1 \leq t \leq k\}$) denote the union of all nodes in blocks $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_N}\}$ (respectively $\{\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_N}\}$) across all time-frames in $\mathcal{U}$. Also, let $out(I)$ (respectively $out(J)$) refer to the set of outputs of I (respectively J). We will partition the nodes in $\mathcal{U}$ into three parts, $\mathcal{U}_I$, $\mathcal{U}_J$ and $\mathcal{U}_R$, as follows.

Let $\mathcal{U}_J$ denote the transitive fanout of $out(J)$ in $\mathcal{U}$. Let $\mathcal{U}_I$ denote the nodes in $\mathcal{U}$ that are in the transitive fanout of $out(I)$, but not in $\mathcal{U}_J$. Finally, let $\mathcal{U}_R$ consist of the remaining nodes in $\mathcal{U}$, outside $\mathcal{U}_I$ and $\mathcal{U}_J$. We know that $\bigwedge_{n=1}^{N}(\mathbf{b}_{j_n} \mathsf{D} \mathbf{b}_{i_n})$, and by Lemma 4.3 and Lemma 4.4, we get $\left(\bigcup_{n=1}^{N} \mathbf{b}_{j_n}\right) \mathsf{D}_\mathsf{k} \left(\bigcup_{n=1}^{N} \mathbf{b}_{i_n}\right)$. Given this and Lemma 4.5, we can imply that any path from $out(I)$ to a primary output must pass through $out(J)$. As a result, these partitions of $\mathcal{U}$ can be represented by the diagram shown in Figure 4.3.

Note that in Figure 4.3, the output constraints are separated into two subsets: $\Phi_Y = \Phi_Y^J \wedge \Phi_Y^R$, where Φ_Y^J (respectively Φ_Y^R) denotes the output constraints applied at the outputs of $\mathcal{U}_J$ (respectively $\mathcal{U}_R$). This separation is only needed for this proof and is not required by our method.

We know that given $e_{i_1} = \texttt{1}, \ldots, e_{i_N} = \texttt{1}$, there exist assignments to the nodes in $\mathcal{U}_I$, $\mathcal{U}_J$ and $\mathcal{U}_R$ satisfying the LHS. Let $\pi(\mathcal{U}_I)$, $\pi(\mathcal{U}_J)$ and $\pi(\mathcal{U}_R)$ refer to these assignments. We want to find assignments $\pi'(\mathcal{U}_I)$, $\pi'(\mathcal{U}_J)$ and $\pi'(\mathcal{U}_R)$, such that given $e_{j_1} = \texttt{1}, \ldots, e_{j_N} = \texttt{1}$, the RHS is satisfied. These assignments are found as follows.

First consider the subset of output constraints applied at the outputs of $\mathcal{U}_R$, denoted by Φ_Y^R in Figure 4.3. Since $\pi(\mathcal{U}_R)$ satisfies Φ_Y^R and the input constraints to $\mathcal{U}_R$ (*i.e.*, $\Phi_S \wedge \Phi_X$) are the

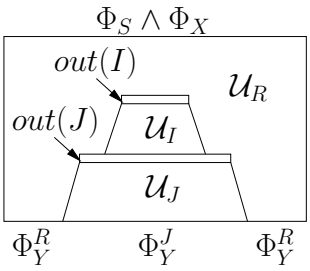


Figure 4.3: Partition of $\mathcal{U}$

same in the LHS and the RHS, setting $\pi'(\mathcal{U}_R) = \pi(\mathcal{U}_R)$ will also satisfy Φ_Y^R in the RHS.

Next, consider $\mathcal{U}_I$. Note that any path from $out(I)$ to a primary output must pass through $out(J)$. Also, setting $e_{j_1} = \mathtt{1}, \ldots, e_{j_N} = \mathtt{1}$ in the RHS disconnects $out(J)$ from their fanins. Therefore, there are no output constraints applied on $\mathcal{U}_I$ (*i.e.*, $\mathcal{U}_I$ is dangling logic in the RHS). As such, $\pi'(\mathcal{U}_I)$ can simply "propagate" the values of $\pi'(\mathcal{U}_R)$ in $\mathcal{U}_I$.

Finally, since the nodes in $out(J)$ are disconnected from their fanins in the RHS, the SAT solver is free to pick any assignment for these variables. Furthermore, setting $\pi'(\mathcal{U}_R) = \pi(\mathcal{U}_R)$ already assigned any inputs to $\mathcal{U}_J$ coming from $\mathcal{U}_R$ to the same values as the LHS. Therefore, we can simply pick $\pi'(\mathcal{U}_J) = \pi(\mathcal{U}_J)$, which will satisfy Φ_Y^J in Figure 4.3. This completes the satisfying assignment π' to all the variables in $\mathcal{U}_I$, $\mathcal{U}_J$ and $\mathcal{U}_R$ in the RHS. Therefore, the RHS is `SAT`.

■

Corollary 4.7 *Given a solution* $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_N}\}$ *and its corresponding satisfying assignment* π *of Debug (3.2), a sequence of corrections for each implied solution* $\{\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_N}\}$ *consists of the assignments to* $\{out(\mathbf{b}_{j_n}^t) | 1 \leq n \leq N, 1 \leq t \leq k\}$ *in* π.

Proof: In the proof of Theorem 4.6, we showed how to build a satisfying assignment π' of the RHS of (4.8) given a satisfying assignment π of the LHS. In particular, we showed that the subset of π' corresponding to $\mathcal{U}_J$ is the same as the subset of π corresponding to $\mathcal{U}_J$. In other terms, $\pi'(\mathcal{U}_J) = \pi(\mathcal{U}_J)$. Since $\mathcal{U}_J$ is simply the transitive fanout of $out(J)$ in $\mathcal{U}$, the subset of π' corresponding to $out(J)$ is also the same as the subset of π corresponding to $out(J)$. As such, given a satisfying assignment π for the original solution $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_N}\}$, a sequence of

corrections for the implied solution $\{\mathbf{b}_{j_1}, \ldots, \mathbf{b}_{j_N}\}$ simply consists of the assignments in π to $out(J) = \{out(\mathbf{b}_{j_n}^t) | 1 \leq n \leq N, 1 \leq t \leq k\}$.

■

4.4.1 Overall Flow

The flowchart in Figure 4.4 illustrates the overall design debugging flow using on-the-fly dominator implications. Algorithm 4.1 is first run to compute $\mathsf{D}(\mathbf{b}_i)$ for every block $\mathbf{b}_i \in \mathcal{B}$. Next, the automated debugger builds the original debugging problem, *Debug* (3.2), and passes it to the SAT solver. If it is `UNSAT`, the flow terminates. Otherwise, a solution $\{\mathbf{b}_{i_1}, \ldots, \mathbf{b}_{i_N}\}$ is returned. A simple implication engine takes in this solution, and using the pre-computed block dominator relation D, generates all newly implied solutions. A blocking clause is added to *Debug* for each of these implied solutions, as well as the original solution. The resulting debugging instance is given again to the automated debugger, and this process is repeated until the problem becomes `UNSAT`.

Example 4.1 *Consider the sequential circuit in Figure 4.1(a) and the corresponding design debugging formulation given in Example 3.2 and illustrated in Figure 3.5. Assume that* $\mathsf{D} \subseteq \mathcal{B} \times \mathcal{B}$ *has been computed using Algorithm 4.1. Furthermore, assume that* $N = 1$*, and that the solver first returns the solution* $\{\mathbf{b}_1\}$*. Since* $\mathsf{D}(\mathbf{b}_1) = \{\mathbf{b}_1, \mathbf{b}_3, \mathbf{b}_4, \mathbf{b}_5\}$*, the solutions* $\{\mathbf{b}_3\}$*,* $\{\mathbf{b}_4\}$ *and* $\{\mathbf{b}_5\}$ *(along with their corrections) can be immediately implied, eliminating three SAT iterations for finding these solutions. After adding the corresponding blocking clauses* $(\neg e_1)$*,* $(\neg e_3)$*,* $(\neg e_4)$ *and* $(\neg e_5)$ *to Debug, the solver returns* `UNSAT`*, indicating that all solutions have been found.*

4.5 Experimental Results

This section presents the experimental results for the proposed dominator-based design debugging flow. All experiments are run using a single core of a Core 2 Quad 2.66 Ghz workstation with 8 GB of RAM and a timeout of 3600 seconds. The proposed debugging framework is implemented using a state-of-the-art hierarchical SAT-based debugger based on [6, 110], with

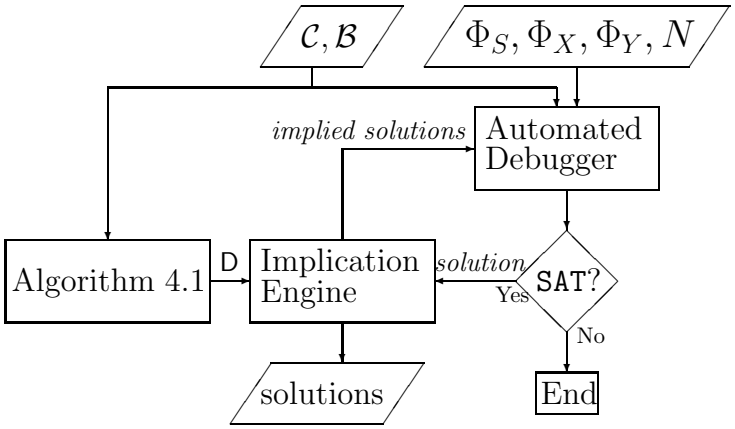


Figure 4.4: Debugging using dominance flow chart

a Verilog front-end to allow for RTL diagnosis. MiniSat-v2.2 [38] is used to solve all SAT instances.

Seven industrial Verilog RTL designs from OpenCores [91] and three commercial designs provided by our industrial partners are used in our experiments. For each design, several debugging instances are generated by inserting different errors into the design. The RTL errors that are injected are based on the experience of our industrial partners. These are common designer mistakes such as wrong state transitions, incorrect operators or incorrect module instantiations. The erroneous design is then run through an industrial simulator with the accompanying testbench, where a failure is detected and a counter-example is recorded. Each block $\mathbf{b}_i \in \mathcal{B}$ consists of the synthesized gates corresponding to a (set of) line(s) in the RTL implementing an assignment, an if statement, a module definition, an instantiation, etc. Experiments are conducted with and without dominator implications. **dbg-trad** refers to the "traditional" debugging flow (without an implication engine), and **dbg-dom** refers to our extended debugging flow using dominator implications, illustrated in Figure 4.4.

Tables 4.1 and 4.2 show the circuit characteristics of our design debugging instances using OpenCore and commercial designs, respectively. In both tables, the first column gives the instance name, which consists of the design name and an appended number indicating a different inserted error. The names of the commercial designs have been replaced by "design1", etc. The following four columns respectively show the number of nodes $|\mathbf{n}|$, the number of blocks $|\mathcal{B}|$, the number of clock-cycles k in the counter-example, and finally the error cardinality N.

Tables 4.3 and 4.4 show the results of all our experiments on the OpenCore and commercial designs, respectively. In both tables, the first column gives the instance name. Columns *o-h* and # *sols* respectively refer to the run-time overhead for setting up the problem (*i.e.*, generating the CNF of *Debug*) and the total number of returned solutions. The *o-h* run-time includes graph optimizations such as dangling logic removal. The *o-h* and the # *sols* are common for both **dbg-trad** and **dbg-dom**. Note that the number of solutions for instances with $N = 2$ can be greater than $\mathcal{B}$ (*e.g.,* usb_funct-3) because each two-block combination $\{\mathbf{b}_{i_1}, \mathbf{b}_{i_2}\}$ that can be modified to correct the counter-example is a solution.

Column four (*dbg*) shows the total SAT solver run-time using **dbg-trad** for finding all

debugging solutions. The remaining columns present the results of our proposed framework, **dbg-dom**. Column *avg* $|\mathsf{D}|$ shows the average size of the sets $\mathsf{D}(\mathbf{b}_i)$ computed by Algorithm 4.1. Next, columns # *impl* and *% impl* respectively show the number of implied solutions for each instance and the percentage of implied solutions among all solutions. Column *dom* shows the run-time of Algorithm 4.1 for computing the block dominator relation D. Column *dbg* gives the total SAT solver run-time using **dbg-dom**, while column *d+d* adds to this the dominator computation run-time of Algorithm 4.1. Finally, column *impr* shows the speed-up achieved by *d+d* over **dbg-trad**, first excluding then including the common overhead. In both tables 4.3 and 4.4, the last row gives the geometric mean of the two improvement measures.

Figure 4.5 plots the ratio of implied solutions for each instance, sorted in increasing order. On average, 66% of all solutions are implied. In other terms, the number of calls to the SAT solver is reduced by a factor of 2.9x due to the early discovery of solutions using our approach. For each solution found by the SAT solver, about 2.6 more solutions are implied on average. This number is significantly less than the average number of dominators of each block, which is 19.5, because many implied solutions in later iterations might have already been found (or implied) in previous iterations. Figure 4.6 plots the number of found solutions versus run-time for both **dbg-trad** and **dbg-dom** for design1-2. It can be seen that while **dbg-trad** returns

Figure 4.5: Ratios of implied solutions to all solutions

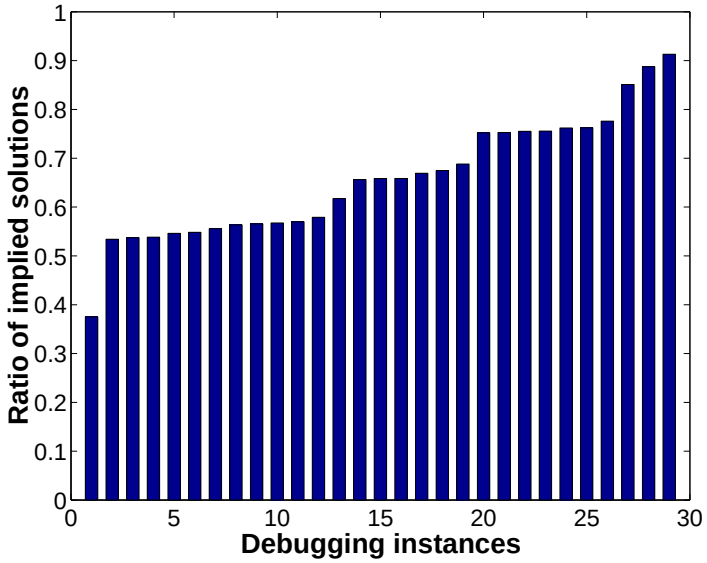
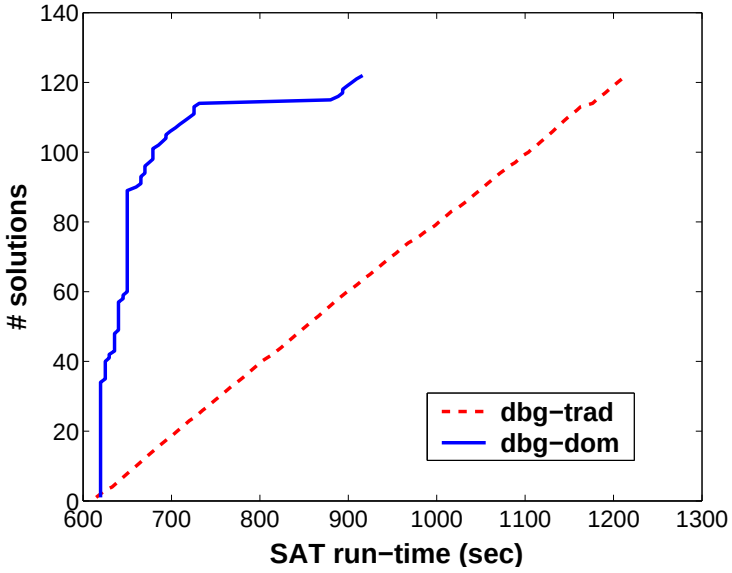


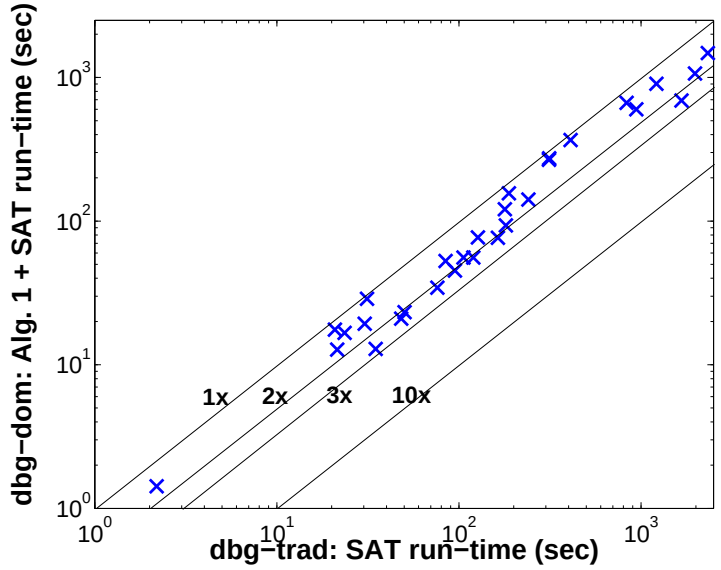
Figure 4.6: # solutions vs. run-time for design1-2



solutions at roughly equal time intervals, **dbg-dom** initially discovers solutions at a fast rate due to new implications, but the rate of discovery of new solutions decreases with time. Returning most solutions early is beneficial because the designer can start examining returned solutions earlier, while the debugger continues to run.

The average speed-up in total SAT run-time from **dbg-trad** to **dbg-dom** is 1.8x. In many

Figure 4.7: **dbg-dom** vs. **dbg-trad** run-time comparison



cases, higher percentages of implied solutions mean less debugging iterations, which result in less total SAT solving time. For instance, in vga-1, 21 out of 23 solutions (91%) are implied, yielding a 2.4x speed-up in total SAT run-time, compared to the averages of 66% implied solutions and a 1.8x speed-up. However, this is not always true because of the unpredictable behavior of SAT solvers. Furthermore, we have not found any clear relationships between design parameters and improvements due to solution implications. Including the time to compute the dominator relation D, the geometric mean of the speed-up from **dbg-trad** to **dbg-dom** is about 1.64x disregarding common overhead, and 1.4x including common overhead. Figure 4.7 plots the run-times of our approach (d+d) versus those of **dbg-trad** on a logarithmic scale, along with the 1x, 2x, 3x and 10x lines, clearly showing the consistent superiority of the proposed method.

4.6 Summary

This chapter first presents an iterative data-flow analysis algorithm for computing dominance relationships between multiple-output blocks of an RTL design. Block dominators are then leveraged for the early detection of potential bugs, called implied solutions, in automated design debugging, thus reducing the number of formal engine calls. Correction waveforms for these implied solutions are automatically generated without explicit formal analysis. Finally, an extensive set of experiments on real industrial designs demonstrates the consistent benefits of the presented framework.

Table 4.1: Instance information (OpenCores)

Instance	$\lvert\mathbf{n}\rvert$	$\lvert\mathcal{B}\rvert$	k	N
fdct-1	365 574	4 665	142	1
fdct-2	365 574	4 666	146	1
mips789-1	63 241	2 750	24	1
mips789-2	30 171	876	68	1
mips789-3	30 711	904	153	2
usb_funct-1	35 158	3 397	31	1
usb_funct_2	36 181	3 477	25	1
usb_funct_3	36 181	3 401	36	2
wb_dma-1	301 812	8 460	20	1
wb_dma-2	187 874	6 236	69	1
fpu-1	79 504	1 988	312	1
fpu-2	139 932	2 145	312	1
opensparc_ddr2-1	64 915	2 777	23	1
opensparc_ddr2-2	58 399	2 779	31	1
vga-1	89 402	1 741	11 308	1
vga-2	89 488	1 833	7	1
vga-3	89 488	1 741	508	2

Table 4.2: Instance information (commercial)

Instance	$\lvert\mathbf{n}\rvert$	$\lvert\mathcal{B}\rvert$	k	N
design1-1	242 086	16 736	25	1
design1-2	532 610	51 564	27	1
design1-3	203 718	10 258	151	1
design1-4	203 706	10 246	5	1
design1-5	532 634	51 564	29	1
design1-6	690 766	51 564	27	1
design2-1	875 837	84 975	212	1
design2-2	875 837	84 975	212	1
design2-3	875 837	84 975	212	1
design3-1	499 705	20 211	562	1
design3-2	499 705	20 211	177	1
design3-3	499 705	20 211	252	2

Table 4.3: Debugging with and without dominance (OpenCores)

Instance	Common		trad	dbg-dom						
	o-h (sec)	# sols	dbg (sec)	avg |D|	# impl	% impl	dom (sec)	dbg (sec)	d+d (sec)	impr (x)
fdct-1	52.4	79	95.1	15.2	52	66%	7.8	37.4	45.2	2.1x/1.5x
fdct-2	52.2	93	188.0	15.2	64	69%	7.8	148.2	156.0	1.2x/1.2x
mips789-1	15.7	162	76.1	14.2	100	62%	1.0	33.4	34.4	2.2x/1.8x
mips789-2	18.1	37	30.4	9.0	21	57%	0.8	18.5	19.3	1.6x/1.3x
mips789-3	38.4	45	84.4	10.6	34	76%	0.6	52.2	52.8	1.6x/1.3x
usb_funct-1	11.9	423	163.5	10.6	231	55%	0.1	76.4	76.5	2.1x/2.0x
usb_funct-2	9.6	93	21.4	10.2	51	55%	0.9	11.8	12.7	1.7x/1.4x
usb_funct-3	14.6	3 517	1 667.1	10.6	2 656	76%	0.2	690.4	690.6	2.4x/2.4x
wb_dma-1	71.0	135	105.8	14.6	75	56%	5.0	50.8	55.8	1.9x/1.4x
wb_dma-2	32.0	234	178.6	20.8	125	53%	1.6	119.3	120.9	1.5x/1.4x
fpu-1	168.0	8	20.8	10.4	3	38%	1.6	16.0	17.6	1.2x/1.0x
fpu-2	18.4	80	23.5	7.0	43	54%	0.6	16.1	16.7	1.4x/1.2x
ddr2-1	17.1	76	48.2	8.6	44	58%	0.2	20.7	20.9	2.3x/1.7x
ddr2-2	11.5	99	34.9	8.6	56	57%	0.2	12.7	12.9	2.7x/1.9x
vga-1	72.0	23	50.2	19.3	21	91%	2.3	20.9	23.2	2.2x/1.3x
vga-2	2.9	52	2.2	19.3	28	54%	0.2	1.2	1.4	1.5x/1.2x
vga-3	11.0	1 226	941.8	19.3	1 088	89%	0.2	600.5	600.7	1.6x/1.6x
geomean										**1.8x/1.5x**

Table 4.4: Debugging with and without dominance (commercial)

Instance	Common		trad	dbg-dom						
	o-h (sec)	# sols	dbg (sec)	avg |D|	# impl	% impl	dom (sec)	dbg (sec)	d+d (sec)	impr (x)
design1-1	94.0	93	240.6	17.9	53	57%	5.8	135.4	141.2	1.7x/1.4x
design1-2	188.2	122	1 214.0	32.4	93	76%	34.2	869.5	903.7	1.3x/1.3x
design1-3	36.3	127	119.8	18.8	85	67%	3.3	52.5	55.8	2.1x/1.7x
design1-4	16.5	41	31.2	22.5	27	66%	0.5	28.3	28.8	1.1x/1.1x
design1-5	174.5	58	832.3	32.3	45	78%	31.1	634.7	665.8	1.3x/1.2x
design1-6	219.0	71	1 978.3	15.7	40	56%	19.6	1 046.4	1 066.0	1.9x/1.7x
design2-1	456.3	40	410.0	22.7	27	68%	39.2	327.7	366.9	1.1x/1.1x
design2-2	472.7	42	312.2	22.7	32	76%	39.2	228.1	267.3	1.2x/1.1x
design2-3	454.8	32	313.1	22.8	21	66%	39.2	234.7	273.9	1.1x/1.1x
design3-1	99.6	117	180.9	48.8	88	75%	13.1	80.2	93.3	1.9x/1.5x
design3-2	84.8	89	127.0	48.8	67	75%	13.1	63.9	77.0	1.6x/1.3x
design3-3	83.4	1 120	2 326.5	48.8	953	85%	13.1	1 467.0	1 480.1	1.6x/1.5x
geomean										**1.5x/1.3x**

Chapter 5

Leveraging Dominators in Circuit-based QBFs

5.1 Introduction

State-of-the-art QBF solvers (e.g. [13, 18, 47, 103]) normally operate on QBF problems with propositional formulas given in conjunctive normal form (CNF). The standardization of this format has carried over from SAT to QBF due to the efficient data-structures and solving strategies developed for CNF-based SAT solvers, which are also used in many QBF solvers. However, QBF instances originating from CAD problems usually have a circuit structure which is lost during the conversion into CNF. Recently, a number of papers have shown that this structure can be effectively exploited in QBF solvers in different ways. In particular, circuit observability don't-cares can be used to improve the performance of QBF solvers [40, 53] just as they did for SAT [98, 123]. Several papers [97, 122] analyze the limitations of CNF for QBF and offer alternative representations addressing some of these issues. More recently, QBF solvers that handle problems directly in a circuit-based format have shown promising results using various solving methods [40, 53, 74].

Here, a novel preprocessing framework for exploiting the circuit structure of QBF problems using *structural dominators* is presented [77]. A node in a circuit is said to dominate another node if every path from the second node to a primary output passes through the first.

Dominator-based search-space pruning techniques are used in many CAD for VLSI applications, such as test pattern generation [64] and technology mapping [29], among others. In a circuit-based QBF setting, the aim is to simplify the problem by searching for nodes that dominate their fanin cones and subsequently removing the dominated subcircuits using proven theoretical reduction rules. It should be noted that our contribution is orthogonal to existing CNF-based QBF preprocessors (e.g. [104]).

A rigorous proof is given for the reduction of subcircuits dominated by single outputs in a circuit-based QBF, irrespective of the subcircuit input quantifiers or the structure of the remaining circuit. More precisely, the dominator of a subcircuit is shown to be replaceable by an appropriately computed constant truth value or a quantified input variable, without affecting the truth of the original QBF. We introduce the circuit-based QBF preprocessor `PReDom`: PRe(process) and ReD(uce) Dom(inators). `PReDom` is independent of any particular QBF solver and efficiently automates the process of reducing dominated subcircuits.

Experimental results are shown on circuit state-space diameter computation problems [48] for a distributed mutual exclusion protocol from NuSMV [28]. The run-time overhead of preprocessing these benchmarks using `PReDom` is at most five seconds and the resulting logically equivalent QBF problems can be given to any QBF solver. `PReDom` reduces the number of clauses by 47% on average compared to the original instances and 19% after standard simplifications. Three state-of-the-art QBF solvers solve 27% to 45% of the resulting instances, compared to *none* after standard simplifications. These results encourage further research in new strategies that exploit the circuit structure of QBFs to increase performance.

This chapter is organized as follows. Section 5.2 contains preliminaries. Section 5.3 presents the formal theory for the reduction of single-output dominated subcircuits in a circuit-based QBF. Section 5.4 describes the `PReDom` algorithm. Section 5.5 gives experimental results and Section 5.6 summarizes the chapter.

5.2 Preliminaries

5.2.1 Circuit-based Quantified Boolean Formulas

The reader is referred to Section 2.3 for an overview of QBFs in prenex normal form, given as $Q.\Phi$, where Q is the prefix and Φ is the matrix given in CNF. In many cases, the matrices of QBF encodings in VLSI CAD are originally given as logic circuits, *e.g.,* [79]. Here, we standardize such *circuit-based QBFs* as $Q.\mathcal{C}$, where $\mathcal{C}$ is a logic circuit consisting of interconnected logic gates, such as `NOT`, `AND`, `OR` and `XOR`, and a single primary output y, which must be satisfied (*i.e.*, set to `1`). $\mathcal{C}$ can be represented as a directed graph (V, E, y) with $|V|$ nodes (primary inputs and internal gates), $|E|$ edges and primary output $y \in V$. The variables in the prefix Q are the primary inputs $\{x_1, \ldots, x_n\}$ of $\mathcal{C}$.

In order to pass $Q.\mathcal{C}$ to a CNF-based QBF solver, first a CNF formula Φ expressing $\mathcal{C}$ is constructed in linear time, as detailed in Section 2.2.2, and the unit clause (y) is conjuncted to Φ, constraining the output to `1`. Recall that this translation to CNF requires the addition of auxiliary variables corresponding the internal circuit gates. Each of these gate variables is quantified existentially in the prefix Q, and is placed in a narrower scope compared to its gate inputs. Recall from Section 2.3 that a scope in the QBF prefix is said to be narrower than another scope if it is quantified after, or to the right, of the latter in the prefix. This forces each gate variable to simply emulate its gate in $\mathcal{C}$. Figure 5.1 shows a one-gate circuit-based QBF and its corresponding QBF in prenex normal form.

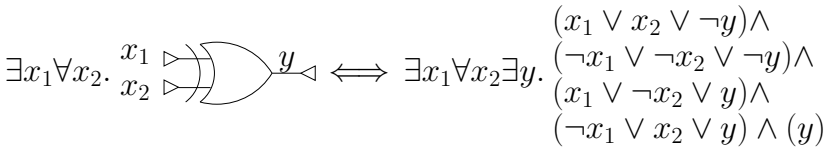


Figure 5.1: Circuit-based QBF to prenex normal form conversion

Recall from Section 2.3 that a closed QBF is one where all variables are bound to quantifiers, whereas an open QBF is one where some variables are free. Two closed QBFs (circuit-based or in prenex normal form) are said to be *logically equivalent* if they evaluate to the same logic

value (*i.e.*, either both are true or both are false). For example, any circuit-based QBF and its corresponding prenex normal form QBF, such as those shown in Figure 5.1, are logically equivalent. Two open QBFs with the same free variables are said to be logically equivalent if for every assignment to their free variables the resulting closed QBFs are logically equivalent. If $Q.\Phi$ and $Q'.\Phi'$ are logically equivalent, we can write $Q.\Phi = Q'.\Phi'$, however this does not imply that $Q = Q'$ or $\Phi = \Phi'$. The same is true for circuit-based QBFs. A function that transforms a QBF into another QBF is said to preserve logical equivalence if the two QBFs are always logically equivalent.

5.2.2 Complete Dominators

In a directed graph $C = (V, E, y)$, a node $u \in V$ *dominates* node $v \in V$, if every path from v to the output y passes through u. The set $dom(v) = \{u \in V | u \text{ dominates } v\}$ consists of nodes that dominate v. The inverse set $dom^{-1}(v) = \{u \in V | v \text{ dominates } u\}$ consists of nodes dominated by v.

Let the set $fanin(v) = \{u \in V | (u, v) \in E\}$ denote the fanins of v, the set $fanin^*(v) = \{u \in V | \exists \text{ a path } u \rightsquigarrow v \text{ in } C\}$ denote the transitive fanin cone of v, and the set $faninPI^*(v) = \{u \in fanin^*(v) | \forall w \in V. \nexists (w, u) \in E\}$ denote the primary inputs in the transitive fanin cone of v. We call a node v a *complete dominator* if it dominates every node in $fanin^*(v)$, *i.e.,* if $dom^{-1}(v) = fanin^*(v)$.

Consider the circuit in Figure 5.2. Here, $dom(x_2) = \{x_2, \alpha, y\}$, $dom^{-1}(g_3) = \{x_1, x_3, g_1, g_2, g_3\}$ and $faninPI^*(\alpha) = \{x_1, x_2, x_3\}$. Note that $dom^{-1}(\alpha) = fanin^*(\alpha) = \{x_1, x_2, x_3, g_1, g_2, g_3, \alpha\}$, hence α is a complete dominator. We refer to the transitive fanin cone $fanin^*$ of a complete dominator α as a *single-output dominated subcircuit* (SODS).

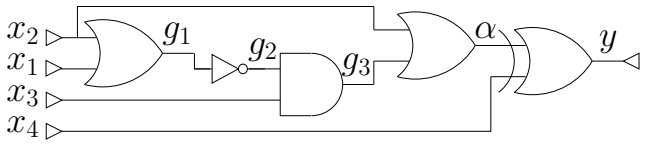


Figure 5.2: A circuit with a complete dominator α

Many algorithms have been developed for computing the set $dom(v)$ for each $v \in V$. The run-times of these algorithms have improved from $O(|V|^2)$ [7] to $O(|E| + |V|)$ [45].

5.3 Reducing SODSes in a Circuit-based QBF

This section presents the theory behind a novel circuit-based QBF reduction scheme, which uses complete dominators. The aim is to obtain a logically equivalent circuit-based QBF which is smaller than the original, and therefore easier to solve by a QBF solver. For instance, we will show that in the QBF $\exists x_1 \forall x_2 \exists x_3 \forall x_4 . \mathcal{C}$, for the circuit given in Figure 5.2, we can assign the complete dominator α to `1` and remove its fanin-cone while preserving logical equivalence.

In the following subsection, we give a motivating example to illustrate the basic idea, where a single `AND` gate can be removed and replaced by a constant. In Subsection 5.3.2, we consider the general case, and give a formal proof showing how to appropriately reduce any SODS in a circuit-based QBF and replace its output by a constant truth value or a quantified primary input, all while preserving logical equivalence. Note that, by definition, the inputs of the SODS must be primary inputs, and the approach is not applicable to dominated subcircuits that do not extend all the way to primary inputs.

5.3.1 A Motivating Example

Before discussing the example, we must introduce some notation. Let $Q.\mathcal{C}$ denote the original circuit-based QBF and $f_\alpha(Q.\mathcal{C})$ denote the reduced QBF after the elimination of the SODS of the complete dominator α. We call $f_\alpha(Q.\mathcal{C})$ the *α-reduced* QBF of $Q.\mathcal{C}$. The function $f_\alpha(Q.\mathcal{C})$ can yield one of the following four types of reductions:

- $[Q.\mathcal{C}]_{\alpha=\mathtt{0}}$ (or $[Q.\mathcal{C}]_{\alpha=\mathtt{1}}$) denotes the QBF $Q.\mathcal{C}$ where α has been replaced by `0` (or `1`), $fanin^*(\alpha)$ has been removed from $\mathcal{C}$ and the variables in $faninPI^*(\alpha)$ have been removed from Q.

- $[Q.\mathcal{C}]_{\alpha=\exists z}$ (or $[Q.\mathcal{C}]_{\alpha=\forall z}$) denotes the formula $Q.\mathcal{C}$ where α has been replaced by an existential (or a universal) primary input z, where $z \in faninPI^*(\alpha)$. Again, $fanin^*(\alpha)$ has

been removed from $\mathcal{C}$ and the variables in $faninPI^*(\alpha) - \{z\}$ have been removed from Q.

Now consider the circuit-based QBF in Figure 5.3. The circuit $\mathcal{C}$ contains the AND gate α, which is a complete dominator. Note that α may fan out to any number of other gates, and $\mathcal{C}$ may contain any number of other inputs, arbitrarily quantified in Q. Let $Q.\mathcal{C} = \forall x_1 \exists x_2 Q'.\mathcal{C}$ denote this circuit-based QBF.

Claim The QBF $Q.\mathcal{C}$ given in Figure 5.3 and $[Q.\mathcal{C}]_{\alpha=0}$ are logically equivalent.

Proof: We must prove that: (a) $Q.\mathcal{C}$ is true $\Rightarrow [Q.\mathcal{C}]_{\alpha=0}$ is true, and (b) $Q.\mathcal{C}$ is false $\Rightarrow [Q.\mathcal{C}]_{\alpha=0}$ is false.

(a) By definition, $Q.\mathcal{C} = \forall x_1 \exists x_2 Q'.\mathcal{C} = \exists x_2 Q'.\mathcal{C}|_{\neg x_1} \wedge \exists x_2 Q'.\mathcal{C}|_{x_1}$. So $Q.\mathcal{C}$ is true $\Rightarrow \exists x_2 Q'.\mathcal{C}|_{\neg x_1}$ is true. But $x_1 = 0 \Rightarrow \alpha = 0$, hence $\exists x_2 Q'.(\mathcal{C}|_{\neg x_1} \wedge (\neg\alpha))$ is true. At this point, since x_2 is completely dominated by α, we can set it to an arbitrary value (*e.g.,* $x_2 = 0$) without affecting the output y, so $Q'.(\mathcal{C}|_{\neg x_1, \neg x_2} \wedge (\neg\alpha))$ is true. Since the inputs and output of α are assigned, we can just remove the AND gate from the circuit. Furthermore, since x_1 and x_2 can affected the primary output y only through the (now removed) AND gate α, we can remove x_1 and x_2 from $\mathcal{C}$ and Q, and hence, $[Q.\mathcal{C}]_{\alpha=0}$ is true.

(b) By definition, $Q.\mathcal{C}$ is false $\Rightarrow \exists x_2 Q'.\mathcal{C}|_{\neg x_1} \wedge \exists x_2 Q'.\mathcal{C}|_{x_1}$ is false. So either (b.1) $\exists x_2 Q'.\mathcal{C}|_{\neg x_1}$ is false or (b.2) $\exists x_2 Q'.\mathcal{C}|_{x_1}$ is false. We show that each of (b.1) and (b.2) implies that $[Q.\mathcal{C}]_{\alpha=0}$ is false.

(b.1) Analogously to (a), (b.1) $\Rightarrow \exists x_2 Q'.(\mathcal{C}|_{\neg x_1} \wedge (\neg\alpha))$ is false $\Rightarrow [Q.\mathcal{C}]_{\alpha=0}$ is false.

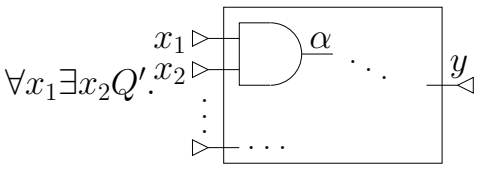


Figure 5.3: A circuit-based QBF

(b.2) By definition, $\exists x_2 Q'.\mathcal{C}|_{x_1} = Q'.\mathcal{C}|_{x_1,\neg x_2} \vee Q'.\mathcal{C}|_{x_1,x_2}$. Therefore, (b.2) $\Rightarrow Q'.\mathcal{C}|_{x_1,\neg x_2}$ is false and $Q'.\mathcal{C}|_{x_1,x_2}$ is false. But $Q'.\mathcal{C}|_{x_1,\neg x_2}$ is false $\Rightarrow [Q.\mathcal{C}]_{\alpha=0}$ is false, since x_1 and x_2 can affect y only through α.

■

Note that $Q.\mathcal{C}$ and $[Q.\mathcal{C}]_{\alpha=0}$ being logically equivalent does not mean that corresponding gates in the two circuits always evaluate to the same value given the same primary input assignment. In fact, setting $x_1 = 1$ and $x_2 = 1$ in $Q.\mathcal{C}$ makes $\alpha = 1$, whereas α is grounded to 0 in $[Q.\mathcal{C}]_{\alpha=0}$. On the other hand, a QBF solver will evaluate both $Q.\mathcal{C}$ and $[Q.\mathcal{C}]_{\alpha=0}$ to the same value (true or false).

5.3.2 The General Case

Given a complete dominator α with an arbitrary SODS, we will show how to construct $f_\alpha(Q.\mathcal{C})$ and prove that it produces a QBF that is logically equivalent to the original QBF $Q.\mathcal{C}$.

The scopes in the prefix Q impose a partial order $\prec$ on the variables $\{x_1, \ldots, x_n\}$, such that $x_i \prec x_j$ if and only if the scope of x_i is wider than the scope of x_j. For the sake of the construction of $f_\alpha(Q.\mathcal{C})$ and its proof of correctness, we will use an arbitrary total order on the prefix Q, which respects this partial order and arbitrarily orders variables that are in the same scope. In other terms, we will write the prefix as:

$$Q = q_1 x_1 \; q_2 x_2 \; \cdots \; q_n x_n$$

Furthermore, let $Q_i = q_{i+1} x_{i+1} \; q_{i+2} x_{i+2} \; \cdots \; q_n x_n$. Hence, $Q_i = q_{i+1} x_{i+1} Q_{i+1}$ and we have:

$$\begin{aligned} Q_0 &= q_1 x_1 \; q_2 x_2 \; \cdots \; q_n x_n = Q \\ Q_1 &= q_2 x_2 \; \cdots \; q_n x_n \\ &\vdots \\ Q_n &= \emptyset. \end{aligned}$$

Notice that $Q_0.\mathcal{C} = Q.\mathcal{C}$. Also, if $1 \leq i \leq n$, then $Q_i.\mathcal{C}$ is an open QBF where the variables $\{x_1, \ldots, x_i\}$ are free (*i.e.*, unquantified in Q_i). We will use the symbol π_i to denote a truth

assignment that assigns exactly the variables in $faninPI^*(\alpha)$ which are free in $Q_i.\mathcal{C}$. In other terms, if $1 \leq i \leq n$, π_i is a truth assignment to $\{x_1, \ldots, x_i\} \cap faninPI^*(\alpha)$, and $\pi_0 = \emptyset$. In order to construct $f_\alpha(Q.\mathcal{C})$, we will need to recursively define a two-argument function $f_\alpha(Q_i.\mathcal{C}, \pi_i)$, which produces a logically equivalent QBF to $Q_i.\mathcal{C}$ *under the truth assignment* π_i. In other terms, $f_\alpha(Q_i.\mathcal{C}, \pi_i)$ must be logically equivalent to $Q_i.\mathcal{C}|_{\pi_i}$. As a result, by definition, we get $f_\alpha(Q_0.\mathcal{C}, \pi_0) = f_\alpha(Q.\mathcal{C}, \emptyset) = f_\alpha(Q.\mathcal{C}|_\emptyset) = f_\alpha(Q.\mathcal{C})$.

Similarly to the final reduction $f_\alpha(Q.\mathcal{C})$, the function $f_\alpha(Q_i.\mathcal{C}, \pi_i)$ can yield one of the following four types of intermediate reductions:

- $[Q_i.\mathcal{C}]_{\alpha=0}$ (or $[Q_i.\mathcal{C}]_{\alpha=1}$) replaces α in $Q_i.\mathcal{C}$ by 0 (or 1), removes $fanin^*(\alpha)$ from $\mathcal{C}$ and removes the variables in $faninPI^*(\alpha)$ from Q_i.

- $[Q_i.\mathcal{C}]_{\alpha=\exists z}$ (or $[Q_i.\mathcal{C}]_{\alpha=\forall z}$) replaces α in $Q_i.\mathcal{C}$ by an existential (or a universal) primary input z, where $z \in \{x_{i+1}, \ldots, x_n\} \cap faninPI^*(\alpha)$. It also removes $fanin^*(\alpha)$ from $\mathcal{C}$ and removes the variables in $faninPI^*(\alpha) - \{z\}$ from Q_i.

Note that unlike $f_\alpha(Q.\mathcal{C})$, the QBF $f_\alpha(Q_i.\mathcal{C}, \pi_i)$ is not necessarily closed, *i.e.,* it can have free variables.

Definition 5.1 *Given a truth assignment* π_i*, the QBF* $f_\alpha(Q_i.\mathcal{C}, \pi_i)$ *is recursively defined as follows:*

(a) If $i = n$*, these are the base cases: All the variables in* $Q_n.\mathcal{C} = \mathcal{C}$ *are free and* π_n *assigns all the variables in* $faninPI^*(\alpha)$*. Then:*

$$f_\alpha(Q_n.\mathcal{C}, \pi_n) \triangleq \begin{cases} [Q_n.\mathcal{C}]_{\alpha=0} = [\mathcal{C}]_{\alpha=0} & \text{if } \alpha = 0 \text{ under } \pi_n \\ [Q_n.\mathcal{C}]_{\alpha=1} = [\mathcal{C}]_{\alpha=1} & \text{if } \alpha = 1 \text{ under } \pi_n \end{cases}$$

(b) If $0 \leq i < n$ *and* $x_{i+1} \notin faninPI^*(\alpha)$*, then:*

$$f_\alpha(Q_i.\mathcal{C}, \pi_i) \triangleq q_{i+1} x_{i+1} f_\alpha(Q_{i+1}.\mathcal{C}, \pi_{i+1}), \text{ where } \pi_i = \pi_{i+1}$$

Note that $\pi_i = \pi_{i+1}$ *is legal since* π_{i+1} *cannot assign* $x_{i+1} \notin faninPI^*(\alpha)$.

(c) If $0 \leq i < n$ *and* $x_{i+1} \in faninPI^*(\alpha)$*, then:*

$$f_\alpha(Q_i.\mathcal{C}, \pi_i) \triangleq \begin{cases} f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = 0\}) \vee f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = 1\}) & \text{if } q_{i+1} = \exists \\ f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = 0\}) \wedge f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = 1\}) & \text{if } q_{i+1} = \forall \end{cases}$$

The resulting QBF $f_\alpha(Q_i.\mathcal{C}, \pi_i)$ *is given in Table 5.1 for each* q_{i+1} *and each reduction type of* $f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = 0\})$ *and* $f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = 1\})$*.*

In Table 5.1, we use the notation $a \uparrow b$ *(respectively,* $a \downarrow b$*) for two variables* a *and* b *to denote the one with the widest (respectively, narrowest) scope. For example, if* $x_i \prec x_j$*,* $x_i \uparrow x_j = x_i$ *and* $x_i \downarrow x_j = x_j$*.*

In the base case (a) of Definition 5.1, given a truth assignment π_n that sets all the variables in $faninPI^*(\alpha)$, the reduction $f_\alpha(Q_n.\mathcal{C}, \pi_n) = f_\alpha(\mathcal{C}, \pi_n)$ is simply obtained using *circuit simulation*: α is assigned to its simulated value under π_n and its fanin cone is removed. This yields $[\mathcal{C}]_{\alpha=0}$ (respectively, $[\mathcal{C}]_{\alpha=1}$) if the simulated value of α is 0 (respectively, 1). There are $2^{|faninPI^*(\alpha)|}$ base cases, one for each assignment combination to π_n.

For case (c) of Definition 5.1, Table 5.1 gives $f_\alpha(Q_i.\mathcal{C}, \pi_i)$ for each $q_{i+1} \in \{\exists, \forall\}$, and for each of the four reduction types to $f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = 0\})$ and $f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = 1\})$, namely $[Q_{i+1}.\mathcal{C}]_{\alpha=0}$, $[Q_{i+1}.\mathcal{C}]_{\alpha=1}$, $[Q_{i+1}.\mathcal{C}]_{\alpha=\exists z}$ and $[Q_{i+1}.\mathcal{C}]_{\alpha=\forall z}$. As such, for each $q_{i+1} \in \{\exists, \forall\}$, the number of different cases that need to be considered is equal to the number of ways to pick two out of these four reduction types for $f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = 0\})$ and $f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = 1\})$, with repetition but without order. This is given by:

$$\binom{4+2-1}{2} = \binom{5}{2} = 10 \text{ cases.}$$

The correctness of the results shown in Table 5.1, as well as the other cases in Definition 5.1, is proven in Theorem 5.1.

We give some intuition for some of the cases in Table 5.1. Case 1 considers the case where $Q_{i+1}.\mathcal{C}$ can be reduced to $[Q_{i+1}.\mathcal{C}]_{\alpha=0}$ under both assignments $\pi_i \cup \{x_{i+1} = 0\}$ and $\pi_i \cup \{x_{i+1} = 1\}$. Assuming that $q_{i+1} = \exists$, by part (c) of Definition 5.1, $f_\alpha(Q_i.\mathcal{C}, \pi_i) = f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = 0\}) \vee f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = 1\}) = [Q_{i+1}.\mathcal{C}]_{\alpha=0} \vee [Q_{i+1}.\mathcal{C}]_{\alpha=0} = [Q_{i+1}.\mathcal{C}]_{\alpha=0}$. This in turn is logically equivalent to $[Q_i.\mathcal{C}]_{\alpha=0}$, since $q_{i+1}x_{i+1}$ is removed from the prefix of $[Q_i.\mathcal{C}]_{\alpha=0}$ anyway

Table 5.1: $f_\alpha(Q_i.\mathcal{C}, \pi_i)$ for case (c) of Definition 5.1 when $i < n$ and $x_{i+1} \in faninPI^*(\alpha)$

Case	$f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x = 0\})$ and $f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x = 1\})$		$f_\alpha(Q_i.\mathcal{C}, \pi_i)$	
			$q_{i+1} = \exists$	$q_{i+1} = \forall$
1	$[Q_{i+1}.\mathcal{C}]_{\alpha=0}$	$[Q_{i+1}.\mathcal{C}]_{\alpha=0}$	$[Q_i.\mathcal{C}]_{\alpha=0}$	$[Q_i.\mathcal{C}]_{\alpha=0}$
2	$[Q_{i+1}.\mathcal{C}]_{\alpha=0}$	$[Q_{i+1}.\mathcal{C}]_{\alpha=1}$	$[Q_i.\mathcal{C}]_{\alpha=\exists x_{i+1}}$	$[Q_i.\mathcal{C}]_{\alpha=\forall x_{i+1}}$
3	$[Q_{i+1}.\mathcal{C}]_{\alpha=0}$	$[Q_{i+1}.\mathcal{C}]_{\alpha=\exists z}$	$[Q_i.\mathcal{C}]_{\alpha=\exists z}$	$[Q_i.\mathcal{C}]_{\alpha=0}$
4	$[Q_{i+1}.\mathcal{C}]_{\alpha=0}$	$[Q_{i+1}.\mathcal{C}]_{\alpha=\forall z}$	$[Q_i.\mathcal{C}]_{\alpha=0}$	$[Q_i.\mathcal{C}]_{\alpha=\forall z}$
5	$[Q_{i+1}.\mathcal{C}]_{\alpha=1}$	$[Q_{i+1}.\mathcal{C}]_{\alpha=1}$	$[Q_i.\mathcal{C}]_{\alpha=1}$	$[Q_i.\mathcal{C}]_{\alpha=1}$
6	$[Q_{i+1}.\mathcal{C}]_{\alpha=1}$	$[Q_{i+1}.\mathcal{C}]_{\alpha=\exists z}$	$[Q_i.\mathcal{C}]_{\alpha=\exists z}$	$[Q_i.\mathcal{C}]_{\alpha=1}$
7	$[Q_{i+1}.\mathcal{C}]_{\alpha=1}$	$[Q_{i+1}.\mathcal{C}]_{\alpha=\forall z}$	$[Q_i.\mathcal{C}]_{\alpha=1}$	$[Q_i.\mathcal{C}]_{\alpha=\forall z}$
8	$[Q_{i+1}.\mathcal{C}]_{\alpha=\exists z_1}$	$[Q_{i+1}.\mathcal{C}]_{\alpha=\exists z_2}$	$[Q_i.\mathcal{C}]_{\alpha=\exists z_1 \downarrow z_2}$	$[Q_i.\mathcal{C}]_{\alpha=\exists z_1 \uparrow z_2}$
9	$[Q_{i+1}.\mathcal{C}]_{\alpha=\exists z_1}$	$[Q_{i+1}.\mathcal{C}]_{\alpha=\forall z_2}$	$[Q_i.\mathcal{C}]_{\alpha=\exists z_1}$	$[Q_i.\mathcal{C}]_{\alpha=\forall z_2}$
10	$[Q_{i+1}.\mathcal{C}]_{\alpha=\forall z_1}$	$[Q_{i+1}.\mathcal{C}]_{\alpha=\forall z_2}$	$[Q_i.\mathcal{C}]_{\alpha=\forall z_1 \uparrow z_2}$	$[Q_i.\mathcal{C}]_{\alpha=\forall z_1 \downarrow z_2}$

(given that $x_{i+1} \in faninPI^*(\alpha)$). Considering $q_{i+1} = \forall$ gives a similar result, and case 5 is analogous to case 1. Case 2 is simply an application of the definition of quantification.

Consider case 7 in Table 5.1 with $q_{i+1} = \forall$. Then, according to Definition 5.1, $f_\alpha(Q_i.\mathcal{C}, \pi_i) = [Q_{i+1}.\mathcal{C}]_{\alpha=1} \wedge [Q_{i+1}.\mathcal{C}]_{\alpha=\forall z}$. This is logically equivalent to $[Q_i.\mathcal{C}]_{\alpha=1} \wedge [Q_i.\mathcal{C}]_{\alpha=\forall z}$ by the same argument as above. The QBFs $[Q_i.\mathcal{C}]_{\alpha=1}$ and $[Q_i.\mathcal{C}]_{\alpha=\forall z}$ have the same free variables (if any): The variables in $\{x_1, \ldots, x_i\}$ that are not in $faninPI^*(\alpha)$. Now notice that for every assignment to these free variables, the QBF $[Q_i.\mathcal{C}]_{\alpha=\forall z}$ cannot be true if the QBF $[Q_i.\mathcal{C}]_{\alpha=1}$ is not true. In other terms, $[Q_i.\mathcal{C}]_{\alpha=\forall z} \Rightarrow [Q_i.\mathcal{C}]_{\alpha=1}$. This in turn means that the truth of $[Q_i.\mathcal{C}]_{\alpha=\forall z}$ is a necessary and sufficient condition for the truth of $[Q_i.\mathcal{C}]_{\alpha=1} \wedge [Q_i.\mathcal{C}]_{\alpha=\forall z}$. In other terms, $[Q_i.\mathcal{C}]_{\alpha=1} \wedge [Q_i.\mathcal{C}]_{\alpha=\forall z} = [Q_i.\mathcal{C}]_{\alpha=\forall z}$, as shown in case 7 when $q_{i+1} = \forall$. Cases 3, 4, 5, 6 and 9 are analogous to case 7, for either $q_{i+1} \in \{\exists, \forall\}$.

Finally, consider case 8 with $q_{i+1} = \exists$. Then, $f_\alpha(Q_i.\mathcal{C}, \pi_i) = [Q_{i+1}.\mathcal{C}]_{\alpha=\exists z_1} \vee [Q_{i+1}.\mathcal{C}]_{\alpha=\exists z_2}$, which is logically equivalent to $[Q_i.\mathcal{C}]_{\alpha=\exists z_1} \vee [Q_i.\mathcal{C}]_{\alpha=\exists z_2}$. Now the key is to notice that if the QBF $[Q_i.\mathcal{C}]_{\alpha=\exists z_1 \uparrow z_2}$ is true, then the QBF $[Q_i.\mathcal{C}]_{\alpha=\exists z_1 \downarrow z_2}$ is also true. The reason is as follows. In $[Q_i.\mathcal{C}]_{\alpha=\exists z_1 \uparrow z_2}$, the scope of α is wider than in $[Q_i.\mathcal{C}]_{\alpha=\exists z_1 \downarrow z_2}$. As such, the Skolem function of α in a Q-model of the QBF $[Q_i.\mathcal{C}]_{\alpha=\exists z_1 \uparrow z_2}$ is a function of a subset of the universal variables that the Skolem function of α in a Q-model of the QBF $[Q_i.\mathcal{C}]_{\alpha=\exists z_1 \downarrow z_2}$ is a function of. Put simply, α has less flexibility in a Q-model of $[Q_i.\mathcal{C}]_{\alpha=\exists z_1 \uparrow z_2}$ than in a Q-model of $[Q_i.\mathcal{C}]_{\alpha=\exists z_1 \downarrow z_2}$. As such, if a Q-model for $[Q_i.\mathcal{C}]_{\alpha=\exists z_1 \uparrow z_2}$ exists, then a Q-model for $[Q_i.\mathcal{C}]_{\alpha=\exists z_1 \downarrow z_2}$ can be constructed from it by simply disregarding those extra universal variables that α is now allowed to be a function of. Therefore, $[Q_i.\mathcal{C}]_{\alpha=\exists z_1 \uparrow z_2} \Rightarrow [Q_i.\mathcal{C}]_{\alpha=\exists z_1 \downarrow z_2}$. This in turn means that the truth of $[Q_i.\mathcal{C}]_{\alpha=\exists z_1 \downarrow z_2}$ is a necessary and sufficient condition for the truth of $[Q_{i+1}.\mathcal{C}]_{\alpha=\exists z_1} \vee [Q_{i+1}.\mathcal{C}]_{\alpha=\exists z_2}$. In other terms, $[Q_{i+1}.\mathcal{C}]_{\alpha=\exists z_1} \vee [Q_{i+1}.\mathcal{C}]_{\alpha=\exists z_2} = [Q_i.\mathcal{C}]_{\alpha=\exists z_1 \downarrow z_2}$, as shown in case 8 when $q_{i+1} = \exists$. Case 10 is analogous to case 8, for either $q_{i+1} \in \{\exists, \forall\}$.

In order to construct $f_\alpha(Q.\mathcal{C}, \emptyset)$ using Definition 5.1, one has to proceed in a bottom-up recursive fashion. Starting from all the base cases, where $i = n$ and $Q_n.\mathcal{C} = \mathcal{C}$, the function $f_\alpha(\mathcal{C}, \pi_n)$ must be computed for every π_n assigning every combination of $faninPI^*(\alpha)$. Case (a) of Definition 5.1 deals with base cases. Next, for $i = n - 1$, we have $Q_{n-1}.\mathcal{C} = q_n x_n.\mathcal{C}$, and now $f_\alpha(q_n x_n.\mathcal{C}, \pi_{n-1})$ must be computed for every π_{n-1} assigning every combination of

$\{x_1, \ldots, x_{n-1}\} \cap faninPI^*(\alpha)$. One of case (b) or (c) of Definition 5.1 would dictate these computations, depending on whether $x_n \in faninPI^*(\alpha)$ or not. At each subsequent step, i is decreased by one, and $f_\alpha(Q_i\mathcal{C}, \pi_i)$ is computed according to Definition 5.1. In the final step, $i = 0$ and $f_\alpha(Q_0.\mathcal{C}, \pi_0) = f_\alpha(Q.\mathcal{C}, \emptyset)$ is computed. This will be illustrated in detail later using an example.

Theorem 5.1 *For all i and any assignment π_i to the variables in $\{x_1, \ldots, x_i\} \cap faninPI^*(\alpha)$, $Q_i.\mathcal{C}|_{\pi_i}$ and $f_\alpha(Q_i.\mathcal{C}, \pi_i)$ are logically equivalent.*

Proof: The proof is by induction over i, and follows the recursive definition of the function $f_\alpha(Q_i.\mathcal{C}, \pi_i)$ in Definition 5.1. The inductive hypothesis is that $Q_{i+1}.\mathcal{C}|_{\pi_{i+1}}$ and $f_\alpha(Q_{i+1}.\mathcal{C}, \pi_{i+1})$ are logically equivalent for all valid π_{i+1}. The inductive step is that $Q_i.\mathcal{C}|_{\pi_i}$ and $f_\alpha(Q_i.\mathcal{C}, \pi_i)$ are logically equivalent for all valid π_i.

(a) If $i = n$ (base cases). All the variables in $Q_n.\mathcal{C} = \mathcal{C}$ are free and any π_n assigns all the variables in $faninPI^*(\alpha)$. So $f_\alpha(Q_n.\mathcal{C}, \pi_n) = f_\alpha(\mathcal{C}, \pi_n) \in \{[\mathcal{C}]_{\alpha=0}, [\mathcal{C}]_{\alpha=1}\}$ simply replaces α by its value under π_n, which is clearly logically equivalent to $\mathcal{C}|_{\pi_n}$.

(b) If $i < n$ and $x_{i+1} \notin faninPI^*(\alpha)$. Note that $\pi_i = \pi_{i+1}$ because $x_{i+1} \notin faninPI^*(\alpha)$. We can add the prefix $q_{i+1}x_{i+1}$ to both sides of the inductive hypothesis because x_{i+1} does not participate in the transformation. We get that $q_{i+1}x_{i+1}f_\alpha(Q_{i+1}.\mathcal{C}, \pi_{i+1})$ and $q_{i+1}x_{i+1}Q_{i+1}.\mathcal{C}|_{\pi_{i+1}}$ are logically equivalent. Since $q_{i+1}x_{i+1}f_\alpha(Q_{i+1}.\mathcal{C}, \pi_{i+1}) = f_\alpha(Q_i.\mathcal{C}, \pi_i)$ by case (b) of Definition 5.1, hence $f_\alpha(Q.\mathcal{C}, \pi)$ and $Q.\mathcal{C}|_\pi$ are logically equivalent.

(c) If $i < n$ and $x_{i+1} \in faninPI^*(\alpha)$. By the definition of quantification, we have:

$$\exists x_{i+1}Q_{i+1}.\mathcal{C}|_{\pi_i} = Q_{i+1}.\mathcal{C}|_{\pi_i,\neg x_{i+1}} \vee Q_{i+1}.\mathcal{C}|_{\pi_i,x_{i+1}} \quad \text{and}$$
$$\forall x_{i+1}Q_{i+1}.\mathcal{C}|_{\pi_i} = Q_{i+1}.\mathcal{C}|_{\pi_i,\neg x_{i+1}} \wedge Q_{i+1}.\mathcal{C}|_{\pi_i,x_{i+1}}.$$

Therefore:

$$Q_i.\mathcal{C}|_{\pi_i} = q_{i+1}x_{i+1}Q_{i+1}.\mathcal{C}|_{\pi_i} = \begin{cases} Q_{i+1}.\mathcal{C}|_{\pi_i,\neg x_{i+1}} \vee Q_{i+1}.\mathcal{C}|_{\pi_i,x_{i+1}} & \text{if } q_{i+1} = \exists \\ Q_{i+1}.\mathcal{C}|_{\pi_i,\neg x_{i+1}} \wedge Q_{i+1}.\mathcal{C}|_{\pi_i,x_{i+1}} & \text{if } q_{i+1} = \forall. \end{cases} \tag{5.1}$$

By the inductive hypothesis:

$Q_{i+1}.\mathcal{C}|_{\pi_i,\neg x_{i+1}}$ and $f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = \mathtt{0}\})$ are logically equivalent, and

$Q_{i+1}.\mathcal{C}|_{\pi_i,x_{i+1}}$ and $f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = \mathtt{1}\})$ are logically equivalent.

Replacing these functions in (5.1), we get that $Q_i.\mathcal{C}|_{\pi_i}$ is logically equivalent to:

$$\begin{cases} f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = \mathtt{0}\}) \vee f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = \mathtt{1}\}) & \text{if } q_{i+1} = \exists \\ f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = \mathtt{0}\}) \wedge f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = \mathtt{1}\}) & \text{if } q_{i+1} = \forall. \end{cases} \tag{5.2}$$

It remains to show that (5.2) is logically equivalent to the results displayed in the last two columns of Table 5.1, for every $f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = \mathtt{0}\})$ and $f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = \mathtt{1}\})$ given by the ten cases in Table 5.1.

- Case 1: Here, $f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = \mathtt{0}\}) = f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = \mathtt{1}\}) = [Q_{i+1}.\mathcal{C}]_{\alpha=\mathtt{0}}$, so irrespective of q_{i+1}, (5.2) yields $[Q_{i+1}.\mathcal{C}]_{\alpha=\mathtt{0}}$.
- Case 2: Here, (5.2) gives:

  $$\begin{cases} [Q_{i+1}.\mathcal{C}]_{\alpha=\mathtt{0}} \vee [Q_{i+1}.\mathcal{C}]_{\alpha=\mathtt{1}} & \text{if } q_{i+1} = \exists \\ [Q_{i+1}.\mathcal{C}]_{\alpha=\mathtt{0}} \wedge [Q_{i+1}.\mathcal{C}]_{\alpha=\mathtt{1}} & \text{if } q_{i+1} = \forall. \end{cases}$$

  In the QBF $[Q_{i+1}.\mathcal{C}]_{\alpha=\mathtt{0}}$ (respectively, $[Q_{i+1}.\mathcal{C}]_{\alpha=\mathtt{1}}$), α is simply a primary input set to $\mathtt{0}$ (respectively, $\mathtt{1}$). Therefore, we can apply the definition of quantification in the other direction as follows:

  $$\begin{cases} [Q_{i+1}.\mathcal{C}]_{\alpha=\mathtt{0}} \vee [Q_{i+1}.\mathcal{C}]_{\alpha=\mathtt{1}} = \exists x_{i+1}[Q_{i+1}.\mathcal{C}]_{\alpha=\exists x_{i+1}} = [Q_i.\mathcal{C}]_{\alpha=\exists x_{i+1}} & \text{if } q_{i+1} = \exists \\ [Q_{i+1}.\mathcal{C}]_{\alpha=\mathtt{0}} \wedge [Q_{i+1}.\mathcal{C}]_{\alpha=\mathtt{1}} = \forall x_{i+1}[Q_i.\mathcal{C}]_{\alpha=\forall x_{i+1}} = [Q_i.\mathcal{C}]_{\alpha=\forall x_{i+1}} & \text{if } q_{i+1} = \forall, \end{cases}$$

  Conveniently replacing α by the variable in the current scope, which is x_{i+1}.
- Case 5: Analogous to case 1, but replacing $[Q_{i+1}.\mathcal{C}]_{\alpha=\mathtt{0}}$ by $[Q_{i+1}.\mathcal{C}]_{\alpha=\mathtt{1}}$.
- Cases 3, 4, 6, 7, 8, 9, 10: Recall from formal logic that if $a \Rightarrow b$, then $a \wedge b = a$ and $a \vee b = b$. As such:
  - If $f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = \mathtt{0}\}) \Rightarrow f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = \mathtt{1}\})$, then (5.2) becomes:

    $$\begin{cases} f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = \mathtt{1}\}) & \text{if } q_{i+1} = \exists \\ f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = \mathtt{0}\}) & \text{if } q_{i+1} = \forall. \end{cases} \tag{5.3}$$

∗ If $f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = \mathtt{1}\}) \Rightarrow f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = \mathtt{0}\})$, then (5.2) becomes:

$$\begin{cases} f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = \mathtt{0}\}) & \text{if } q_{i+1} = \exists \\ f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = \mathtt{1}\}) & \text{if } q_{i+1} = \forall. \end{cases} \tag{5.4}$$

Hence, it suffices to prove the implication $f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = \mathtt{0}\}) \Rightarrow f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_{i+1} = \mathtt{1}\})$ (or vice-versa) for each of these cases.

3 $[Q_{i+1}.\mathcal{C}]_{\alpha=\mathtt{0}} \Rightarrow [Q_{i+1}.\mathcal{C}]_{\alpha=\exists z}$. If $[Q_{i+1}.\mathcal{C}]_{\alpha=\mathtt{0}}$ is true, we can make $[Q_{i+1}.\mathcal{C}]_{\alpha=\exists z}$ true by setting $z = \mathtt{0}$.

4 $[Q_{i+1}.\mathcal{C}]_{\alpha=\forall z} \Rightarrow [Q_{i+1}.\mathcal{C}]_{\alpha=\mathtt{0}}$. By definition, if $[Q_{i+1}.\mathcal{C}]_{\alpha=\forall z}$ is true, it must be true for both $z = \mathtt{0}$ and $z = \mathtt{1}$. In particular, it must be true when $z = \mathtt{0}$.

6 $[Q_{i+1}.\mathcal{C}]_{\alpha=\mathtt{1}} \Rightarrow [Q_{i+1}.\mathcal{C}]_{\alpha=\exists z}$. Analogous to case 3.

7 $[Q_{i+1}.\mathcal{C}]_{\alpha=\forall z} \Rightarrow [Q_{i+1}.\mathcal{C}]_{\alpha=\mathtt{1}}$. Analogous to case 4.

8 $[Q_{i+1}.\mathcal{C}]_{\alpha=\exists z_1 \uparrow z_2} \Rightarrow [Q_{i+1}.\mathcal{C}]_{\alpha=\exists z_1 \downarrow z_2}$. Without loss of generality, assume that $z_1 \uparrow z_2 = z_1$ and $z_1 \downarrow z_2 = z_2$. If $[Q_{i+1}.\mathcal{C}]_{\alpha=\exists z_1}$ is true, then it has a Q-model in which the value of α is a function of all universal variables with a wider scope than z_1. Since the scope of z_2 is narrower, in $[Q_{i+1}.\mathcal{C}]_{\alpha=\exists z_2}$ the value of α is a function of all those and possibly additional universal variables. Therefore, the value of α in a Q-model of $[Q_{i+1}.\mathcal{C}]_{\alpha=\exists z_2}$ can emulate α in $[Q_{i+1}.\mathcal{C}]_{\alpha=\exists z_1}$ by ignoring the remaining universal variables with scopes between z_1 and z_2, which α is also a function of in $[Q_{i+1}.\mathcal{C}]_{\alpha=\exists z_2}$.

9 $[Q_{i+1}.\mathcal{C}]_{\alpha=\forall z_2} \Rightarrow [Q_{i+1}.\mathcal{C}]_{\alpha=\exists z_1}$. The proof can be composed from Cases 3 and 4 as follows: $[Q_{i+1}.\mathcal{C}]_{\alpha=\forall z_2} \Rightarrow [Q_{i+1}.\mathcal{C}]_{\alpha=\mathtt{0}} \Rightarrow [Q_{i+1}.\mathcal{C}]_{\alpha=\exists z_1}$.

10 $[Q_{i+1}.\mathcal{C}]_{\alpha=\forall z_1 \downarrow z_2} \Rightarrow [Q_{i+1}.\mathcal{C}]_{\alpha=\forall z_1 \uparrow z_2}$. Without loss of generality, assume that $z_1 \downarrow z_2 = z_1$ and $z_1 \uparrow z_2 = z_2$. Widening the scope of the universal α from that of z_1 to that of z_2 will make the existential variables between those scopes also a function of α. Since $[Q_{i+1}.\mathcal{C}]_{\alpha=\forall z_1}$ is true, to produce a Q-model for $[Q_{i+1}.\mathcal{C}]_{\alpha=\forall z_2}$ the existential variables between the scopes of z_1 and z_2 can emulate their values in a Q-model of $[Q_{i+1}.\mathcal{C}]_{\alpha=\forall z_1}$, ignoring α.

Finally, in the results of all the sub-cases of (c) except case 2, we can simply replace each of $[Q_{i+1}.\mathcal{C}]_{\alpha=0}$, $[Q_{i+1}.\mathcal{C}]_{\alpha=1}$, $[Q_{i+1}.\mathcal{C}]_{\alpha=\exists z}$ and $[Q_{i+1}.\mathcal{C}]_{\alpha=\forall z}$ by $[Q_i.\mathcal{C}]_{\alpha=0}$, $[Q_i.\mathcal{C}]_{\alpha=1}$, $[Q_i.\mathcal{C}]_{\alpha=\exists z}$ and $[Q_i.\mathcal{C}]_{\alpha=\forall z}$, respectively, since $x_{i+1} \in faninPI^*(\alpha)$ will be removed from the prefix either way.

■

Corollary 5.2 *$Q.\mathcal{C}$ and $f_\alpha(Q.\mathcal{C})$ are logically equivalent.*

Proof: In Theorem 5.1, setting $i = 0$, we have $\pi_0 = \emptyset$. Hence $Q_0.\mathcal{C}|_\emptyset = Q.\mathcal{C}$ and $f_\alpha(Q_0.\mathcal{C}, \emptyset) = f_\alpha(Q.\mathcal{C})$ are logically equivalent.

■

The following example demonstrates the construction of $f_\alpha(Q.\mathcal{C})$.

Example 5.1 *Consider the circuit-based QBF given in Figure 5.4, where α is a complete dominator. Figure 5.5 illustrates the construction of the logically equivalent α-reduced QBF $f_\alpha(Q.\mathcal{C})$ in a bottom-up recursive fashion. Each node in the tree gives what α is set to in $f_\alpha(Q_i.\mathcal{C}, \pi_i)$, where Q_i is shown at the same level of the node and to the left of the tree, whereas π_i assigns all the literals shown on the edges of the path from the root to that node. For example, at $i = 2$, the third node from the left is 0. This means that $f_\alpha(Q_2.\mathcal{C}, \{x_1 = 1, x_2 = 0\}) = [Q_2.\mathcal{C}]_{\alpha=0} = [\exists x_3 \forall x_4.\mathcal{C}]_{\alpha=0}$.*

*The construction starts with the base cases (*i.e., *case (a) in Definition 5.1), occurring at the leaves of the tree in Figure 5.5, where $Q_4.\mathcal{C} = \mathcal{C}$ has only free variables. Each assignment π_4 corresponding to a different leaf, assigns the variables in $faninPI^*(\alpha) = \{x_1, x_2, x_3\}$, and $f_\alpha(\mathcal{C}, \pi_4)$ sets α to its simulated value under that π_4. For example, in the second leaf from*

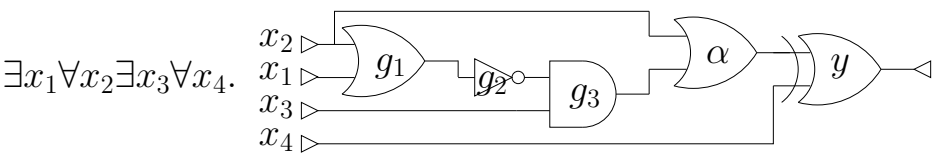


Figure 5.4: A circuit-based QBF with complete dominator α

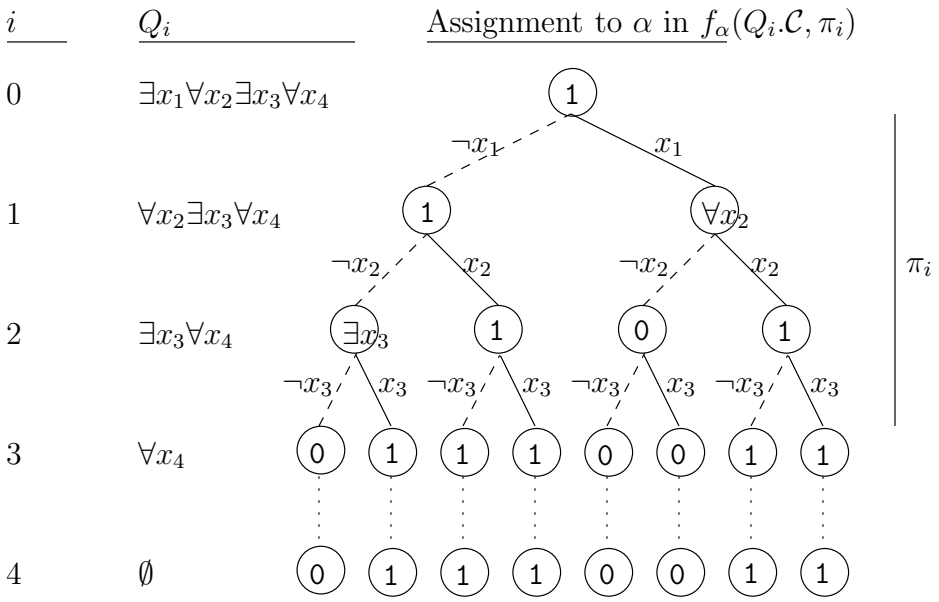


Figure 5.5: Constructing $f_\alpha(Q.\mathcal{C})$

the left, $\pi_4 = \{\neg x_1, \neg x_2, x_3\}$*, which yields* $\alpha = 1$*, and hence* $f_\alpha(\mathcal{C}, \{\neg x_1, \neg x_2, x_3\}) = [\mathcal{C}]_{\alpha=1}$ *(denoted as* 1 *in the figure for brevity). As such, the base cases are constructed by simulating* α *for all combinations of* $\{x_1, x_2, x_3\}$*.*

Once all $2^3 = 8$ *base cases are computed, we consider* $Q_3.\mathcal{C} = \forall x_4.\mathcal{C}$*. Since* $x_4 \notin faninPI^*(\alpha)$*, by case (b) of Definition 5.1, we get* $f_\alpha(Q_3.\mathcal{C}, \pi_3) = \forall x_4 f_\alpha(Q_4.\mathcal{C}, \pi_4)$*, where* $\pi_4 = \pi_3$*. So if* $f_\alpha(Q_4.\mathcal{C}, \pi_4) = [\mathcal{C}]_{\alpha=0}$ *(respectively,* $[\mathcal{C}]_{\alpha=1}$*), then* $f_\alpha(Q_3.\mathcal{C}, \pi_3) = \forall x_4.[\mathcal{C}]_{\alpha=0}$ *(respectively,* $\forall x_4.[\mathcal{C}]_{\alpha=1}$*), where* $\pi_4 = \pi_3$*. As such, the reduction at* α *remains the same at the level of variables in the prefix which are not in* $faninPI^*(\alpha)$*, such as* x_4*. The dotted lines above each leaf of the tree in Figure 5.5 demonstrate this.*

Next, each of $x_1, x_2, x_3 \in faninPI^*(\alpha)$*, and therefore they follow case (c) of Definition 5.1. Here, a node* $f_\alpha(Q_i.\mathcal{C}, \pi_i)$ *is a function of its two children* $f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_i = 0\})$ *and* $f_\alpha(Q_{i+1}.\mathcal{C}, \pi_i \cup \{x_i = 1\})$*. For instance, consider* $Q_1.\mathcal{C} = \forall x_2 \exists x_3 \forall x_4.\mathcal{C}$ *and* $\pi_1 = \{x_1\}$*, which corresponds to the right child of the root in Figure 5.5. We get* $f_\alpha(Q_1.\mathcal{C}, \{x_1 = 1\}) = f_\alpha(Q_2.\mathcal{C}, \{x_1 = 1, x_2 = 0\}) \wedge f_\alpha(Q_2.\mathcal{C}, \{x_1 = 1, x_2 = 1\}) = [Q_2.\mathcal{C}]_{\alpha=0} \wedge [Q_2.\mathcal{C}]_{\alpha=1} = [Q_1.\mathcal{C}]_{\alpha=\forall x_2}$*, by applying case* 2 *in Table 5.1. This is denoted as* $\forall x_2$ *in the figure for brevity.*

The final result is given at the root of the tree, where $Q_0.\mathcal{C} = Q.\mathcal{C}$*,* $\pi_0 = \emptyset$ *and* $f_\alpha(Q.\mathcal{C}, \emptyset) =$

$f_\alpha(Q.\mathcal{C}) = [Q.\mathcal{C}]_{\alpha=1}$. In other terms, we can set $\alpha =$ `1` and remove its fanin cone without affecting the value of the original QBF. Propagating $\alpha =$ `1` into the `XOR` gate yields a `NOT` gate, and the original QBF reduces to Figure 5.6. Clearly, this smaller QBF is much easier to solve.

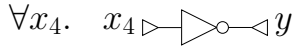


Figure 5.6: The α-reduced QBF of Figure 5.4

Notice that during the construction of $f_\alpha(Q.\mathcal{C})$, variables not in $faninPI^*(\alpha)$, such as x_4 in Example 5.1, can be disregarded in practice. As such, one only needs to examine the SODS of α and can disregard the remaining circuit when constructing $f_\alpha(Q.\mathcal{C})$.

It is also worth emphasizing that even though the number of simulations to compute $f_\alpha(Q.\mathcal{C})$ using a tree similar to that of Figure 5.5 is exponential in $|faninPI^*(\alpha)|$, reducing SODSes using the techniques given in this section can still provide massive run-time improvements. For instance, consider a circuit-based QBF with $n = 16$ primary inputs. If there are 4 sets of 4 primary inputs, with each set completely dominated by a separate node, then reducing all four SODSes would require $4 \cdot 2^4 = 64$ simulations, and solving the resulting QBF problem, which would have at most 4 inputs, would require an additional $2^4 = 16$ simulations in the worst-case. This results in at most $64 + 16 = 80$ simulations, whereas solving the original QBF could potentially require $2^{16} = 65\,536$ simulations. Furthermore, reductions of existing complete dominators can create new complete dominators, which can themselves be reduced iteratively. The `PReDom` algorithm described in the following section finds and reduces such SODSes iteratively.

5.4 The `PReDom` Algorithm

In this section, we describe our implementation of the preprocessor `PReDom`, which searches for complete dominators and applies the theory described in Section 5.3 to reduce their SODSes. In our implementation, the resulting circuit-based QBF is translated back to prenex normal form,

so that state-of-the-art CNF-based QBF solvers can take advantage of the reductions. In order to do so, `PReDom` also stores the CNF representation of the circuit-based problem. Actions such as eliminating a subcircuit and setting dominator outputs to constants are done by removing the clauses corresponding to the gates in that subcircuit and adding unit literals to the CNF formula, respectively.

Algorithm 5.1 gives a simplified view of the `PReDom` procedure. Before searching for complete dominators, `PReDom` performs several standard optimization steps, grouped in SIMPLIFY(Q,$\mathcal{C}$) (line 1). Unit constraint propagation and *universal reduction* [18] are performed directly in the CNF. *Equivalence reduction* is achieved efficiently by removing `NOT`s and `BUFFER`s in the circuit (and therefore their clauses in the CNF) and adding fanin polarity information for the remaining gates. Finally, dangling gates are also removed.

On line 2, $dom(v)$ is computed for every $v \in V$, where V is the set of nodes in $\mathcal{C}$. We use the method in [7] because it is simple and sufficiently fast in practice, but faster algorithms exist, e.g. [45]. Next, we compute the inverse map $dom^{-1}(v)$, for every $v \in V$, which stores the set of nodes dominated by v.

In the main loop of the preprocessor, each vertex v is traversed in topological order (line 5), and the set $faninPI^*(v)$ is computed recursively (lines 7 to 9). On line 10, the preprocessor checks that the number of primary inputs in the fanin cone of v is not more than a user-defined upper-bound MAX, which we have set to 20. This is needed to avoid memory explosion, since the number of base cases in the reduction is exponential in $|faninPI^*(v)|$, as shown in Figure 5.5.

Line 11 is the condition for v to be a complete dominator, which is reduced on line 12. Here PI denotes the set of primary inputs in $\mathcal{C}$. The function REDUCE uses parallel simulations to efficiently produce all the input combinations of $faninPI^*(v)$, and applies a scheme similar to the one shown in Example 5.1, to reduce the SODS of v. When a complete dominator is found and reduced, it is propagated and its $faninPI^*(v)$ is reset. The *ignore* flags are used to disregard nodes which are already known to have more than MAX primary inputs in their fanin cones.

Finally, it should be noted that `PReDom` first decomposes gates with more than two inputs

Algorithm 5.1: PReDom algorithm

input : Prefix Q, matrix $\mathcal{C}$

output: Reduced prefix Q, matrix $\mathcal{C}$

```
1  SIMPLIFY(Q,C);
2  dom← COMPUTEDOM(C);
3  dom^{-1}← COMPUTEINV(dom);
4  foreach v ∈ V do ignore[v] ← false;
5  foreach v ∈ V in topological order ∧ ¬ignore[v] do
6      if v ∈ PI then
7          faninPI*[v] ← {v} ;
8      else
9          faninPI*[v] ← ⋃_{u∈faninPI*[v]} faninPI*[u];
10     if |faninPI*[v]| < MAX then
11         if faninPI*[v] = dom^{-1}[v] ∩ PI then
12             v ← REDUCE(v,faninPI*[v],Q,C) ;
13             faninPI*[v] ← {v} ;
14     else
15         ignore[v] ← true ;
16         ∀_{u∈fanout[v]}.ignore[u] ← true ;
```

into several two-input gates. For example, a 4-input `AND` gate is decomposed into a cascade of three 2-input `AND` gates. This is beneficial because if any one of these 2-input `AND` gates is a complete dominator of its transitive fanin-cone, its SODS can be reduced even if the original 4-input `AND` gate is not a complete dominator itself.

5.5 Experimental Results

We implemented our circuit-based QBF preprocessor `PReDom` in C++. The input format of the preprocessor is composed of three parts: (1) A circuit description in ISCAS85 format using `NOT`, `BUFFER`, `AND`, `OR`, `NAND`, `NOR`, `XOR` and `XNOR` gates, (2) the corresponding QBF in prenex normal form, and (3) a file mapping between the circuit nodes and variables/clauses in the CNF. The preprocessor first applies standard optimizations (SIMPLIFY), then searches for complete dominators and reduces their SODSes, as described in this chapter. The resulting problems are then given to QBF solvers.

The benchmarks are a suite of circuit state-space diameter computation problems called *dme* [48] for a distributed mutual exclusion protocol from NuSMV [28]. State-space diameter computation is an important problem in formal verification which is required for the completeness of bounded model checking (BMC). The problems are originally given in the QBF1.0 format [46], and are converted in negligible time into our specified format using the converter of [53]. The results of three state-of-the-art QBF solvers, namely `sKizzo-v0.10` [13], `2clsQ` [103] and `quantor-v3.0` [18] (with the recommended `picosat` [19] back end) are compared with and without `PReDom`. All experiments are conducted on a Pentium IV 2.8 GHz Linux platform with 12 GB of memory and a time limit of 5 hours.

Table 5.2 shows the preprocessing results using `PReDom`. The first column gives the instance name. Column `#` *clause orig* gives the original number of clauses in the CNF of each problem instance. Columns `#` *clause+sim* and `#` *clause+dom* respectively show the number of clauses after the optimizations in SIMPLIFY and after preprocessing by `PReDom`. Column *% red orig-final* (respectively, *% red sim-final*) shows the percentage reduction in the number of clauses from the original (respectively, simplified) instance to the final preprocessed instance. Column `#` *SODS*

gives the total number of SODSes which were reduced. Column *dom (sec)* gives the run-time to compute the sets $dom(v), \forall v \in V$, using the algorithm from [7]. Finally, column *total (sec)* gives the total run-time of `PReDom` on each instance in seconds.

Notice that the average reduction in the number of clauses from the original to the final instance is 47%. This includes both standard optimizations in SIMPLIFY and dominator-based reductions. The average reduction in the number of clauses after SIMPLIFY is 19%. This number varies significantly across these benchmarks. Figure 5.7 illustrates the reduction percentages. For each instance, 100% represents all the original clauses. Each bar is partitioned into the clauses reduced by SIMPLIFY (top), then the dominator-based reductions (middle), and the final clauses (bottom). The high dominator-based reductions, *e.g.,* 53% for dme1_5 and 52% for dmeSmall_8, occur when a complete dominator with high fanout is replaced by a constant value, resulting in the elimination of even more circuitry using unit constraint propagation. This also explains the seeming inverse relationship between the number of reduced SODSes and the dominator-based reductions, because when a complete dominator is replaced by a constant, propagation already eliminates other complete dominators which might dominate the first one. Due to the reduction of SODSes in topological order in `PReDom`, the maximum size of $faninPI^*(\alpha)$ for a reduced complete dominator α was 3.

Looking at the last two columns of Table 5.2, we can see that computing the sets $dom(v)$ takes a significant portion of the preprocessing run-time. This is because we use a simple algorithm [7], which has an $O(|V|^2)$ worst-case time complexity. Once these sets are computed, the time for finding complete dominators and replacing their SODSes is small.

Table 5.3 shows the results of the application of the QBF solvers `sKizzo`, `2clsQ` and `quantor` on the instances after standard simplifications only (column *+sim*) and after complete preprocessing using `PReDom` (column *+dom*). The effect of SIMPLIFY by itself is minimal because these QBF solvers already apply equivalent simplifications at the CNF level. `sKizzo`, `2clsQ` and `quantor` time-out or mem-out on all instances where our techniques are not used to reduce SODSes, even with a time-out of 5 hours. On the other hand, they respectively solve 5/11, 3/11 and 5/11 of the instances after preprocessing using `PReDom`, collectively solving 55% of all instances, and the run-times are usually less than one second. Since none of the *+sim* runs

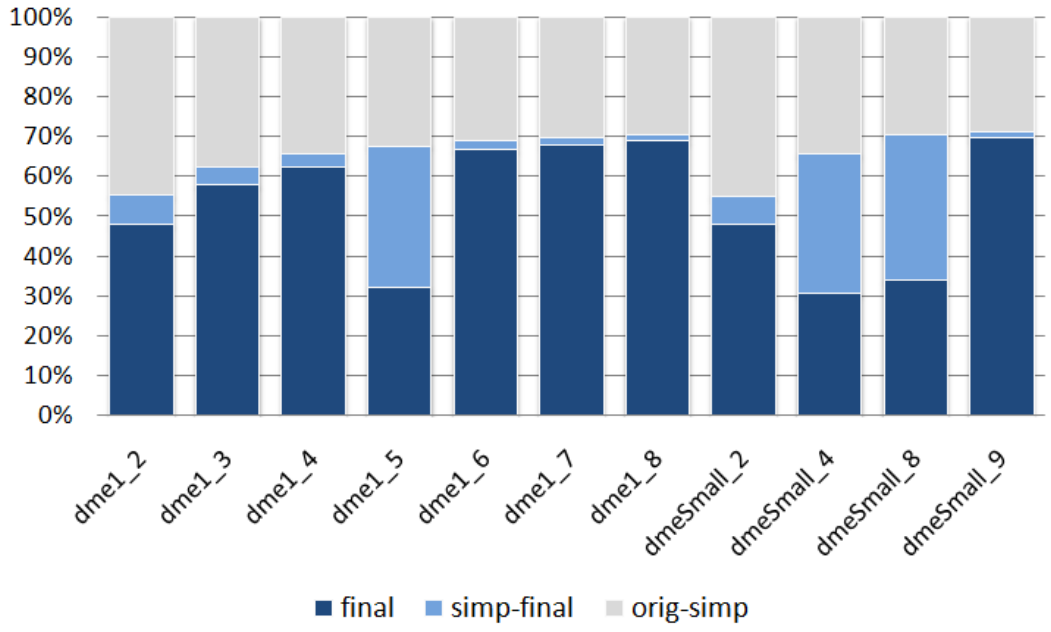


Figure 5.7: Clause reduction percentages

terminate, our results were validated using the circuit-based QBF solver CirQit [53].

It should be noted that many of the QBF problems in the non-prenex non-CNF track of the QBFEVAL'08 competition [46] did not have SODSes. The experiments show the benefits of reducing SODSes, if they exist. On the other hand, even if no SODSes exist, the preprocessing time is negligible. We also developed a circuit-based QBF solver which performs dominator-based reductions *on-the-fly*, rather than as a preprocessing step. However, the overhead was too large to result in run-time improvements. We discuss this in more detail in Chapter 7.

5.6 Summary

This chapter develops a theory for exploiting the circuit structure of a circuit-based QBF by leveraging complete dominators. A methodology and a rigorous proof are given for removing subcircuits that are dominated by single nodes in a circuit-based QBF. We present the QBF preprocessor `PReDom`, which applies the described theory to return smaller but logically equivalent QBF instances, in order to expedite the QBF solving process. Experimental results on circuit diameter computation problems show a significant increase in solved instances after pre-

processing. This work encourages further research in strategies that exploit the circuit structure of QBFs to increase performance.

Table 5.2: PReDom preprocessing results

Benchmark	# clause orig	PReDom # clause +sim	PReDom # clause +dom	PReDom % red orig-final	PReDom % red sim-final	PReDom # SODS	PReDom dom (sec)	PReDom total (sec)
dme1_2	20 984	11 565	10 068	52%	13%	125	0.9	1.2
dme1_3	32 888	20 488	18 991	42%	7%	125	1.4	1.8
dme1_4	44 792	29 411	27 914	38%	5%	125	1.9	2.5
dme1_5	56 696	38 334	18 173	68%	53%	76	2.4	3.2
dme1_6	68 600	47 257	45 760	33%	3%	125	2.9	3.9
dme1_7	80 504	56 180	54 683	32%	3%	125	3.4	4.6
dme1_8	92 408	65 103	63 606	31%	2%	125	4.0	5.3
dmeSmall_2	13 984	7 701	6 708	52%	13%	83	0.4	0.5
dmeSmall_4	29 856	19 595	9 136	69%	53%	73	0.9	1.1
dmeSmall_8	61 600	43 383	21 020	66%	52%	73	1.8	2.5
dmeSmall_9	69 536	49 330	48 337	30%	2%	83	2.0	2.8
average				**47%**	**19%**			

Table 5.3: QBF solver evaluation

Benchmark	`sKizzo` time (sec)		`2clsQ` time (sec)		`quantor` time (sec)	
	+sim	+dom	+sim	+dom	+sim	+dom
dme1_2	[mem-out]	[mem-out]	[time-out]	[time-out]	[mem-out]	[mem-out]
dme1_3	[mem-out]	**0.2**	[time-out]	[time-out]	[mem-out]	**0.2**
dme1_4	[mem-out]	[mem-out]	[time-out]	[time-out]	[mem-out]	**0.1**
dme1_5	[mem-out]	**0.5**	[time-out]	**0.6**	[mem-out]	**0.1**
dme1_6	[mem-out]	[mem-out]	[time-out]	[time-out]	[mem-out]	[mem-out]
dme1_7	[mem-out]	[mem-out]	[time-out]	[time-out]	[mem-out]	[mem-out]
dme1_8	[mem-out]	[mem-out]	[time-out]	[time-out]	[mem-out]	[mem-out]
dmeSmall_2	[mem-out]	**16.2**	[time-out]	[time-out]	[mem-out]	[mem-out]
dmeSmall_4	[mem-out]	**0.2**	[time-out]	**0.1**	[mem-out]	**0.1**
dmeSmall_8	[mem-out]	**0.6**	[time-out]	**0.7**	[mem-out]	**0.1**
dmeSmall_9	[mem-out]	[mem-out]	[time-out]	[time-out]	[mem-out]	[mem-out]
summary	**0/11**	**5/11**	**0/11**	**3/11**	**0/11**	**5/11**

Chapter 6

Reconfigurability in Partially Programmable Circuits

6.1 Introduction

Partially programmable circuits (PPCs) [120] are obtained from conventional combinational logic circuits by replacing some subcircuits with reconfigurable elements such as Look-Up Tables (LUTs) and configurable multiplexers (MUXs). Their original intention was to bypass faults post-silicon via reconfigurations. In this chapter, we examine the power of reconfigurability in PPCs for several applications. In addition to correcting manufacturing faults, the programmable bits in a PPC can be used to mask some localized design errors that escape verification and propagate into the silicon. Furthermore, we investigate the use of PPCs for implementing *engineering change orders* (ECOs), namely small changes in the specification at later stages of the design cycle. It is well-known that even minor ECOs can lead to vastly different synthesized implementations [114] if a new iteration of the automated flow is used. This is usually unwanted because of the effort already invested in optimizing the original design [22, 114]. As such, synthesis for ECOs strives to make the smallest number of changes to the implementation [22, 66, 114] so that the design complies to its new specification. In a PPC, LUT/MUX reconfigurations can be used to implement such changes at virtually zero cost, avoiding time-consuming design iterations.

Along these observations, the contribution here is multi-fold [81]. We define the *fault tolerance* of a PPC as the percentage of stuck-at-faults that can be made unobservable using post-silicon reconfigurations. *Design error tolerance* is defined in a similar fashion. We show how to compute both of these metrics using formal techniques. Following these contributions, we present a new method for performing ECOs in PPCs using reconfigurations. Finally, we define a measure for quantifying the effectiveness of a PPC in implementing ECOs, given an initial specification. We refer to this as the *ECO coverage* of a PPC architecture and we develop a methodology to compute it. We use QBFs as the underlying language for our encodings. Our formulations demonstrate the theoretical appropriateness of QBFs for dealing with reconfigurability and we capitalize on the considerable advances in QBF solvers in recent years.

It should be noted that this work does not attempt to construct PPCs that maximize ECO coverage or error tolerance. Instead, it lays the theoretical groundwork for calculating these quantities, as well as for performing ECOs. The existence of such evaluation tools is a first step in the generation of optimized PPCs. As such, the work here remains orthogonal and complementary to that in [120] which is strictly focused on constructing PPCs using heuristics. Experimental results are presented evaluating PPCs from [120], demonstrating the applicability and accuracy of the proposed QBF formulations.

This chapter is organized as follows. Section 6.2 contains preliminaries on PPCs. Section 6.3 presents our formulations for calculating fault and design error tolerance. Section 6.4 gives QBF encodings for performing ECOs and quantifying ECO coverage. Section 6.5 shows experimental results and Section 6.6 provides a summary of the chapter.

6.2 Preliminaries

The following notation is used throughout the chapter. We use the symbol $\mathcal{C}$ to denote a conventional combinational circuit, and $\hat{\mathcal{C}}$ to denote the corresponding PPC. The sets $\mathbf{x} = \{x_1, x_2, \ldots, x_{|\mathbf{x}|}\}$, $\mathbf{y} = \{y_1, y_2, \ldots, y_{|\mathbf{y}|}\}$ and $\mathbf{g} = \{g_1, g_2, \ldots, g_{|\mathbf{g}|}\}$ respectively refer to the sets of primary inputs, primary outputs and gates of $\mathcal{C}$. A *node* v can refer to a gate or a primary input. The sets $fanout(v)$ and $fanin(v)$ denote the fanout and fanin nodes of v, respectively. The set

$\mathbf{l} = \{(u, v) \mid u, v \in \mathbf{x} \cup \mathbf{g}$ and $v \in fanout(u)\}$ contains all *lines* (also referred to as *connections* or *branches*) in $\mathcal{C}$. For each $\mathbf{z} \in \{\mathbf{x}, \mathbf{y}, \mathbf{g}, \mathbf{l}\}$, $\hat{\mathbf{z}} = \{\hat{z}_1, \hat{z}_2, \ldots, \hat{z}_{|\hat{\mathbf{z}}|}\}$ denotes the corresponding set in $\hat{\mathcal{C}}$. Throughout the chapter, bold ($\mathbf{z}$) versus regular (z) symbols differentiate sets from single variables, and a hat ($\hat{z}$ versus z) differentiates between variables in $\hat{\mathcal{C}}$ and $\mathcal{C}$, respectively.

6.2.1 Partially Programmable Circuits

The type of a node v is given by $type(v) \in \{\texttt{IN}, \texttt{AND}, \texttt{OR}, \ldots, \texttt{LUT}, \texttt{MUX}\}$. A PPC $\hat{\mathcal{C}}$ is a Boolean network with three types of nodes [120]:

- Conventional logic gates, such as `AND`, `OR`, `NOT` and `XOR`.
- `LUT`s, whose internal functionality can be reconfigured.
- `MUX`s, whose select lines are controlled by programmable memory cells.

To simplify the presentation, we assume that the original circuit $\mathcal{C}$ does not contain `LUT`s/`MUX`s. Note that a `LUT` can itself be represented as a multiplexer with configurable data inputs. As such, the *configuration bits* of a `LUT` $\hat{g}_i$ are the set of Boolean variables:

$$\hat{\mathbf{c}}(\hat{g}_i) = \{\hat{c}_j(\hat{g}_i) \mid j = 1, \ldots, 2^n\},$$

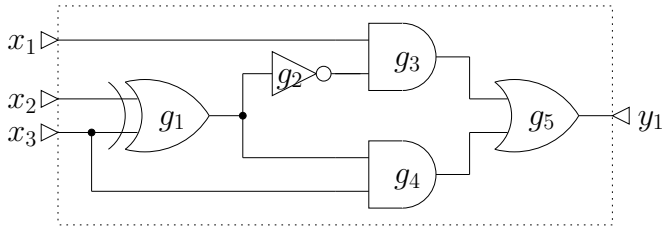
where n denotes the number of input select lines of the `LUT`. On the other hand, the configuration bits of a configurable `MUX` $\hat{g}_i$ in a PPC are its select lines. They are given by:

$$\hat{\mathbf{c}}(\hat{g}_i) = \{\hat{c}_j(\hat{g}_i) \mid j = 1, \ldots, \lceil \log_2 n \rceil\},$$

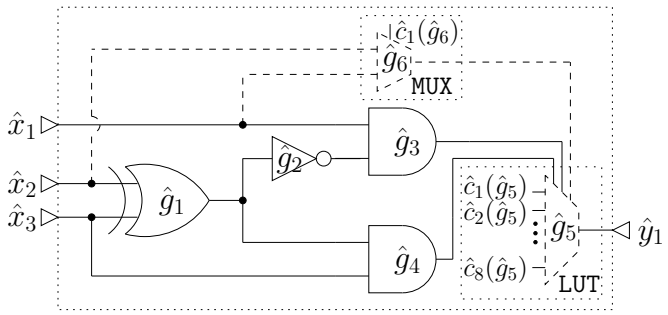
where n denotes the number of data inputs of the configurable `MUX`.

Figures 6.1(a) and 6.1(b) show a combinational circuit $\mathcal{C}$ and a corresponding PPC $\hat{\mathcal{C}}$. Note that y_1 (respectively, $\hat{y}_1$) is the primary output label for g_5 (respectively, $\hat{g}_5$) and does not represent a separate node. In Figure 6.1(b), variables $\hat{c}_1(\hat{g}_6)$ and $\hat{c}_j(\hat{g}_5)$ $(j = 1, \ldots, 8)$ are the configuration bits of $\hat{g}_6$ and $\hat{g}_5$, respectively.

In [120], PPCs are constructed as follows. First, given an original circuit $\mathcal{C}$, an initial PPC is generated by replacing certain subcircuits of $\mathcal{C}$ with `LUT`s using simple heuristics. Next, redundant lines are added from selected nodes to some of these `LUT`s in an effort to increase the



(a) $\mathcal{C}$



(b) $\hat{\mathcal{C}}$

Figure 6.1: A circuit and its corresponding PPC

number of so-called *robust* connections in the PPC. A robust connection is a line where a stuck-at-0 and a stuck-at-1 can be made unobservable by reprogramming the PPC post-silicon. These added redundant lines are selected as follows. For each line (u, v), a set of new connections are added to the LUT inputs such that the *functional flexibilities* of the LUTs, represented by their *sets of pairs of functions to be distinguished* (SPFDs), allow them to be reconfigured to bypass stuck-at-faults at (u, v). Of course, this is not always possible given limited resources, so not all lines can be made robust. If more than one redundant line needs to be added to a certain LUT, a configurable MUX is placed in front of the LUT, which selects between these redundant lines. We are concerned with evaluating rather than constructing PPCs, hence the reader is referred to [120] for more details on their algorithms. Our techniques for evaluating PPCs can be applied to any PPC.

In the PPC shown in Figure 6.1(b), gate g_5 is replaced by a LUT $\hat{g}_5$. Of course, $\hat{g}_5$ can be easily programmed to implement $\texttt{OR}(\hat{g}_3, \hat{g}_4)$. Next, we have added redundant connections (shown using dashed lines) from $\hat{x}_1$ and $\hat{x}_2$ to a MUX $\hat{g}_6$, which is input to the LUT $\hat{g}_5$. In the

coming sections, we present QBF formulations that can show that this PPC structure has 100% single stuck-at-fault tolerance (disregarding stuck-at-faults at the primary output), 100% design error tolerance (assuming single gate arbitrary errors) and 100% ECO coverage (using $\mathcal{C}$ as the initial specification and our ECO coverage definition).

6.3 Fault and Design Error Tolerance

In this section, we first construct a QBF formulation for calculating the stuck-at-fault tolerance of a PPC. We use stuck-at-faults because this type of fault can model many defects [58]. Then, we extend this formulation to calculate the gate design error tolerance of a PPC. Finally, for single stuck-at-fault tolerance and single gate design error tolerance, we show how to partition our formulations into a linear number of smaller parallelizable problems in order to achieve faster QBF solving times by taking advantage of modern multi-core architectures.

6.3.1 Fault Tolerance

Given a specification $\mathcal{C}$, and a corresponding implementation in the form of a PPC $\hat{\mathcal{C}}$ *with a fixed configuration*, we say that a stuck-at-fault (or a design error) in $\hat{\mathcal{C}}$ is *unobservable* if there does not exist any primary input vector for which $\hat{\mathcal{C}}$ and $\mathcal{C}$ produce different primary outputs. This can be extended to N stuck-at-faults, where N denotes the cardinality of simultaneous stuck-at-faults. In what follows, we use the term *N-fault* to denote N simultaneous stuck-at-faults.

Definition 6.1 *Given a specification $\mathcal{C}$, a PPC $\hat{\mathcal{C}}$ and a stuck-at-fault cardinality N, the* fault tolerance *of $\hat{\mathcal{C}}$ is the percentage of N-faults that can be made unobservable using reconfigurations.*

Using $N = 1$ for illustration purposes, We emphasize that different single stuck-at-faults are allowed to be made unobservable by *different* PPC reconfigurations. The goal is that in silicon, if a stuck-at-fault is detected during testing, we would like to be able to reprogram the PPC to "mask" it. In general, if for a given N-fault there exists a PPC reconfiguration making it unobservable, this N-fault counts towards the fault tolerance of the PPC. Again, reconfigurations can vary for different N-faults. Clearly, a high fault tolerance increases manufacturing yield

because faults that otherwise would make the circuit unusable can now be made unobservable by reconfiguring the PPC LUTs/MUXs.

The key idea is to build a QBF instance where each Q-model corresponds to an N-fault that cannot be made unobservable by any reconfiguration of $\hat{\mathcal{C}}$. In what follows, we explain how to create the matrix of our QBF formulation using an appropriate circuit construction. In order to assist the reader in visualizing our descriptions, Figure 6.2 illustrates this construction (which is described shortly) for $\mathcal{C}$ and $\hat{\mathcal{C}}$ given in Figures 6.1(a) and 6.1(b).

We first create an enhanced version of $\hat{\mathcal{C}}$, which we call $\hat{\mathcal{C}}_{saf}$. To prevent any confusion, we stress that any enhancements to $\hat{\mathcal{C}}_{saf}$ are only added to construct our QBF formulation. We do not modify the actual PPC $\hat{\mathcal{C}}$ in any way. We start by adding a special multiplexer in front of each gate, each line and each primary input, which determines whether or not a stuck-at-fault is excited at that gate, line or primary input. Note that gate and line stuck-at-faults in this context correspond to *stem* and *branch* stuck-at-faults [58], respectively. Of course, if a gate has only one fanout, we do not double-count by adding two multiplexers at its output. The shaded multiplexers in Figure 6.2 illustrate this process for gate $\hat{g}_1$, line $(\hat{g}_1, \hat{g}_4)$ and primary input $\hat{x}_1$. We do not show the multiplexers for the remaining gates, lines and primary inputs to avoid overcrowding that figure. The select-line of each of these multiplexers is called an *excitation* variable, denoted by the letter $\hat{e}$.

In more detail, at each gate $\hat{g}_i$ (respectively, each line $(\hat{u}, \hat{v})$ and each primary input $\hat{x}_j$), setting $\hat{e}(\hat{g}_i) = \mathtt{1}$ (respectively, $\hat{e}(\hat{u}, \hat{v}) = \mathtt{1}$ and $\hat{e}(\hat{x}_j) = \mathtt{1}$) "excites" the stuck-at-fault, by disconnecting $\hat{g}_i$ (respectively, $(\hat{u}, \hat{v})$ and $\hat{x}_j$) from its fan-ins, and instead connecting it to a newly created variable $\hat{w}(\hat{g}_i)$ (respectively, $\hat{w}(\hat{u}, \hat{v})$ and $\hat{w}(\hat{x}_j)$), which we call a *replacement* variable. As will be seen later, these $\hat{w}$'s will denote the polarities of the stuck-at-faults. On the other hand, setting $\hat{e} = \mathtt{0}$ keeps the gate/line/primary input unchanged, as can be seen in Figure 6.2.

Next, we apply common primary inputs ($\mathbf{x}$) to both $\mathcal{C}$ and $\hat{\mathcal{C}}_{saf}$, as shown in Figure 6.2. Furthermore, at least one primary output is forced to be different. Finally, a *cardinality constraint* Φ_N is added to force the number of simultaneously active (*i.e.*, assigned to $\mathtt{1}$) excitation variables to a pre-specified constant N. This can be done using a bitonic sorter [11]. This

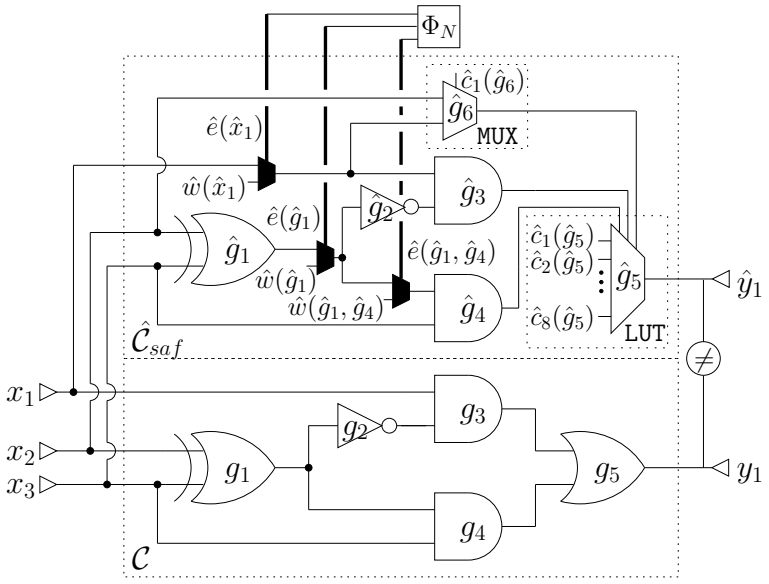


Figure 6.2: Stuck-at-fault tolerance matrix

completes the matrix of our QBF formulation.

In order to abbreviate the prefix of our QBF, as well as the remaining QBFs in this chapter, we use the following notation:

$$\begin{aligned}
\hat{\mathbf{e}}(\hat{\mathbf{g}}) &= \{\hat{e}(\hat{g}_i) \mid \forall \hat{g}_i \in \hat{\mathbf{g}}\} & \hat{\mathbf{w}}(\hat{\mathbf{g}}) &= \{\hat{w}(\hat{g}_i) \mid \forall \hat{g}_i \in \hat{\mathbf{g}}\} \\
\hat{\mathbf{e}}(\hat{\mathbf{l}}) &= \{\hat{e}(\hat{u}, \hat{v}) \mid \forall (\hat{u}, \hat{v}) \in \hat{\mathbf{l}}\} & \hat{\mathbf{w}}(\hat{\mathbf{l}}) &= \{\hat{w}(\hat{u}, \hat{v}) \mid \forall (\hat{u}, \hat{v}) \in \hat{\mathbf{l}}\} \\
\hat{\mathbf{e}}(\hat{\mathbf{x}}) &= \{\hat{e}(\hat{x}_i) \mid \forall \hat{x}_i \in \hat{\mathbf{x}}\} & \hat{\mathbf{w}}(\hat{\mathbf{x}}) &= \{\hat{w}(\hat{x}_i) \mid \forall \hat{x}_i \in \hat{\mathbf{x}}\}
\end{aligned} \tag{6.1}$$

And the sets of all excitation and replacement variables are respectively given by:

$$\hat{\mathbf{e}} = \hat{\mathbf{e}}(\hat{\mathbf{g}}) \cup \hat{\mathbf{e}}(\hat{\mathbf{l}}) \cup \hat{\mathbf{e}}(\hat{\mathbf{x}}) \qquad \hat{\mathbf{w}} = \hat{\mathbf{w}}(\hat{\mathbf{g}}) \cup \hat{\mathbf{w}}(\hat{\mathbf{l}}) \cup \hat{\mathbf{w}}(\hat{\mathbf{x}}) \tag{6.2}$$

When the context of the type of excitation/replacement variable is clear, we just use the symbols $\hat{e} \in \hat{\mathbf{e}}$ and $\hat{w} \in \hat{\mathbf{w}}$ for brevity.

Recall that the set $\hat{\mathbf{c}}(\hat{g}_i)$ refers to the configuration bits of the `LUT`/`MUX` $\hat{g}_i$. Let:

$$\hat{\mathbf{c}} = \bigcup_{\substack{\hat{g}_i \in \hat{\mathbf{g}}, \\ type(\hat{g}_i) \in \{\texttt{LUT}, \texttt{MUX}\}}} \hat{\mathbf{c}}(\hat{g}_i)$$

denote the set of all configuration bits in $\hat{\mathcal{C}}$.

Informally, the QBF problem can be stated as follows:

> *Is it possible to assign exactly N excitation variables in $\hat{\mathbf{e}}$ to* `1`*, and set what each corresponding gate/line/primary input is "stuck-at" (by assigning $\hat{\mathbf{w}}$), such that for all configurations of the PPC (assignments to $\hat{\mathbf{c}}$), there exists a primary input vector satisfying the constraints in Figure 6.2?*

This question can be formalized as:

$$\exists \hat{\mathbf{e}}, \hat{\mathbf{w}}\ \forall \hat{\mathbf{c}}\ \exists \mathbf{x}, \mathbf{g}, \hat{\mathbf{g}}\ .\ \mathcal{C}(\mathbf{x}, \mathbf{y}, \mathbf{g}) \wedge \hat{\mathcal{C}}_{saf}(\mathbf{x}, \hat{\mathbf{y}}, \hat{\mathbf{g}}, \hat{\mathbf{c}}, \hat{\mathbf{e}}, \hat{\mathbf{w}}) \wedge (\mathbf{y} \neq \hat{\mathbf{y}}) \wedge \Phi_N(\hat{\mathbf{e}}) \tag{6.3}$$

Notice that the placement of $\hat{\mathbf{w}}$ in the widest existential scope forces their assignment before the assignment of primary inputs, producing the semantics of stuck-at-faults. Adding constraints on these $\hat{w}$'s or moving them in the prefix can result in different error models, as will be seen shortly. If (6.3) is false, then every N-fault can be made unobservable (*i.e.*, is "maskable") by a reconfiguration of the PPC.

In order to count the number of maskable (or unmaskable) stuck-at-faults using (6.3), we need to add another term to the matrix in (6.3). In fact, notice that if a certain excitation variable $\hat{e}$ is not active, its corresponding $\hat{w}$ can simply be "grounded" to `0`, since its value does not propagate through the multiplexer. As such, we add the following constraints to (6.3):

$$\bigwedge_{\hat{e} \in \hat{\mathbf{e}}} (\neg \hat{e} \rightarrow \neg \hat{w}) \tag{6.4}$$

Adding (6.4) prunes the search-space of the QBF solver, such that in any Q-model of (6.3), the $\hat{w}$'s corresponding to the inactive $\hat{e}$'s are assigned to `0`. As such, two Q-models of this QBF that differ in their truth assignments to the widest existential scope $(\hat{\mathbf{e}}, \hat{\mathbf{w}})$ will correspond to two different N-faults that cannot be fixed by the PPC. Therefore, finding all distinct truth assignments to $\hat{\mathbf{e}}, \hat{\mathbf{w}}$ that satisfy (6.3) (*i.e.*, that can be extended to Q-models of (6.3)) is equivalent to finding all unmaskable N-faults. This can be done using a QBF solver, by blocking the assignment to $\hat{\mathbf{e}}, \hat{\mathbf{w}}$ in the returned Q-model using a blocking clause and re-solving (6.3) iteratively until the problem becomes false. Subtracting the number of such solutions from the total number of N-fault combinations, and dividing the result by this number gives the stuck-at-fault tolerance of the PPC for cardinality N.

6.3.2 Design Error Tolerance

In this subsection, we propose a QBF formulation to quantify the effectiveness of a PPC in masking localized design errors that escape verification and slip into the silicon. Our design error model consists of *any* functional modification in the function of a gate. We use the term *N-gates* to denote a set of N gates.

Definition 6.2 *Given a specification $\mathcal{C}$, a PPC $\hat{\mathcal{C}}$ and a gate design error cardinality N, the* design error tolerance *of $\hat{\mathcal{C}}$ is the percentage of N-gates where* any *simultaneous modifications can be made unobservable using reconfigurations.*

For instance, if $N = 1$, the design error tolerance is equal to the percentage of individual gates where any design error can be masked by a reconfiguration. In other terms, gates where at least one type of design error cannot be masked by any reconfiguration do not contribute to the design error tolerance. In the event that a design error is identified post-silicon, a PPC with high design error tolerance is likely to offer a configuration fix, allowing the circuit to operate correctly without the need for a costly respin.

In this subsection, we modify the QBF in (6.3) to deal with gate design errors. We model design errors by again enhancing $\hat{\mathcal{C}}$. Here, $\hat{\mathcal{C}}_{de}$ adds similar multiplexers as in Figure 6.2 but now only at the outputs of gates. Furthermore, the $\hat{w}(\hat{g}_i)$'s are no longer unconstrained and instead are the outputs of newly added `LUT`s whose select lines are $\hat{g}_i$'s inputs. This allows each $\hat{w}(\hat{g}_i)$ to be any function of the inputs of $\hat{g}_i$, thus implementing any gate design error. This construction is illustrated in Figure 6.3, where shaded multiplexers are added for gates $\hat{g}_1$ and $\hat{g}_2$. For each gate, $\hat{g}_i$, the set:

$$\hat{\mathbf{d}}(\hat{g}_i) = \{\hat{d}_j(\hat{g}_i) \mid j = 1, \ldots, 2^{|fanin(\hat{g}_i)|}\}$$

refers to the configuration bits of the *replacement* `LUT` $\hat{w}(\hat{g}_i)$.

Again, applying common primary inputs, forcing different primary outputs and adding cardinality constraints yields the matrix in Figure 6.3. Using this, our QBF formulation is

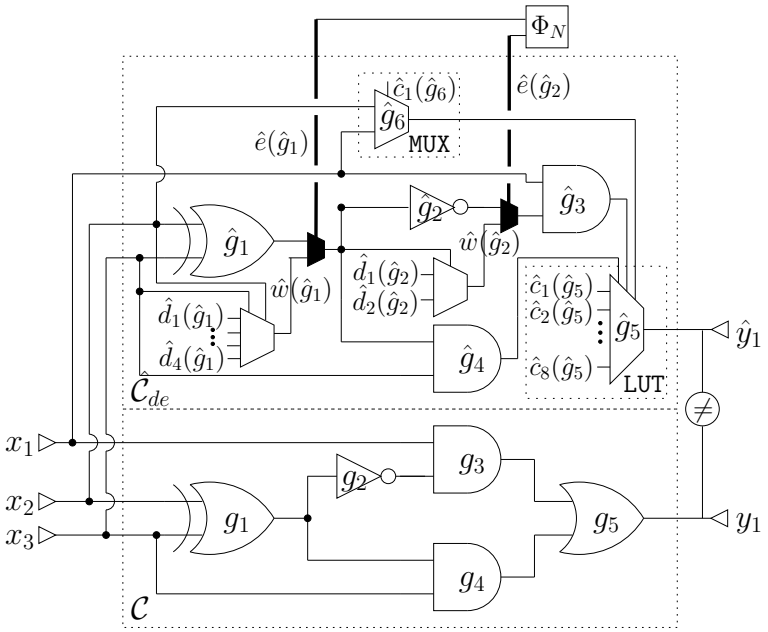


Figure 6.3: Gate design error tolerance matrix

given as follows:

$$\exists \hat{\mathbf{e}}(\hat{\mathbf{g}}), \hat{\mathbf{d}}(\hat{\mathbf{g}})\ \forall \hat{\mathbf{c}}\ \exists \mathbf{x}, \mathbf{g}, \hat{\mathbf{g}}, \hat{\mathbf{w}}(\hat{\mathbf{g}})\ .$$
$$\mathcal{C}(\mathbf{x}, \mathbf{y}, \mathbf{g}) \wedge \hat{\mathcal{C}}_{de}(\mathbf{x}, \hat{\mathbf{y}}, \hat{\mathbf{g}}, \hat{\mathbf{c}}, \hat{\mathbf{e}}(\hat{\mathbf{g}}), \hat{\mathbf{w}}(\hat{\mathbf{g}}), \hat{\mathbf{d}}(\hat{\mathbf{g}})) \wedge (\mathbf{y} \neq \hat{\mathbf{y}}) \wedge \Phi_N(\hat{\mathbf{e}}) \tag{6.5}$$

which asks whether there exist N-gates that can be arbitrarily modified such that for all PPC configurations ($\hat{\mathbf{c}}$), there is always an input vector exhibiting the error at a primary output. Similarly to (6.4), we add the following constraints that ground the configuration bits $\hat{\mathbf{d}}(\hat{g}_i)$ of $\hat{w}(\hat{g}_i)$ for gates whose excitation variables are inactive:

$$\bigwedge_{\hat{e}(\hat{g}_i) \in \hat{\mathbf{e}}(\hat{\mathbf{g}})} \left(\neg \hat{e}(\hat{g}_i) \rightarrow \left(\bigwedge_{\hat{d}_j(\hat{g}_i) \in \hat{\mathbf{d}}(\hat{g}_i)} \neg \hat{d}_j(\hat{g}_i) \right) \right) \tag{6.6}$$

This is done in order to create a one-to-one correspondence between different satisfying assignments to $\hat{\mathbf{e}}(\hat{\mathbf{g}}), \hat{\mathbf{d}}(\hat{\mathbf{g}})$ and different N-gate design errors that cannot be masked by the PPC. Finding all these satisfying assignments using blocking clauses enables us to calculate the gate design error tolerance of the PPC for cardinality N.

6.3.3 Problem Partitioning

Most often, we are interested in calculating single stuck-at-fault tolerance and single gate design error tolerance. It is usually difficult to mask multiple simultaneous faults or errors, especially with limited redundancy as in PPCs. Here, we show that when $N = 1$, we can partition the QBF problem (for both (6.3) and (6.5)) into a linear number of independently solvable and much easier subproblems, in order to take advantage of the modern multi-core architectures in solving these QBF instances. Note that our partitioning scheme is also applicable to higher cardinalities, however the number of independent subproblems increases exponentially with N.

For single stuck-at-fault tolerance, the partitioning is done by enumerating each $\hat{e} \in \hat{\mathbf{e}}$ and the corresponding two polarities of $\hat{w}$. For each gate/line/primary input with excitation variable $\hat{e}^*$ and replacement variable $\hat{w}^*$, and each stuck-at value $b \in \{0, 1\}$, we let:

$$\hat{\mathcal{C}}_{saf}|_{\hat{e}^*,\hat{w}^*=b} \triangleq \hat{\mathcal{C}}_{saf} \wedge \hat{e}^* \wedge (\hat{w}^* = b) \wedge \bigwedge_{\hat{e} \in \hat{\mathbf{e}} - \{\hat{e}^*\}} (\neg \hat{e}) \tag{6.7}$$

denote the PPC with only that gate/line/primary input stuck-at-b. We now ask whether there exists a PPC configuration, such that for all primary inputs, this faulty circuit produces the same outputs as $\mathcal{C}$. Formally, this is stated as:

$$\exists \hat{\mathbf{c}}\ \forall \mathbf{x}\ \exists \mathbf{g}, \hat{\mathbf{g}}\ .\ \mathcal{C}(\mathbf{x}, \mathbf{y}, \mathbf{g}) \wedge \hat{\mathcal{C}}_{saf}(\mathbf{x}, \hat{\mathbf{y}}, \hat{\mathbf{g}}, \hat{\mathbf{c}}, \hat{\mathbf{e}}, \hat{\mathbf{w}})|_{\hat{e}^*,\hat{w}^*=b} \wedge (\mathbf{y} = \hat{\mathbf{y}}) \tag{6.8}$$

Note that the cardinality constraints are no longer necessary because $\hat{\mathbf{e}}$ is already assigned a-priori, and all the inactive shaded multiplexers in Figure 6.2 can be discarded due to (6.7). Now although (6.8) must be solved for every single stuck-at-fault, each of these QBF instances is completely independent and much easier to solve than (6.3). As such, the number of maskable single stuck-at-faults can be computed by heavily parallelizing all the QBFs of the form (6.8) and simply counting the number of true results.

A similar partitioning can be accomplished for the single gate design error tolerance formulation in (6.5). Here, for each gate $\hat{g}_i$, we let:

$$\hat{\mathcal{C}}_{de}|_{\hat{e}(\hat{g}_i)} \triangleq \hat{\mathcal{C}}_{de} \wedge \hat{e}(\hat{g}_i) \wedge \bigwedge_{\hat{e}(\hat{g}_j) \in \hat{\mathbf{e}}(\hat{\mathbf{g}}) - \{\hat{e}(\hat{g}_i)\}} (\neg \hat{e}(\hat{g}_j)) \tag{6.9}$$

denote the PPC where only $\hat{g}_i$ can have a design error. We now ask whether for all possible design errors at $\hat{g}_i$, there exists a PPC configuration that masks the error. Formally,

$$\forall \hat{\mathbf{d}}(\hat{g}_i)\ \exists \hat{\mathbf{c}}\ \forall \mathbf{x}\ \exists \mathbf{g}, \hat{\mathbf{g}}, \hat{w}(\hat{g}_i)\ .$$
$$\mathcal{C}(\mathbf{x}, \mathbf{y}, \mathbf{g}) \wedge \hat{\mathcal{C}}_{de}(\mathbf{x}, \hat{\mathbf{y}}, \hat{\mathbf{g}}, \hat{\mathbf{c}}, \hat{\mathbf{e}}(\hat{\mathbf{g}}), \hat{\mathbf{w}}(\hat{\mathbf{g}}), \hat{\mathbf{d}}(\hat{\mathbf{g}}))|_{\hat{e}(\hat{g}_i)} \wedge (\mathbf{y} = \hat{\mathbf{y}}) \tag{6.10}$$

In each QBF of the form of (6.10), all $\hat{\mathbf{d}}(\hat{g}_j)$ and $\hat{w}(\hat{g}_j)$ with $j \neq i$ can be disregarded, since they cannot propagate through the shaded multiplexers in Figure 6.3. Again, for each gate, a QBF of the form of (6.10) must be solved to determine whether all possible errors at that gate can be masked by the PPC. All these QBFs can be solved in parallel. The single gate design error tolerance of the PPC is equal to the ratio of these QBFs that are true.

6.4 Engineering Change Order

In this section, we first construct a QBF for performing an ECO using a PPC. Then, we define the ECO coverage of a PPC and show how to compute it using a QBF.

6.4.1 Performing ECOs

ECOs are small changes in the specification at later stages of the design cycle. Synthesis for ECOs strives to make the smallest number of changes to the implementation [22, 66, 114]. PPCs can be used to implement ECOs pre- or post-silicon by simply reprogramming the `MUX`s/`LUT`s.

Given a modified specification $\mathcal{C}_{mod}$, if there exists a configuration of the PPC $\hat{\mathcal{C}}$, such that for all primary inputs, $\hat{\mathcal{C}}$ and $\mathcal{C}_{mod}$ behave identically, then the ECO can be implemented by reprogramming the PPC. This is easily expressed as the following QBF:

$$\exists \hat{\mathbf{c}}\ \forall \mathbf{x}\ \exists \mathbf{g}, \hat{\mathbf{g}}\ .\ \mathcal{C}_{mod}(\mathbf{x}, \mathbf{y}, \mathbf{g}) \wedge \hat{\mathcal{C}}(\mathbf{x}, \hat{\mathbf{y}}, \hat{\mathbf{g}}, \hat{\mathbf{c}}) \wedge (\mathbf{y} = \hat{\mathbf{y}}) \tag{6.11}$$

Figure 6.4 illustrates the matrix of (6.11) given a specification $\mathcal{C}_{mod}$ where the `NOT` gate g_2 has been eliminated and $g_4 = \texttt{AND}(x_3, g_1)$ has been replaced by $g_4 = \texttt{NAND}(x_1, g_1)$. Using a QBF solver, it can be easily verified that the QBF (6.11) with the matrix shown in Figure 6.4 is true. The satisfying assignment to the configuration bits $\hat{\mathbf{c}}$ returned by the solver can be used to reprogram the PPC to implement the modified specification at essentially zero-cost.

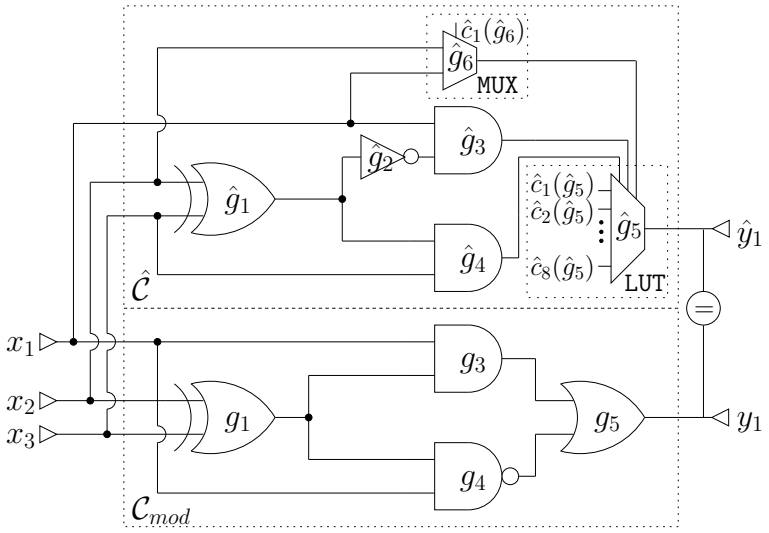


Figure 6.4: Engineering change matrix

Interestingly, the QBF in (6.11) is similar to a formulation used for FPGA technology mapping given in [72].

6.4.2 ECO Coverage

Given a PPC $\hat{\mathcal{C}}$ and an original specification $\mathcal{C}$, we would like to measure the effectiveness of this PPC architecture in implementing small changes in $\mathcal{C}$. Given a change cardinality N, a simple way to model small changes in the specification netlist $\mathcal{C}$ is to allow N gates to be changed *arbitrarily*. As such, we define the *ECO coverage* of a PPC as follows:

Definition 6.3 *Given an original specification $\mathcal{C}$, a PPC $\hat{\mathcal{C}}$ and a change cardinality N, the* ECO coverage *of $\hat{\mathcal{C}}$ is the percentage of N-gates in $\mathcal{C}$, where* any *simultaneous modifications can be implemented using reconfigurations in $\hat{\mathcal{C}}$.*

Note that many ECOs involve changes at a higher abstraction level, for which different models should be considered. Furthermore, since this chapter deals with combinational PPCs, sequential specification changes are not covered. Our formulation for ECO coverage is essentially the dual of the formulation for design error tolerance given in (6.5). Here, we must enhance the specification circuit $\mathcal{C}$ instead of $\hat{\mathcal{C}}$, since we are allowing the specification to change. A multi-

plexer is added at the output of each gate g_i in $\mathcal{C}$, with *excitation* select line $e(g_i)$. Furthermore, similarly to Figure 6.3, the $w(g_i)$'s are the outputs of newly added replacement `LUT`s, whose select lines are g_i's inputs. This allows each $w(g_i)$ to be any function of the inputs of g_i, thus modeling any gate change at g_i, when $e(g_i) = \mathtt{1}$. This construction is illustrated in Figure 6.5, where shaded multiplexers are added for gates g_1 and g_4 (we have skipped the remaining gates to avoid overcrowding the figure). As before, for each gate g_i, the set:

$$\mathbf{d}(g_i) = \{d_j(g_i) \mid j = 1, \ldots, 2^{|fanin(g_i)|}\}$$

refers to the configuration bits of the replacement `LUT` $w(g_i)$.

Informally, the QBF problem can be stated as follows:

> *Do there exist N gates in the specifications (*$\mathbf{e}(\mathbf{g})$*), such that for* any *modification of these gates* ($\mathbf{d}(\mathbf{g})$*), there exists a PPC configuration (*$\hat{\mathbf{c}}$*), such that for all primary inputs, this PPC correctly implements the modified specification?*

Adding cardinality constraints $\Phi_N(\mathbf{e})$, applying common primary inputs and forcing the primary outputs to be equal, we get the matrix in Figure 6.5 and the following QBF formulation:

$$\exists \mathbf{e}(\mathbf{g})\ \forall \mathbf{d}(\mathbf{g})\ \exists \hat{\mathbf{c}}\ \forall \mathbf{x}\ \exists \mathbf{g}, \hat{\mathbf{g}}, \mathbf{w}(\mathbf{g})\ .$$

$$\mathcal{C}_{eco}(\mathbf{x}, \mathbf{y}, \mathbf{g}, \mathbf{e}(\mathbf{g}), \mathbf{w}(\mathbf{g}), \mathbf{d}(\mathbf{g})) \wedge \hat{\mathcal{C}}(\mathbf{x}, \hat{\mathbf{y}}, \hat{\mathbf{g}}, \hat{\mathbf{c}}) \wedge (\mathbf{y} = \hat{\mathbf{y}}) \wedge \Phi_N(\mathbf{e}) \tag{6.12}$$

where $\mathbf{e}(\mathbf{g})$ and $\mathbf{w}(\mathbf{g})$ are defined similarly to (6.1).

In (6.12), only $\mathbf{e}(\mathbf{g})$ is in the widest scope, so counting all the satisfying assignments to $\mathbf{e}(\mathbf{g})$ using blocking clauses gives the number of N-gates where any change can be implemented by the PPC using reconfigurations.

6.4.3 Problem Partitioning

In the case where exactly one gate is allowed to arbitrarily change in the specification (*i.e.*, $N = 1$), (6.12) can be partitioned into $|\mathbf{g}|$ smaller, independent QBFs by enumerating each $e(g_i) \in \mathbf{e}(\mathbf{g})$. For each gate g_i, we let:

$$\mathcal{C}_{eco}|_{e(g_i)} \triangleq \mathcal{C}_{eco} \wedge e(g_i) \wedge \bigwedge_{e(g_j) \in \mathbf{e}(\mathbf{g}) - \{e(g_i)\}} (\neg e(g_j)) \tag{6.13}$$

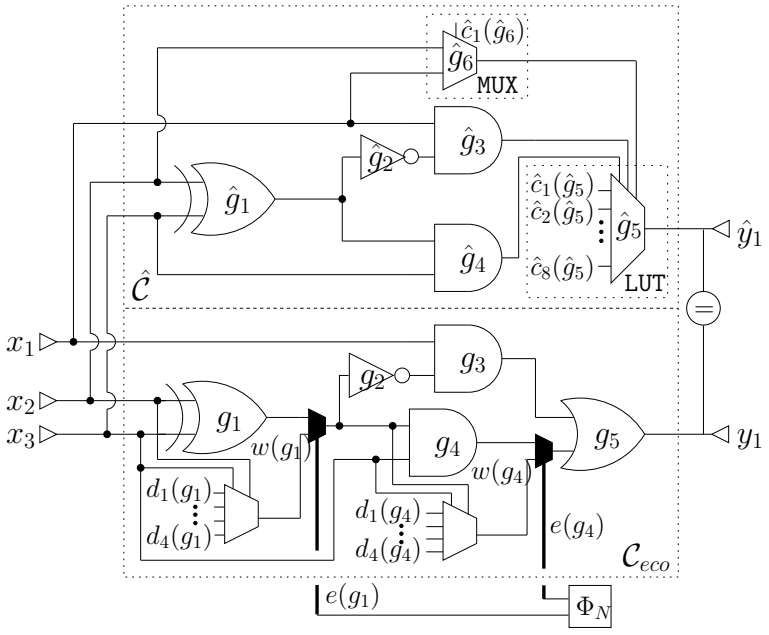


Figure 6.5: ECO coverage matrix

denote the specification where only g_i is allowed to change. We now ask whether for all possible changes at g_i, there exists a PPC configuration that can implement it. Formally,

$$\forall \mathbf{d}(g_i)\ \exists \hat{\mathbf{c}}\ \forall \mathbf{x}\ \exists \mathbf{g}, \hat{\mathbf{g}}, w(g_i)\ .\ \mathcal{C}_{eco}(\mathbf{x}, \mathbf{y}, \mathbf{g}, \mathbf{e}(\mathbf{g}), \mathbf{w}(\mathbf{g}), \mathbf{d}(\mathbf{g}))|_{e(\hat{g}_i)} \wedge \hat{\mathcal{C}}(\mathbf{x}, \hat{\mathbf{y}}, \hat{\mathbf{g}}, \hat{\mathbf{c}}) \wedge (\mathbf{y} = \hat{\mathbf{y}}) \quad (6.14)$$

In each QBF of the form of (6.14), all $\mathbf{d}(g_j)$ and $w(g_j)$ with $j \neq i$ can be disregarded, since they cannot propagate through the shaded multiplexers in Figure 6.5. For each gate, a QBF of the form of (6.14) must be solved to determine whether all possible modifications at that gate in the specification can be implemented by the PPC. All these QBFs can be solved in parallel. The ECO coverage of the PPC is equal to the ratio of these QBFs that are true. Again, this partitioning scheme can be extended to higher cardinalities, however the number of independent subproblems increases exponentially with N.

6.5 Experimental Results

This section presents the experimental evaluation of 21 PPCs from [120] using our proposed QBF formulations. These PPCs are generated by [120] from some of the MCNC benchmark

circuits [121]. Experiments are run on a quad-core Intel i5, 3.1 Ghz workstation with 16 GB of RAM. Since complex faults can be modeled using single stuck-at-faults [58], and given the limited number of LUTs in the PPCs of [120], we set $N = 1$ in our tolerance and coverage calculations. We use the proposed QBF partitioning schemes in Subsections 6.3.3 and 6.4.3 to speed up the solving process. For each tolerance/coverage computation, the QBF subproblems are solved in parallel over the four cores. A time-out of 100 seconds is used for *each* QBF subproblem. The QBF solver `sKizzo-v0.11c` [13] is used to solve all QBF instances. Other QBF solvers, such as `QuBE7` [47] give similar results.

Table 6.1 gives some basic information on the PPC instances from [120]. The first five columns respectively show the PPC name, its number of gates $|\hat{\mathbf{g}}|$, lines $|\hat{\mathbf{l}}|$, added LUTs and added MUXs. Next, columns *added lines* and *% added lines* respectively show the number of redundant lines added by [120] to the LUTs/MUXs and the percentage of added lines to all lines in the PPC. Column *% LUTs+MUXs* gives the percentage of gates that are added LUTs/MUXs compared to all gates in $|\hat{\mathbf{g}}|$.

Table 6.2 shows the results of our evaluations. The first two columns respectively show the *fault tolerance* of $\hat{\mathcal{C}}$ and the total time required for all the corresponding QBF subproblems to terminate. The next two columns give the *design error tolerance* of $\hat{\mathcal{C}}$ and the total time to compute it. And finally, the *ECO coverage* measure along with its computation run-time are given.

For the circuit *pair* shown in table 6.2, the fault tolerance and ECO coverage are, respectively, at least 40% and at least 52%, because a small number (roughly 5%) of the QBF subproblems for each of these calculations does not terminate by 100 seconds. Note that since the QBF subproblems used in our computations are independent, it is easy to improve our run-times by simply parallelizing more heavily.

Figures 6.6 and 6.7 plot the calculated metrics against *% added lines* and *% LUTs+MUXs*, respectively. As expected, adding more redundant lines to the LUTs/MUXs, and replacing more gates with LUTs increases both fault tolerance and ECO coverage. On the other hand, the correlation of these two variables with design error tolerance is weaker, at least given the considered family of PPCs.

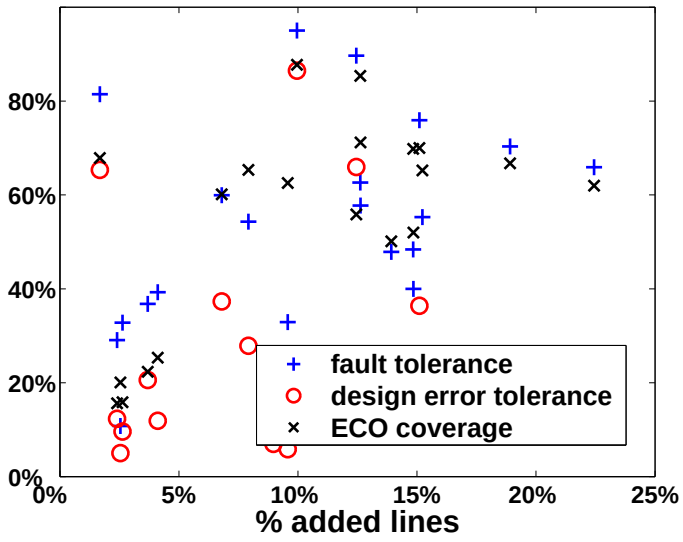


Figure 6.6: Fault tolerance, design error tolerance and ECO coverage vs. % added lines

On average, only 10% of the lines in the PPCs are added as overhead, and only 12% of the gates are added LUTs or MUXs. In fact, LUTs *replace* other gates in the original circuit, so the overhead in the number of added gates is much less than 12%. We found that these PPCs have a 53% average single stuck-at-fault tolerance, a 26% average single gate design error tolerance,

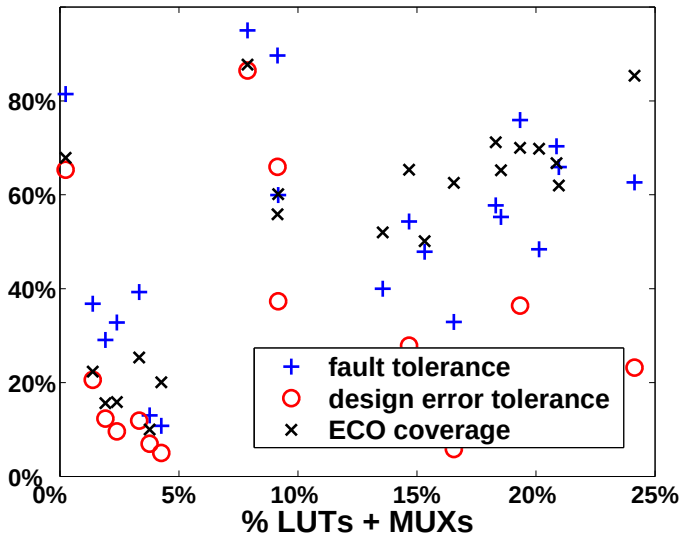


Figure 6.7: Fault tolerance, design error tolerance and ECO coverage vs. % LUTs+MUXs

and a 52% average ECO coverage. From these results, we can conclude that the small hardware overhead is more than compensated by the fault/error tolerance and ECO coverage that these architectures demonstrate, confirming that PPCs are attractive architectures to increase silicon yield and reduce the cost of the design/manufacturing cycle. Furthermore, the existence of methods for computing these metrics encourages further research on improving PPCs.

6.6 Summary

This chapter lays the theoretical groundwork for evaluating the reconfigurability of PPC architectures. QBF encodings are given to calculate the fault tolerance and design error tolerance of a PPC. Next, QBF formulations are proposed for performing ECOs, and for quantifying the ECO coverage of a PPC architecture. The presented encodings demonstrate the applicability of QBFs for dealing with reconfigurability, and the experimental results confirm the attractiveness of PPCs for increasing silicon yield and reducing the cost of the design/manufacturing cycle.

Table 6.1: PPC instance information

$\hat{\mathcal{C}}$	$\lvert\hat{\mathbf{g}}\rvert$	$\lvert\hat{\mathbf{I}}\rvert$	LUTs	MUXs	added lines	% added lines	% LUTs +MUXs
alu2	335	797	5	3	21	3%	2%
alu4	627	1459	7	5	35	2%	2%
apex6	866	1805	98	29	143	8%	15%
apex7	295	586	37	17	74	13%	18%
b9	163	349	20	14	66	19%	21%
c8	145	282	18	6	27	10%	17%
cc	116	206	19	9	26	13%	24%
comp	106	234	2	2	21	9%	4%
example2	477	997	65	31	148	15%	20%
f51m	109	250	8	2	17	7%	9%
frg1	94	197	3	1	5	3%	4%
lal	150	298	19	10	45	15%	19%
mux	60	146	1	1	6	4%	3%
pair	1364	3246	100	85	482	15%	14%
t481	824	2319	1	1	39	2%	$< 1\%$
term1	186	458	10	7	57	12%	9%
too_large	436	1029	3	3	38	4%	1%
vda	749	1928	39	20	192	10%	8%
x1	359	783	35	20	109	14%	15%
x3	912	1871	99	70	285	15%	19%
x4	563	1279	71	47	287	22%	21%

Table 6.2: PPC evaluation results

$\hat{\mathcal{C}}$	fault tolerance	time (sec)	DE tolerance	time (sec)	ECO coverage	time (sec)
alu2	33%	35.5	10%	11.2	16%	12.1
alu4	29%	1145.3	12%	216.7	16%	269.3
apex6	54%	461.0	28%	140.1	65%	197.3
apex7	58%	39.4	22%	9.1	71%	13.6
b9	70%	8.0	23%	2.0	67%	3.0
c8	33%	2.5	6%	0.8	63%	1.2
cc	63%	1.8	23%	0.5	85%	0.7
comp	13%	96.2	7%	60.1	10%	37.8
example2	48%	102.7	17%	22.3	70%	39.2
f51m	60%	1.3	37%	0.5	60%	0.6
frg1	11%	7.1	5%	3.0	20%	4.7
lal	76%	4.7	36%	1.2	70%	1.5
mux	39%	6.9	12%	3.9	25%	4.6
pair	$\geq 40\%$	18431.8	12%	8736.4	$\geq 52\%$	10566.9
t481	81%	7843.7	65%	1871.6	68%	2921.8
term1	90%	64.2	66%	15.9	56%	16.6
too_large	37%	9128.1	21%	1633.7	22%	1946.7
vda	95%	980.6	86%	257.4	88%	234.9
x1	48%	140.4	16%	31.9	50%	42.0
x3	55%	823.4	18%	171.6	65%	222.7
x4	66%	325.5	22%	54.9	62%	78.3

Chapter 7

Conclusion and Future Work

7.1 Summary of Contributions

The VLSI CAD flow contains a myriad of important NP-complete and PSPACE-complete problems. As such, the ever increasing IC design sizes have an exponential effect on the performance of the algorithms used to solve these tasks. Instead of developing dedicated solutions for each of these problems, the trend during the last decade has been to encode them in formal languages, such as SAT and QBF, and focus academic resources on improving SAT and QBF solvers. Stimulated by yearly competitions, the great advances in these solvers have validated and propelled this strategy.

In more detail, a formal methodology for tackling a difficult CAD problem entails (1) the establishment of its complexity class, (2) encoding it in an appropriate formal language, (3) problem-specific optimizations to prune its solution space, and (4) generic improvements in the relevant formal engine. The theoretical and practical contributions of this dissertation span all four of these aspects, albeit for different CAD problems.

- In Chapter 3, several results are presented on the theoretical computational complexity of debugging. Although formal debugging methodologies encoding the problem into SAT and QBF have been competitive, the complexity class of debugging has not been investigated before. We first illustrate that the problem of (a) combinational, (b) gate-level debugging, where (c) no primary input or initial-state variable is unassigned in the counter-example,

and using (d) single error cardinality, is solvable in polynomial time. Next, we provide four proofs showing that relaxing any one of the assumptions (a) to (d) moves the problem from the complexity class P to that of NP-completeness. This establishes that the general debugging problem is NP-complete.

- In Chapter 4, a novel strategy is presented for advancing existing debugging methods, by leveraging dominance relationships between RTL blocks to expedite the bug discovery process. We first give an iterative algorithm that computes dominance relationships between RTL blocks (*e.g.,* always blocks or module definitions). Next, we prove that for each RTL block returned by the automated debugger as a *solution*, an RTL block that dominates it is an implied solution. A solution is a potentially buggy block where a modification can correct the counter-example returned by verification. As such, applying our algorithm as a preprocessing step, the number of formal engine calls for finding all potential error sources is significantly reduced. Furthermore, we prove that corrections at implied solutions can be automatically extracted without explicit formal analysis. These results are shown to be valid for any error cardinality. An extensive set of experiments on real industrial designs demonstrates that 66% of solutions are discovered early due to dominator implications, resulting in a three-fold reduction in the number of formal engine calls and a 1.64x overall performance speed-up.

- In Chapter 5, a new framework is illustrated for exploiting the circuit structure of QBFs by leveraging structural dominators. A methodology and a rigorous proof are given for the removal of subcircuits that are completely dominated by single outputs in a circuit-based QBF, irrespective of the subcircuit input quantifiers or the structure of the remaining circuit, and without affecting the truth of the original QBF. We present a circuit-based QBF preprocessor, called `PReDom`, which automates the process of recursively reducing dominated subcircuits according to the presented methodology. In our experimental results, three state-of-the-art QBF solvers solve 27% to 45% of the QBF instances preprocessed using `PReDom`, compared to *none* without preprocessing.

- In Chapter 6, a series of QBF encodings are presented for evaluating the reconfigurability

of PPCs. We define the metrics of *fault tolerance* and *design error tolerance* as the percentages of stuck-at-faults and localized design errors, respectively, that can be made unobservable using reconfigurations in the PPC. QBF formulations are given for the exact evaluation of both of these measures. Furthermore, the use of PPCs for performing ECOs using QBF is investigated. Finally, the *ECO coverage* of a PPC is defined and a QBF formulation is presented for computing it. The formulations and experimental results in this chapter demonstrate the theoretical and practical appropriateness of QBFs for dealing with reconfigurability.

7.2 Future Work

The following provides a summary of extensions and future directions relating to the contributions of Chapters 4, 5 and 6.

- In Chapter 4, the contrapositive of Theorem 4.6 is also true. In other terms, if we can tell that an RTL block is a *non-solution*, *i.e.,* no change at its outputs can correct the counter-example, then all the blocks it dominates are non-solutions. However, whereas solution blocks are returned by the solver as soon as they are found, non-solutions blocks are not known until the end of the solving process.

  In order to detect non-solutions on-the-fly, we must modify the SAT solver. This can be done by using a SAT branching scheme where error-select variables are decided upon first, and by monitoring the decision tree for learned error-select variables, and therefore non-solutions. Blocks dominated by detected non-solutions can then be implied as non-solutions, further pruning the solution-space. This extension has already been published [68], and a related journal paper has been submitted [78] that combines it with the contributions of Chapter 4. Experiments demonstrate that performing both solution and non-solution implications results in a further speedup of 1.7x in SAT solving time over performing only solution implications [68].

- One of the limitations of the work in Chapter 5 is that many circuit-based QBFs do not have complete dominators at preprocessing time. However, during the solving process,

truth assignments to certain circuit nodes might create complete dominators on-the-fly, which can subsequently be reduced in that decision branch. Complete dominators can also be created due to certain nodes becoming don't-cares as a result of assignments to other nodes. In order to exploit this, a circuit-based QBF solver first needs to observe nodes that have the potential to become complete dominators during the solving process, and later detect and reduce *induced* complete dominators on-the-fly.

We made a prolonged attempt to build a QBF solver that reduces subcircuits which become SODSes on-the-fly, and that outperforms state-of-the-art solvers. We devised node observation heuristics, and created a dynamic complete dominator detection algorithm. We also extended the QBF reduction cases in Table 5.1 to yield *conflicts*, which can happen inside a solver and would force it to backtrack earlier than it would otherwise. This was done using CirQit [53] as the base QBF solver. However, the results were disappointing because the overhead was too high. Unlike in `PReDom`, gates had to be removed and re-inserted over and over due to solver backtracks, which was costly. Furthermore, learning clauses was made difficult when variables changed scopes due to reductions.

Other potential improvements to our work on leveraging dominators in QBF include dealing with multiple-output complete dominators, which occur more frequently than single-output complete dominators. Unfortunately, without multiple-vertex blocks that are specified *a priori*, such as in Chapter 4, computing multiple-output dominators is a resource-intensive process [8, 54, 65].

Another idea is to use a BDD to represent the decision tree shown in Figure 5.5. This can make the reduction process more scalable, which can make it possible to handle bigger SODSes with more primary inputs. This in turn can lead to more reductions.

- Finally, there are three areas for extensions and future directions with respect to the contributions of Chapter 6. The first is to use higher-level models for design errors and ECOs in our tolerance and coverage metrics. For instance, for computing the percentage of specification changes that can be implemented using reconfigurations, the SVA debugging models in [62] are a good starting point. Second, today's QBF solvers cannot handle the

the presented encodings using large designs. As such, an important challenge is to come up with faster estimation techniques for these metrics, with or without QBF, at the cost of acceptably small errors. Finally, the ultimate goal is to use these metrics as objective functions in iterative algorithms that create PPCs by placing the reconfigurable LUTs in a way to maximize reconfigurability.

Bibliography

[1] *International technology roadmap for semiconductors*, 2007.

[2] *International technology roadmap for semiconductors*, 2010.

[3] M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*. Computer Science Press, 1990.

[4] M. Abramovici, C. Stroud, and M. Emmert, "Using embedded FPGAs for SoC yield improvement," in *Design Automation Conference*, 2002, pp. 713–724.

[5] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[6] M. F. Ali, S. Safarpour, A. Veneris, M. Abadir, and R. Drechsler, "Post-verification debugging of hierarchical designs," in *International Conference on CAD*, 2005, pp. 871–876.

[7] F. E. Allen and J. Cocke, "Graph-theoretic constructs for program flow analysis," Technical Report RC 3923 (17789), IBM Thomas J. Watson Research Center, Tech. Rep., 1972.

[8] S. Alstrup, J. Clausen, and K. Jørgensen, "An $O(|V|*|E|)$ algorithm for finding immediate multiple-vertex dominators," *Information Processing Letters*, vol. 59, no. 1, pp. 9–11, 1996.

[9] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup, "Dominators in linear time," *SIAM Journal on Computing*, vol. 28, no. 6, pp. 2117–2132, 1999.

[10] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *International Joint Conference on Artificial Intelligence*, 2009, pp. 399–404.

[11] K. Batcher, "Sorting networks and their applications," in *AFIPS, Spring Joint Computer Conference*, 1968, pp. 307–314.

[12] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen, "Scalable sequential equivalence checking across arbitrary design transformations," in *International Conference on Computer Design*, 2006, pp. 259–266.

[13] M. Benedetti, "sKizzo: a suite to evaluate and certify QBFs," in *International Conference on Automated Deduction*, 2005, pp. 369–376.

[14] M. Benedetti, A. Lallouet, and J. Vautard, "QCSP made practical by virtue of restricted quantification," in *International Joint Conference on Artificial Intelligence*, 2007, pp. 38–43.

[15] M. Benedetti, "Evaluating QBFs via symbolic skolemization," in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 2004, pp. 285–300.

[16] ——, "Quantifier trees for QBFs," in *International Conference on Theory and Applications of Satisfiability Testing*, 2005, pp. 378–385.

[17] M. Benedetti and H. Mangassarian, "QBF-based formal verification: experience and perspectives," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 5, pp. 133–191, 2008.

[18] A. Biere, "Resolve and expand," in *International Conference on Theory and Applications of Satisfiability Testing*, 2004, pp. 238–246.

[19] ——, "PicoSAT essentials," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 4, pp. 75–97, 2008.

[20] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, 1999, pp. 193–207.

[21] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in computers*, vol. 58, pp. 117–148, 2003.

[22] D. Brand, A. Drumm, S. Kundu, and P. Narain, "Incremental synthesis," in *International Conference on CAD*, 1994, pp. 14–18.

[23] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[24] H. K. Büning and U. Bubeck, "Theory of quantified Boolean formulas," *Handbook of Satisfiability*, pp. 735–760, 2009.

[25] H. K. Büning, M. Karpinski, and A. Flögel, "Resolution for quantified Boolean formulas," *Information and Computation*, vol. 117, pp. 12–18, 1995.

[26] M. Cadoli, M. Schaerf, A. Giovanardi, and M. Giovanardi, "An algorithm to evaluate quantified Boolean formulae and its experimental evaluation," *Journal of Automated Reasoning*, vol. 28, no. 2, pp. 101–142, 2002.

[27] Y. Chen, S. Safarpour, J. Marques-Silva, and A. Veneris, "Automated design debugging with maximum satisfiability," *IEEE Transactions on CAD*, vol. 29, pp. 1804–1817, 2010.

[28] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: an opensource tool for symbolic model checking," in *Computer Aided Verification*, 2002, pp. 359–364.

[29] J. Cong and Y. Ding, "An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Transactions on CAD*, vol. 13, pp. 1–12, 1994.

[30] S. Cook, "The complexity of theorem proving procedures," in *Symposium on Theory of Computing*, 1971, pp. 151–158.

[31] K. Cooper, T. Harvey, and K. Kennedy, "A simple, fast dominance algorithm," *Software Practice & Experience*, vol. 4, pp. 1–10, 2001.

[32] F. Corblin, L. Bordeaux, E. Fanchon, Y. Hamadi, and L. Trilling, "Connections and integration with SAT solvers: a survey and a case study in computational biology," *Hybrid Optimization*, pp. 425–461, 2011.

[33] T. Cormen, C. Leierson, and R. Rivest, *Introduction to Algorithms*. MIT Press, 2009.

[34] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving," *Communications of the ACM*, vol. 5, pp. 394–397, 1962.

[35] N. Dershowitz, Z. Hanna, and J. Katz, "Bounded model checking with QBF," in *International Conference on Theory and Applications of Satisfiability Testing*, 2005, pp. 408–414.

[36] A. Doumar and H. Ito, "Detecting, diagnosing, and tolerating faults in SRAM-based field programmable gate arrays: a survey," *IEEE Transactions on VLSI Systems*, vol. 11, no. 3, pp. 386–405, 2003.

[37] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *International Conference on Theory and Applications of Satisfiability Testing*, 2005, pp. 102–104.

[38] N. Eén and N. Sörensson, "An extensible SAT-solver," in *International Conference on Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.

[39] ——, "Translating pseudo-Boolean constraints into SAT," vol. 2, pp. 1–26, 2006.

[40] U. Egly, M. Seidl, and S. Woltran, "A solver for QBFs in negation normal form," *Constraints*, vol. 14, no. 1, pp. 38–79, 2009.

[41] M. Fahim Ali, A. Veneris, S. Safarpour, R. Drechsler, A. Smith, and M. Abadir, "Debugging sequential circuits using Boolean satisfiability," in *International Conference on CAD*, 2004, pp. 204–209.

[42] H. Foster, "From volume to velocity: the transforming landscape in function verification," in *Design Verification Conference*, 2011.

[43] A. T. Freitas, H. C. Neto, and A. L. Oliveira, "On the complexity of power estimation problems," in *International Workshop on Logic and Synthesis*, 2000, pp. 239–244.

[44] M. Ganai and A. Gupta, "Efficient BMC for multi-clock systems with clocked specifications," in *Asia and South Pacific Design Automation Conference*, 2007, pp. 310–315.

[45] L. Georgiadis and R. E. Tarjan, "Finding dominators revisited: extended abstract," in *ACM-SIAM Symposium on Discrete Algorithms*, 2004, pp. 869–878.

[46] E. Giunchiglia, M. Narizzano, and A. Tacchella, "Quantified Boolean formulas satisfiability library (QBFLIB)," 2001, http://www.qbflib.org.

[47] ——, "QuBE: A system for deciding quantified Boolean formulas satisfiability," *Journal of Automated Reasoning*, pp. 364–369, 2001.

[48] ——, "Quantifier structure in search-based procedures for QBFs," *IEEE Transactions on CAD*, vol. 26, no. 3, pp. 497–507, 2007.

[49] E. Goldberg and Y. Novikov, "On complexity of equivalence checking," Technical Report CDNL-TR-2003-0826, Cadence Berkeley Labs, CA, Tech. Rep., 2003.

[50] E. Goldberg, M. Prasad, and R. Brayton, "Using SAT for combinational equivalence checking," in *Design, Automation and Test in Europe*, 2001, pp. 114–121.

[51] C. Gomes, B. Selman, and H. Kautz, "Boosting combinatorial search through randomization," in *National Conference on Artificial Intelligence (AAAI)*, 1998, pp. 431–437.

[52] M. Gort and J. Anderson, "Reducing FPGA router run-time through algorithm and architecture," in *International Conference on Field Programmable Logic and Applications*, 2011, pp. 336–342.

[53] A. Goultiaeva, V. Iverson, and F. Bacchus, "Beyond CNF: A circuit-based QBF solver," in *International Conference on Theory and Applications of Satisfiability Testing*, 2009, pp. 412–426.

[54] R. Gupta, "Generalized dominators and post-dominators," in *Symposium on Principles of Programming Languages*, 1992, pp. 246–257.

[55] S. Huang and K. Cheng, *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publishers, 1998.

[56] S.-Y. Huang and K.-T. Cheng, "Errortracer: design error diagnosis based on fault simulation techniques," *IEEE Transactions on CAD*, vol. 18, no. 9, pp. 1341–1352, 1999.

[57] O. Ibarra and S. Sahni, "Polynomially complete fault detection problems," *IEEE Transactions on Computers*, vol. 24, pp. 242–249, 1975.

[58] N. Jha and S. Kundu, *Testing and Reliable Design of CMOS Circuits*. Kluwer Academic Publishers, 1990.

[59] T. Jussila and A. Biere, "Compressing BMC encodings with QBF," in *International Workshop on Bounded Model Checking*, 2006, pp. 1–14.

[60] J. Kam and J. Ullman, "Global data flow analysis and iterative algorithms," *Journal of the ACM*, vol. 23, no. 1, pp. 158–171, 1976.

[61] B. Keng, S. Safarpour, and A. Veneris, "Bounded model debugging," *IEEE Transactions on CAD*, vol. 29, no. 11, pp. 1790–1803, 2010.

[62] ——, "Automated debugging of SystemVerilog assertions," in *Design, Automation and Test in Europe*. IEEE, 2011, pp. 1–6.

[63] B. Keng and A. Veneris, "Scaling VLSI design debugging with interpolation," in *International Conference on Formal Methods in CAD*, 2009, pp. 144–151.

[64] T. Kirkland and M. R. Mercer, "A topological search algorithm for ATPG," in *Design Automation Conference*, 1987, pp. 502–508.

[65] R. Krenz and E. Dubrova, "A fast algorithm for finding common multiple-vertex dominators in circuit graphs," in *Asia and South Pacific Design Automation Conference*, 2005, pp. 529–532.

[66] Y. Kuo, Y. Chang, S. Chang, and M. Marek-Sadowska, "Engineering change using spare cells with constant insertion," in *International Conference on CAD*, 2007, pp. 544–547.

[67] T. Larrabee, "Test pattern generation using Boolean satisfiability," *IEEE Transactions on CAD*, vol. 11, pp. 4–15, 1992.

[68] B. Le, B. Keng, H. Mangassarian, and A. Veneris, "Non-solution implications using reverse domination in a modern SAT-based debugging environment," in *Design, Automation and Test in Europe*, 2012.

[69] D. le Berre, O. Roussesl, and L. Simon, "The international SAT competition webpage," 2010, http://www.satcompetition.org.

[70] T. Lee, W. Chuang, I. Hajj, and W. Fuchs, "Circuit-level dictionaries of CMOS bridging faults," in *IEEE VLSI Test Symposium*, 1994, pp. 386–391.

[71] T. Lengauer, R. Endre, and T. Jan, "A fast algorithm for finding dominators in a flow-graph," *ACM Transactions on Programming Languages and Systems*, vol. 1, pp. 121–141, 1979.

[72] A. Ling, D. Singh, and S. Brown, "FPGA PLB architecture evaluation and area optimization techniques using Boolean satisfiability," *IEEE Transactions on CAD*, vol. 26, no. 7, pp. 1196–1210, 2007.

[73] J. Liu and A. Veneris, "Incremental fault diagnosis," *IEEE Transactions on CAD*, vol. 24, no. 2, pp. 240–251, 2005.

[74] F. Lonsing and A. Biere, "Nenofex: expanding NNF for QBF solving," in *International Conference on Theory and Applications of Satisfiability Testing*, 2008, pp. 196–210.

[75] F. Lu and S. Malik, "Conflict driven learning in a quantified Boolean satisfiability solver," in *International Conference on CAD*, 2002, pp. 442–449.

[76] R. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

[77] H. Mangassarian, B. Le, A. Goultiaeva, A. Veneris, and F. Bacchus, "Leveraging dominators for preprocessing QBF," in *Design, Automation and Test in Europe*, 2010, pp. 1695–1700.

[78] H. Mangassarian, B. Le, and A. Veneris, "Debugging with dominance: On-the-fly RTL debug solution and non-solution implications," *IEEE Transactions on CAD*, 2012 (submitted).

[79] H. Mangassarian, A. Veneris, and M. Benedetti, "Robust QBF encodings for sequential circuits with applications to verification, debug, and test," *IEEE Transactions on Computers*, vol. 59, no. 7, pp. 981–994, 2010.

[80] H. Mangassarian, A. Veneris, and F. Najm, "Maximum circuit activity estimation using pseudo-Boolean satisfiability," *IEEE Transactions on CAD*, vol. 31, no. 2, pp. 271–284, 2012.

[81] H. Mangassarian, H. Yoshida, A. Veneris, S. Yamashita, and M. Fujita, "On error tolerance and engineering change with partially programmable circuits," in *Asia and South Pacific Design Automation Conference*, 2012, pp. 695–700.

[82] H. Mangassarian, A. G. Veneris, D. E. Smith, and S. Safarpour, "Debugging with dominance: On-the-fly RTL debug solution implications," in *International Conference on CAD*, 2011, pp. 587–594.

[83] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning SAT solvers," *Handbook of Satisfiability*, pp. 131–154, 2009.

[84] J. Marques-Silva and K. Sakallah, "GRASP: a search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 506–521, 1999.

[85] D. McGrath, "De Geus touts new products, says ICs will rebound," in *EE Times*, March 2009.

[86] A. Mishchenko, M. Case, R. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis," in *International Conference on CAD*, 2008, pp. 234–241.

[87] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *Design Automation Conference*, 2001, pp. 530–535.

[88] G.-J. Nam, K. Sakallah, and R. Rutenbar, "A new FPGA detailed routing approach via search-based Boolean satisfiability," *IEEE Transactions on CAD*, vol. 21, no. 6, pp. 674–684, 2002.

[89] T. Niermann and J. H. Patel, "HITEC: a test generation package for sequential circuits," in *European Design Automation Conference*, 1991, pp. 214–218.

[90] W. Ning, "Strongly NP-hard discrete gate-sizing problems," *IEEE Transactions on CAD*, vol. 13, no. 8, pp. 1045–1051, 1994.

[91] OpenCores.org, "http://www.opencores.org," 2007.

[92] C. Papadimitriou, *Computational complexity*. John Wiley and Sons Ltd., 2003.

[93] C. Peschiera, L. Pulina, A. Tacchella, U. Bubeck, O. Kullmann, and I. Lynce, "The seventh QBF solvers evaluation (QBFEVAL'10)," *International Conference on Theory and Applications of Satisfiability Testing*, pp. 237–250, 2010.

[94] L. Pulina and A. Tacchella, "A self-adaptive multi-engine solver for quantified Boolean formulas," *Constraints*, vol. 14, no. 1, pp. 80–116, 2009.

[95] P. Rashinkar, P. Paterson, and L. Singh, *System-on-a-Chip Verification: Methodology and Techniques*. Kluwer Academic Publishers, 2000.

[96] J. Rintanen, "Planning and SAT," in *Handbook of Satisfiability*, 2009, pp. 483–504.

[97] A. Sabharwal, C. Ansótegui, C. P. Gomes, J. W. Hart, and B. Selman, "QBF modeling: Exploiting player symmetry for simplicity and efficiency," in *International Conference on Theory and Applications of Satisfiability Testing*, 2006, pp. 382–395.

[98] S. Safarpour, A. Veneris, R. Drechsler, and J. Hang, "Managing don't-cares in Boolean satisfiability," in *Design, Automation and Test in Europe*, 2004, pp. 260–265.

[99] S. Safarpour, "Formal methods in automated design debugging," Ph.D. dissertation, University of Toronto, 2009.

[100] S. Safarpour and A. Veneris, "Automated design debugging with abstraction and refinement," *IEEE Transactions on CAD*, vol. 28, no. 10, pp. 1597–1608, 2009.

[101] S. Sahni and A. Bhatt, "The complexity of design automation problems," in *Design Automation Conference*, 1980, pp. 402–411.

[102] H. Samulowitz and F. Bacchus, "Binary clause reasoning," in *International Conference on Theory and Applications of Satisfiability Testing*, 2006, pp. 353–367.

[103] ——, "Dynamically partitioning for solving QBF," in *International Conference on Theory and Applications of Satisfiability Testing*, 2007, pp. 215–229.

[104] H. Samulowitz, J. Davies, and F. Bacchus, "Preprocessing QBF," in *International Conference on Principles and Practice of Constraint Programming*, 2006, pp. 514–529.

[105] S. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, and J. Torrellas, "Patching processor design errors with programmable hardware," *IEEE Micro*, vol. 27, no. 1, pp. 12–25, 2007.

[106] W. Savitch, "Relationships between nondeterministic and deterministic tape complexities," *Journal of Computer and System Sciences*, vol. 4, no. 2, pp. 177–192, 1970.

[107] L. Silva, J. Marques-Silva, L. Silveira, and K. Sakallah, "Timing analysis using propositional satisfiability," in *IEEE International Conference on Electronics, Circuits and Systems*, vol. 3, 1998, pp. 95–98.

[108] A. Sistla and E. Clarke, "The complexity of propositional linear temporal logics," *Journal of the ACM*, vol. 32, no. 3, pp. 733–749, 1985.

[109] A. Smith, "Diagnosis of combinational logic circuits using Boolean satisfiability," Master's thesis, University of Toronto, 2004.

[110] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Transactions on CAD*, vol. 24, no. 10, pp. 1606–1621, 2005.

[111] A. Smith, A. Veneris, and A. Viglas, "Design diagnosis using Boolean satisfiability," in *Asia and South Pacific Design Automation Conference*, 2004, pp. 218–223.

[112] S. Staber and R. Bloem, "Fault localization and correction with QBF," in *International Conference on Theory and Applications of Satisfiability Testing*, 2007, pp. 355–368.

[113] A. Sülflow, G. Fey, and R. Drechsler, "Using QBF to increase accuracy of SAT-based debugging," in *IEEE International Symposium on Circuits and Systems*, 2010, pp. 641–644.

[114] G. Swamy, S. Rajamani, C. Lennard, and R. Brayton, "Minimal logic re-synthesis for engineering change," in *IEEE International Symposium on Circuits and Systems*, vol. 3, 1997, pp. 1596–1599.

[115] D. Tang and S. Malik, "Solving quantified Boolean formulas with circuit observability don't cares," in *International Conference on Theory and Applications of Satisfiability Testing*, 2006, pp. 368–381.

[116] G. S. Tseitin, "On the complexity of derivations in the propositional calculus," in *Studies in Constructive Mathematics and Mathematical Logic*, 1968, pp. 115–125.

[117] A. Veneris and S. Safarpour, "The day Sherlock Holmes decided to do EDA," in *Design Automation Conference*, 2009, pp. 631–634.

[118] Y. Wu and D. Chang, "On the NP-completeness of regular 2-D FPGA routing architectures and a novel solution," in *International Conference on CAD*, 1994, pp. 362–366.

[119] S. Yamashita, H. Sawada, and A. Nagoya, "SPFD: A new method to express functional flexibility," *IEEE Transactions on CAD*, vol. 19, no. 8, pp. 840–849, 2000.

[120] S. Yamashita, H. Yoshida, and M. Fujita, "Increasing yield using partially-programmable circuits," in *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, 2010, pp. 237–242.

[121] S. Yang, "Logic synthesis and optimization benchmarks user guide version 3.0," *MCNC*, 1991.

[122] L. Zhang, "Solving QBF with combined conjunctive and disjunctive normal forms," in *National Conference on Artificial Intelligence (AAAI)*, 2006.

[123] Q. Zhu, N. Kitchen, A. Kuehlmann, and A. Sangiovanni-Vincentelli, "SAT sweeping with local observability don't-cares," in *Design Automation Conference*, 2006, pp. 229–234.