This is to certify that the

dissertation entitled

DESIGN RECOVERY FOR COMBINATIONAL LOGIC
EXPLOITING BOOLEAN RELATIONSHIPS

presented by

Travis E. Doom

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Computer Science

Major professor

Date July 2, 1998

DATE DUE	DATE DUE	DATE DUE

DESIGN RECOVERY FOR COMBINATIONAL LOGIC EXPLOITING BOOLEAN RELATIONSHIPS

By

Travis Edward Doom

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science and Engineering

1998

ABSTRACT

DESIGN RECOVERY FOR COMBINATIONAL LOGIC EXPLOITING BOOLEAN RELATIONSHIPS

By

Travis Edward Doom

The reengineering of digital circuits is an increasingly important problem in design automation. To effectively redesign a digital system, it must be completely described at a level detailed enough to allow for the synthesis of a new design. Recovering this level of design from an existing implementation may be complicated as a result of errors in the original design, incomplete information, or missing documentation. This dissertation describes new techniques that facilitate the recovery of high-level design information from low-level, and possibly incomplete, descriptions of digital systems.

The recovery of design information from a complete logic layout can sometimes be achieved by identifying common implementations of high-level design entities in the layout. Existing *syntactic pattern matching* techniques attempt to identify high-level modules by identifying subcircuits within the layout that exactly correspond to common implementations of the high-level module. However, such syntactic approaches fail to identify functionally equivalent subcircuits that are optimized, obfuscated,

or otherwise different from standard implementations. This dissertation presents a mechanism for *semantic pattern matching* that overcomes many of these limitations by identifying subcircuits that are functionally (as opposed to structurally) equivalent to high-level modules.

This thesis also presents techniques based upon binary decision diagrams (BDDs) that allow the representation of relationships between structures in a known or partially known combinational logic. Designs are represented as structural BDDs (SBDDs), which contain decision variables for internal circuit structures. This thesis presents SBDD-based techniques for representing partially specified logic and the relationships that determine the behavior of the represented device. These techniques are used to detect conflicts and deduce unspecified functional behavior from structural context and available additional information.

Both semantic pattern matching and SBDD-based techniques provide a mechanism for the effective recovery of combinational device designs and of the combinational logic present in sequential devices. These techniques have proven to be effective tools for redesign that can represent internal Boolean relationships in a fully or partially specified multiple-output combinational logic circuit with a single data structure and that can identify the high-level functionality of structures within the device.

DEDICATED TO MY FAMILY, WHO NEVER GOT TIRED OF WAITING FOR ME TO FINISH IT.

ACKNOWLEDGMENTS

I would like to thank my advisor, Anthony Wojcik, for his insight, guidance, and advise in both this research and in my career. I am also exceptionally grateful to Gregory Chisholm of Argonne National Laboratory who encouraged this research and introduced me to the larger scope of the problem domain.

I gratefully acknowledge the contributions of the Design Automation Research Group at Michigan State University's College of Engineering whose comments and feedback helped decide the direction of this research. I would particularly like to thank Moon-Jung Chung and Chin-Long Wey for their support in this research. Likewise, I gratefully acknowledge the contributions of my colleagues at Argonne National Laboratory whose initial work provided a foundation for my research. I would particularly like to thank Steve Eckmann and Ken Dritz for their aid and insight.

This work was supported in part by Argonne National Laboratory's Division of Information Science and through a dissertation completion fellowship provided by the Graduate School of Michigan State University. I am greatly indebted to both institutions for their support.

Lastly, and most importantly, I thank Jennifer White for her professional and personal support. Her dissertation research has proved invaluable to this approach. Her constant encouragement, support, and friendship has made this research possible.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Of considerable interest in the design automation community is the remanufacture and reengineering of digital hardware systems. The basic goal of remanufacture is to develop a set of specifications for an existing system for the purpose of making a clone of the original hardware system (Rekoff, 1985). The remanufacture of so-called legacy systems[1] is common (Dukes, 1994). The basic goal of reengineering is to respecify a digital system so that it is better, in some way, than the original form. In both cases, hardware specifications must be determined through an analysis of available system information. This analysis, which is the subject of this dissertation, is referred to as reverse engineering.

Many circuits are developed without the assistance of a comprehensive, computer-aided design (CAD) process. As a result, detailed information about the system at various levels of design may not be available (Bryant, 1993). Moreover, even when an extensive amount of system design documentation is available, portions of the

[1]**Legacy system:** An older or outdated device currently being used in the field.

documentation may be missing, or out-of-date, or design rules and transformations may have been employed that are not able to be used to reverse the design process and thus specify the design at a more abstract level. Of course, no matter how much documentation is available, many systems are designed or modified manually without proper documentation, a situation that can result in the system documentation being incorrect (Keutzer, 1996).

For any existing digital system, the most accurate and up-to-date system representation is the working physical hardware. Design documentation, if available, cannot be trusted to describe the current system accurately. In acknowledgment of this potential discrepancy between how a digital system actually functions and how its operation is specified in the documentation, this dissertation describes a methodology for the reverse engineering of such systems.

Chapter 1 reviews current reverse engineering terminology and provides an overview of a conceptual model. To explain the motivation behind this research, it briefly describes an existing problem for which sophisticated reverse engineering techniques are necessary. It also provides information on the state-of-the-art reverse engineering solutions and technologies and points out the relevance of the thesis in this context. It describes the contributions made by the work described in this dissertation towards solving reverse engineering problems. Finally, it outlines the remainder of the dissertation.

1.1 Background

The successful recovery of the design information for a digital system depends on how well information at various levels of detail can be transformed during the reverse engineering process. Before addressing any specific aspects of our approach to design recovery, we review the current terminology used in the reengineering literature and describe the traditional abstraction levels and design transformations that occur in the typical design process. We also discuss the role of verification in the design process and the applicability of such verification techniques to reverse engineering.

1.1.1 Reengineering taxonomy

Definitions for the fundamental ideas that characterize reengineering are often implicitly assumed when this subject is discussed. Although reverse engineering had its origin in the analysis of hardware (Rekoff, 1985), reverse engineering approaches are now commonly to applied to software systems as well (Chikofsky and Cross, 1990). We present here only those terms that deal with the reengineering of hardware and define these terms in that context.

Based on the concepts of life cycles and abstractions described by Chikofsky and Cross (1990), one can assume that an orderly *life-cycle model* exists for the hardware design and development process. Even though the exact design methodology may vary, the early stages of the design process deal with general, implementation-independent concepts. Later stages emphasize implementation details. Although there are usually iterative cycles within the design process, the process has a general

direction that allows one to reasonably define forward and backward activities. Spanning life-cycle phases involves a transition from a description of the *subject system* expressed at less detailed, higher *levels of abstraction* in the early stages to more detailed, lower levels of abstraction in the later stages. The subject system is represented in a manner appropriate to the level of abstraction present at each life-cycle phase. The subject system may be defined in terms of requirements, behavior, VHDL[2] code, logic structure, metal-layer geometries, or any other appropriate form.

Figure 1.1 illustrates a general model of the design life-cycle. As the level of abstraction decreases, the amount of information represented at each phase of the life-cycle becomes more detailed (hence the pyramid shape).

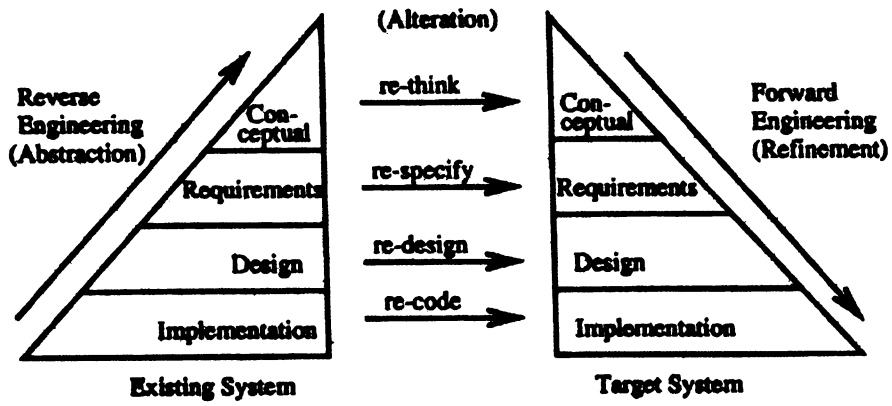


Figure 1.1: **General model for reengineering.** (Byrne, 1992)

Under this definition of the design process, *forward engineering* is defined as "the traditional process of moving from high-level abstractions and logical,

[2]VHDL = very-high-speed integrated circuit (VHSIC) hardware description language. VHDL is a large, high-level VLSI design language with Ada-like syntax that meets the U.S. Department of Defense standard for hardware description (IEEE 1076).

implementation-independent designs to the physical implementation of a system" (Chikofsky and Cross, 1990, page 14). *Reverse engineering* is defined as "the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction" (Chikofsky and Cross, 1990, page 15). Reverse engineering does not involve the modification or remanufacture of the subject system. "It is a process of examination, not a process of change or replication" (Chikofsky and Cross, 1990, page 15). While reverse engineering often involves using an existing functional system as its subject, this is not a requirement; given an appropriate description of the subject system, reverse engineering can be performed starting at any stage of the life cycle.

The term *design recovery* refers to a subarea of reverse engineering "in which domain knowledge, external information, and deduction... are added to the observations of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself. ... Design recovery must reproduce all of the information required for a person to fully understand what a [subject system] does, how it does it, why it does it, and so forth" (Chikofsky and Cross, 1990, page 15).

Finally, the term *reengineering* refers to the "examination and alteration of a subject system to reconsitute it in a new form and the subsequent implementation of the new form" (Chikofsky and Cross, 1990, page 15). Reengineering generally involves some form of reverse engineering, followed by a transformation or alternation of the subject system description at some level of abstraction (*restructuring*), leading to some form of forward engineering. The term *remanufacture* or *clone* refers to a

subarea of reengineering in which the only restructuring done is that necessary to reimplement the subject system by means of current synthesis and fabrication technologies. The goal of remanufacture is not to add additional or better functionality, but to create a surrogate device that preserves the the subject system's functional and semantic external behavior exactly.

Approaches to both software and hardware design recovery depend upon the ability to formally represent a design at various levels of abstraction. It is necessary to be able to construct proofs of correctness that verify that two designs at different levels of abstraction are equivalent. The formal approaches presented in this dissertation are amenable to design recovery for digital hardware. These approaches have not, however, proven amenable to more complex problem of design recovery in software systems.

1.1.2 The design process

In practice, the design phase of a digital system's life-cycle is divided into a number of subphases to make the process tractable. After specifying the conceptual requirements of the design, designers of complex digital hardware use a top-down methodology and hierarchical levels of abstraction to simplify the design process. An important step in this process is a mechanism for verifying that each level of the description hierarchy does not conflict with the level preceding it. Figure 1.2 illustrates the typical hardware design process.

A behavioral model of a digital system is the most abstract design commonly

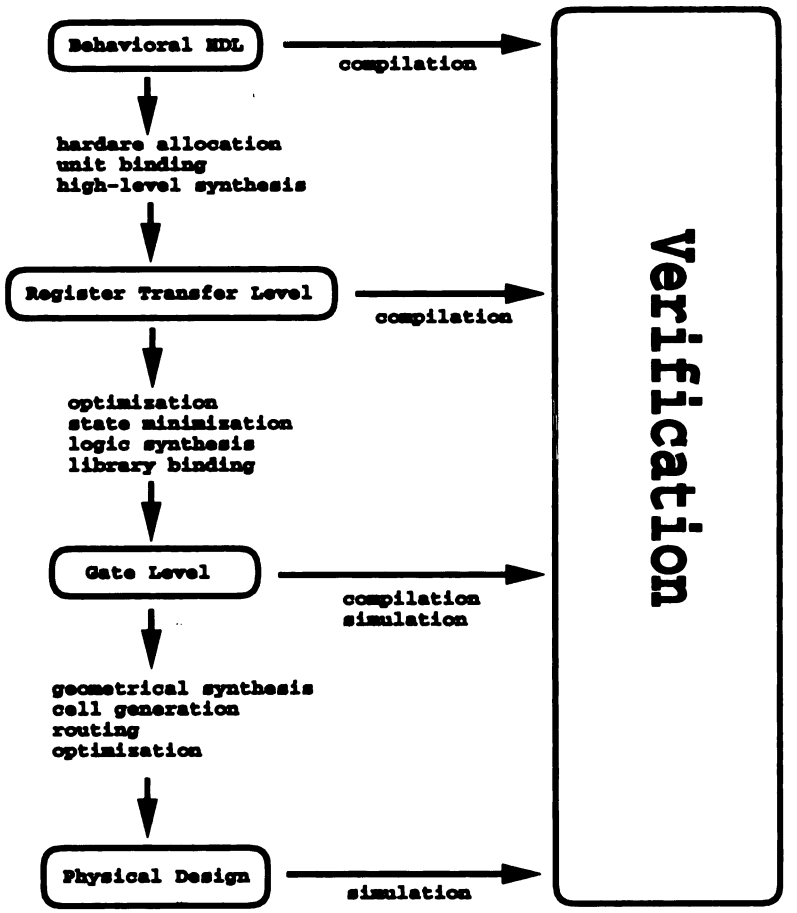


Figure 1.2: **The design process.**

produced. *Behavioral* descriptions describe the function of a system component, regardless of its implementation. At this stage, the designers are able to specify high-level interconnections among high-level functions to create a device that meets the conceptual requirements of the system. This behavioral model is usually written in a hardware description language (HDL), such as VHDL, which allows the designer to test the design against its conceptual requirements though simulation.

When the designers are satisfied that the high-level design is adequate, they produce a more detailed design description for the elements in the behavioral design. *Structural* descriptions describe the system as an interconnection of physical modules. *High-level synthesis* (sometimes referred to as structural or architectural-level synthesis) consists of generating a structural description from a behavioral model. This process involves determining an assignment of the circuit functions to operators, as well as their interconnections and the timing of their execution. The structural description of the macroscopic (i.e., block-level) components of a design generated in this way is referred to as the design's *register-transfer-level* (RTL) description.

Once an adequate RTL description has been determined, the next step is to specify an interconnection of microscopic (i.e., *gate-level*) components that perform the function of the macroscopic design. The process of manipulating and optimizing the logic specifications of the block-level components to create an interconnection of logic primitives is referred to as *logic synthesis.* The complicated task of transforming functional (logic-level) descriptions of RTL components into an optimized interconnection of gate-level library cells is referred to as *library binding* or *technology mapping.*

The lowest level of abstraction included in the design process is the geometrical

description of the physical design. *Geometrical synthesis* is the process of specifying all of the geometric patterns that determine the physical layout of the chip as well as their position. This level of design is sometimes referred to as the *transistor-level* description or the *implementation design.*

The role of verification in synthesis and design recovery

Design verification is the process of proving to some degree of confidence that design descriptions from different levels of abstraction perform the same task. Verification is necessary since optimization decisions make it difficult to prove that high-level components are equivalent to their low-level implementations produced via synthesis. Because synthesis sometimes introduces errors into a design, verification is an important aspect of each level in the design cycle.

Traditionally, the low-level verification of simple designs has been performed by generating test vectors and simulating the input-output behavior of the system (De Micheli, 1994). However, the generation of test vectors and simulation testing is time-consuming and, unless they are exhaustive, do not necessarily guarantee equivalence. More recently, *equivalence checking* has been proposed as a more effective verification technique for complex designs. In equivalence checking, logic functions representing design components are proven to be equivalent, if possible. Although the equivalence checking of designs is one of the most important problems in CAD for logic design, it is known to be a coNP-complete[3] problem, and therefore it uses a number of heuristic

[3]coNP-complete = complementary nondeterministic polynomial time complete: A set or property of computational decision problems with a yes/no answer where the complementary no/yes problem is in the set NP-complete. The set NP-complete is a subset of NP (i.e., can be solved by

techniques to achieve efficient performance (Jain et al., 1997).

These heuristic techniques attempt to identify similarities between circuits to reduce the problem size. Kunz proposed an indirect implication method, called *recursive learning* (Kunz and Pradhan, 1994), which he applied successfully to the equivalence checking problem. Jain et al. proposed another indirect implication technique, called *functional learning* (Jain et al., 1995), which was applied to this same problem. These and similar *structural verification* techniques attempt to identify related nodes in design descriptions to simplify the equivalence checking or test generation process. These techniques determine implications between signal lines and other design structures or otherwise "reason about design intent" to perform this task. It should be kept in mind that the goal of these and other verification techniques (such as *symbolic model checking* (Burch et al., 1990)) is to verify the equivalence between two related design descriptions. Not only must these descriptions be complete, but the correspondences between the variables representing the design inputs and outputs must be clearly defined.

Design recovery, however, is concerned primarily with the *development* of a high-level design description from available low-level design information and the *determination* of variable correspondences between the descriptions. Verification techniques are not designed to make these determinations. This fact does not imply that some of the same heuristics that "reason about design intent" in verification algorithms

a nondeterministic Turing Machine in polynomial time), with the additional property that it is also NP-hard. Thus, a solution for one NP-complete problem would solve all problems in NP. There is always a polynomial-time algorithm for transforming an instance of any NP-complete problem into an instance of an other NP-complete problem. Therefore, if you could solve one, you could solve any other by transforming it to the solved one (Martin, 1997).

could not be made useful in some reverse engineering approaches. However, it is not immediately obvious how to make these heuristics useful when the only reliable design information available is that extracted from a working device. Once a design has been reverse engineered, of course, verification techniques can be employed to validate the recovered design.

1.2 Motivation

Reverse engineering has been defined as the act of creating a set of specifications for a hardware component, primarily as a result of analyzing an existing device. In his groundbreaking paper on reverse engineering, Rekoff (1985, page 244) states that:

> Reverse engineering might seem to be an unusual application of the art and science of engineering, but it is a fact of everyday life. Reverse engineering may be applied to overcome defects in or to extend the capabilities of existing apparatus. Reverse engineering is practiced by the General Motors Corporation on Ford Motor Company products (and visa versa) to maintain a competitive posture. Reverse engineering is practiced by all major military powers on whatever equipment of their antagonists that they can get their hands on. Reverse engineering might even conceivably be used by major powers to provide spare parts and maintenance support to smaller powers who are no longer friendly with the original manufacturers of the weapons they have in their inventory.

We now consider some current problems in reverse engineering that motivate our research. As an example, we present an existing reverse engineering problem being investigated by both government and industry. We then describe some state-of-the-art reverse engineering technologies and focus on how they apply to this problem. An understanding of state-of-the-art reverse engineering technologies is necessary to provide a framework in which the significance of the research presented in this dissertation is clear.

1.2.1 Legacy system reengineering

A significant motivating factor for reverse engineering is the critical need within industry and government agencies to sustain, maintain, and continually modernize systems being used in the field (legacy systems). Most companies that build and maintain fleets of high-cost, long-lived, electronic-dependent systems face the problem of proper documentation, continuous upgrades, and documentation retrieval (Dukes et al., 1994). To meet the necessary demands of form, fit, function, interface (F^3I), and performance, computer-aided engineering (CAE) tools and techniques must be developed to reengineer these legacy systems with state-of-the-art technology. Section 1.2.2 discusses this problem in detail.

There are substantial hindrances to implementing successful design recovery. Systems are often a blend of digital, analog, and software components. Many sources of system data might be available, such as the physical hardware, software source code, test program sets, manufacturing artwork, paper documentation, and data from obsolete design tools. With all these potential sources of information, however, the problem remains that some of the system data might be contradictory. Moreover, even though staggering amounts of information may be available, it still might not be possible to completely specify a system. That is, portions of the system may be known only as a "blackbox".

Legacy systems often lack full documentation regarding the functional roles that all components played and how those roles were met. Modern design has emphasized the need to understand how a system's functionality is achieved as the result of

the interplay of its individual components. In the domain of digital system design, the use of representations such as VHDL have become a necessity (Dukes et al., 1994). A representation of the functionality of a system's components is the key to understanding not only the overall system's behavior but also the roles played by the system components. When adequate documentation about functional roles and how they are met is not available, replacement of a system means a costly *de novu* design of the complete system.

Determining the functional roles of the existing system components is the starting point for reengineering. Delineating these roles allows for a number of possible outcomes that might not otherwise be possible:

- Maintenance of the existing system;
- Replacement of one or more system components as technology advances or as current components become unavailable;
- Verification that the reengineered system meets the intended behavioral specification of the system; or
- Determination of current processing bottlenecks and the components responsible for the them, which thus become candidates for redesign to enhance performance.

Design recovery should be viewed as a vital step in the reengineering process, capable of providing error-free retroactive documentation of an existing digital system.

1.2.2 DoD's obsolete component problem

Microelectronic components are the enabling technology for "smart" systems that have become prevalent in critical systems. Such smart systems are used throughout the government and industry; for example, these systems control weapon deployment in military aircraft, temperature levels in nuclear power plants, instruments on NASA satellites, etc. Because of the important role these components play in vital systems, they are subject to exhaustive and expensive testing.

The increasing pace of technological advances currently causes a turnover in fabrication technology every 18 months (REW'98, 1998). As old process lines close, replacement parts for tested systems become unavailable, forcing new product development and another complete testing cycle. This constant redesign and retesting has become a billion dollar problem in the Department of Defense (DoD) alone.

At the beginning of the digital revolution, the federal government made up a large portion (approximately 30%) of the market. This market dominance has fallen sharply over the last several years. In fact, DoD's market share has fallen to less than 1% in 1997. This diminished manufacturing sources (DMS) situation has caused DoD to consider new ways to reengineer existing, tested systems for new technologies thereby achieving low-price refabrication without resorting to an expensive redesign and retesting cycle.

Unfortunately, government organizations often do not have the resources available to purchase complete documentation for microelectronic systems. And often, even when such documentation does exist, it is out of date, incomplete, or does not ac-

curately represent engineering change orders and last minute modifications made to the device in service. Therefore, during the reengineering process, DoD requires the extraction the necessary redesign information from the existing device.

Current DoD remanufacturing methodologies are not significantly automated, but are reported to work reasonably well on some low complexity designs of older digital families (TTL, ECL, Metal Gate CMOS), analog microcircuits, and hybrids (REW'98, 1998). Su and Dukes proposed techniques to automate the process of identifying a component netlist of the wiring connections between components recognized in the digitized schematics of VLSI systems and to then generate a structural or functional VHDL model of the system (Su et al., 1994; Dukes et al., 1994). However, existing methodologies lack the automation necessary to perform well (or in some cases, at all) when applied to undocumented medium to high complexity digital designs, particularly in cases in which the system interface and functional requirements are not fully documented (REW'98, 1998).

Although automation in reverse engineering of digital hardware has so far been largely ignored in the mainstream literature, it is obviously an important and significant issue. This dissertation proposes several techniques that we hope will significantly further the state of the art in this field.

1.2.3 State-of-the-art design recovery

Attempts to recover a design from documentation alone are error prone, since engineering change orders (ECOs) and last-minute decisions are often not addressed in

the available documentation. To reengineer a legacy device with fidelity, it is often necessary to recover design information from an existing system. This system may contain errors not specified in the original design, a situation that further complicates the reengineering process. In many cases, the only information available about a system is that which can be extracted from a final implementation.

In several ways, the reverse engineering process is the complement of the synthesis tasks described in Section 1.1.2. Table 1.1 summarizes the state of the art in design recovery technologies as identified in the 1998 Reengineering Workshop sponsored by Argonne National Laboratory (REW'98, 1998). Findings from that workshop and the defined "levels" of the design recovery process are presented here.

Image Acquisition $(0 \rightarrow 1)$

When design information is being recovered from an existing system, the first step required is to extract the design layout of the chip or chips which make up the system. A standard technique for the extraction of layout geometry information is to expose a layer though destructive etching (which must take place at a fabrication facility) and then to capture a series of high-resolution images (micrographs) of the exposed layer with a field scanning electron microscope (SEM). To examine an entire device at high resolution, it is necessary to collect a series of images and assemble them (a process referred to as mosaicing) to form a collage of micrographs that acts as a large-scale mural of the device in question. Each layer of the chip is exposed (via etching), in turn, for imaging. Etching is difficult and expensive, since the etching process used is dependent upon the fabrication technology which was used to produce the chip.

Because of the high magnification required, only a small portion of the device can

Level	Technologies	Technical Challenges
0	*Sample Preparation*	
	Etching	Etching
1	*Image Acquisition*	
	SEM Image processing BMP to GDS-II	Accuracy Geometry Staging Unconventional technology
2	*Geometric Description*	
	Postprocessing Design rule checkers	Process information
3	*Transistor Netlist*	
	Syntactic matching Semantic matching Semantic matching	Exact models of gates Unconventional technology
4	*Gate Netlist*	
4a	*Layout*	
	Pattern matching Syntactic matching Semantic matching Contextual matching Optimization tools	Combinatorics Optimizations Library support Incomplete information Clustering Function-centric naming
4b	*Timing*	
	Simulation and modeling Technology-specific information	Unconventional technology
5	*Register Transfer*	
	Model generation Sequential functionality Timing Domain-specific information	Complexity Validation Automating process
6	*Behavioral*	

Table 1.1: **Reverse engineering technologies and challenges, 1997.**

be imaged at a time. Therefore, a series of images must be taken, one at a time, to capture the geometry of the entire layer. The staging precision of current SEMs is not accurate enough to guarantee that consecutive micrographs are precisely adjacent. During the mosaicing process, consecutive images that overlap must be identified and smoothed. Consecutive images that contain a gap in the field boundaries must be identified and filled in. Because it is difficult to position the chip so that it is completely level, this process is further complicated by the fact that various portions of the chip surface require different foci.

The chip scanner represents a state-of-the-art system for image acquisition. It consists of an SEM, a highly accurate stage, and mosaicing software that aligns each image to the next, so overlaps and gaps in the field boundaries do not distort the overall view. The chip scanner micrographs are stored as a series of bitmap data (BMP) files.

Geometric Description $(1 \rightarrow 2)$

After an aligned, high-resolution image is acquired, geometric data must be extracted from it. This process is currently performed though the use of chip scanner software. This software converts bitmap images into a geometric data stream (such as Cadence's GDSII format) that contains position information. This process works well for some conventional technologies but is subject to complications when used with unconventional technologies. Furthermore, chip scanners allow the positions of any additional images to be specified though the use of position information. This allows information from areas of obvious error to be obtained. However, although image acquisition is not necessarily 100% complete and accurate, a high degree of accuracy is

often possible, given sufficient time and samples. The feasibility of automatically extracting mask layer information from scanned images of digital electronics has been well demonstrated (Augustus, 1990; Fretheim, 1988; Hayden, 1989; Mueller, 1989; Querns, 1989).

Transistor-Level Description $(2 \rightarrow 3)$

The next goal is to translate the geometrical level description of the device to the transistor level. This goal is accomplished through the use of commercially available CAD tools (design rule checkers) that can examine geometric data and recognize physical structures (such as transistors, resistors, and the like). Furthermore, design rule checkers may report possible errors in the description, improving the quality of the geometric description.

Gate-Level Netlist Layout $(3 \rightarrow 4a)$

The next step in the design recovery process is the determination of the the gate-level netlist. The current approach relies upon finding known transistor-level implementations of logic gates within the transistor-level description. Currently, this process is carried out by software which performs syntactic (or structural) matching. *Syntactic matching* techniques, such as the University of Washington's subgemini algorithm (Ohlrich et al., 1993), require extensive and complete libraries and are subject to failure when implementations of gates are unconventional. Because modifications are commonly made to the transistor-level implementations of gates to affect power levels, timing, and other design issues this process is not necessarily able to identify all gate-level devices.

Gate-Level Netlist Timing ($3 \rightarrow 4b$)

In addition to obtaining the design layout of the gates, the extraction of timing information has been identified as a crucial step in the design recovery process. Since timing is based on a variety of factors (including doping levels, parasitics, and delays in interconnects), the extraction of gate-level timing information from a transistor-level netlist has not yet been studied extensively. It is believed, however, that this problem will not pose a significant challenge.

Register-Transfer-Level Devices Description ($4a \rightarrow 5$)

The task of identifying RTL modules from a gate-level description of a device is exceptionally complicated. Initial approaches attempt to syntactically match RTL devices to specific gate-level implementations of those devices as defined in a library. These structural approaches, although efficient, are limited by the completeness of the library and the optimization techniques used in the design of the original device. More general *semantic matching* techniques are required to allow the identification of block-level modules whose implementation is not known *a priori*. Furthermore, the identification of RTL modules in systems for which complete information is not available is particularly challenging.

The transformation from the gate-level to the register-transfer-level has been identified as the current critical area of research in the design recovery process. This dissertation focuses primary on research related to this transformation. Chapters 3 and 4 discuss an approach to the challenge of incomplete information during this transformation. Chapter 5 discusses a technique for semantic equivalence checking which allows efficient semantic matching during this transformation.

Register-Transfer-Level Timing Description ($4b \rightarrow 5$)

The transformation of a device's gate-level timing description to the RTL timing description is currently unexplored. It is believed that accurate, descriptive information can be obtained for each RTL device identified through the use of the gate-level timing information obtained in level 4b. The RTL devices with timing information can be represented in any hardware description language (HDL) that includes timing descriptions. VITAL (VHDL Initiative for Timing Annotated Libraries) was proposed as an appropriate HDL.

Behavioral-Level Description ($5 \rightarrow 6$)

The final step in the design recovery process is to construct a complete and functional behavioral-level model of the device under study. Such a model can be validated and used to specify the design of a new implementation of the device. The problem of transforming a description from the RTL level to the behavioral level is currently open to solution. This transformation may be inherently intractable and therefore difficult to automate; intervention by a human engineer is currently considered necessary.

1.3 Contributions

The research presented in this thesis concerns design recovery for the combinational logic in obsolete components (Section 1.2.2). The transformation of the component design from the gate-level to the RTL has been identified as the current critical area of research in the design recovery process (Section 1.2.3). The goal of this transformation is to produce a RTL description of the obsolete component suitable

for restructuring and reengineering from a recovered gate-level description and any additional available design information. As will be discussed in Chapter 3, one of the key problems with this transformation is that a complete gate-level description of the component is often unavailable. Furthermore, since RTL devices have any number of legal gate-level implementations, this transformation is inherently difficult.

The major contributions of this work are three-fold. First, we discuss a systematic approach for both the representation of the known functionality and the deduction of unknown functionality for a system under redesign. This methodology includes techniques for discovering *don't care* conditions and using available information about the overall behavior of the circuit (such as test set information or out-of-date paper schematics) to reconstruct the intended functionality of the circuit. The objective of this design recovery approach is to allow the engineer to represent both the functionality and the structure of the system, represent additional system knowledge, determine missing information, and detect conflict between available design information and the actual design implementation. This approach is a new mechanism for reconstructing such information efficiently.

Furthermore, we present a semantic matching technique that allows the identification of both standard and nonstandard implementations of RTL devices in a gate-level description. Existing syntactic matching techniques have proven incapable of identifying devices that have been optimized during synthesis, constructed from nonstandard cell libraries, or otherwise obfuscated. A new semantic equivalence checking technique for testing matchings between library devices and gate-level subcircuits is introduced which is significantly more efficient than previous techniques.

Lastly, we propose a preliminary approach to the formal design recovery of obsolete systems. We discuss the capabilities of this approach, as well as its limitations.

The algorithms presented in this dissertation have been implemented in software. Results are presented at the ends of Chapters 4 and 5 to illustrate the utility of these techniques.

1.4 Dissertation outline

The remainder of this dissertation is structured as follows. Chapter 2 presents terms and notation used in this field, commonly used mathematical models for circuits and functions, and commonly defined operations. In particular, it focuses on the capabilities and limitations of the binary decision diagram (BDD).

Chapter 3 discusses the limitations of the traditional BDD representation and presents the structural BDD (SBDD), a specific interpretation of a standard BDD which is used extensively in our approach to design recovery when only partial implementation specifications are available. It discusses unknown (blackbox) portions of combinational circuits and shows how SBDDs can be used to represent multiple Boolean relationships among known circuit structures, especially *don't care* relationships.

Chapter 4 introduces the reengineering from partial specifications (RFPS) problem. It discusses a methodology which allows unknown portions of gate-level description to be specified through the use of SBDD techniques. It also discusses the representation of partial knowledge as SBDD constraints and the effect of *don't care*

relationships on blackbox specification. Furthermore, this chapter presents experimental results from these techniques.

Chapter 5 addresses the problem of identifying high-level components in a gate-level netlist. It discusses existing techniques that perform structural (syntactic) matching, and introduces new functional (semantic) matching techniques. Furthermore, it presents heuristic techniques that help to overcome the factorial search space inherent in semantic matching. Preliminary results are presented to illustrate this technique.

Next Chapter 6 proposes a design recovery approach which incorporates the SBDD and semantic matching techniques presented in Chapters 3-5. It discusses the capabilities and limitations of the proposed approach, as well as discussing its correctness and the the complexity issues inherent in this problem and our proposed solution.

Finally, Chapter 7 concludes the dissertation with a summary of the dissertation and a discussion of future research directions.

Chapter 2

Representation of Boolean relationships

The goal of digital design recovery is to recover the design intent underlying a known or partially specified circuit implementation. This problem is particularly challenging because it requires automated techniques to provide information about the function, purpose, and structure of the existing design and transformations that guarantee that the functionality of the circuit is not compromised. This task is complicated by the fact that many digital design implementations undergo iterative optimizations that can hinder understanding the high-level function of a component in the final implementation. Furthermore, complete specifications of the design may be incomplete or unavailable in many situations.

For any reverse engineering methodology to overcome these complications, it must enable one to recognize the functionality of any component of an implementation in the context of the overall circuit function. This recognition of functionality may be

inherently impossible given an incomplete implementation, but available information may allow the automated deduction of a component's general function. Such an approach requires formal techniques for the representation and manipulation of available information.

This chapter presents the terminology and notation used in formal approaches to digital design and verification. It also presents commonly used mathematical models for circuits and Boolean functions and commonly defined operations. In particular, it defines the binary decision diagram (BDD), which has quickly become the standard mechanism for representing Boolean relationships in tools for design automation. It also discusses the capabilities and limitations of the BDD in the design recovery process.

2.1 Notation and terminology

A system description passes through different levels of design in a hierarchical framework. This framework expresses the behavior of a design as it evolves and becomes more detailed (Chapter 1.1.2). The iterative flow of the design process or life-cycle generally starts with the design specification (*behavioral level*). This design is defined in more detail by using library circuits and glue logic to create a RTL (*register-transfer level*) description. This description is synthesized and optimized to create a *netlist* of gates (*structural level*). This structural-level implementation is the lowest level of design considered in this dissertation.

Descriptions of formal approaches for design synthesis between the RTL and the

gate-level of design use a specific vocabulary and terminology. Relationships at these levels are customarily described by using Boolean algebraic notation. We will use this same terminology when describing reverse engineering approaches between these levels. Our notation follows Bryant (1986) and Mailhot (1991).

2.1.1 Basic notation

Definition 2.1 $\mathcal{B} = \{0, 1\}$ *is the* two-valued Boolean domain. *Boolean variables are denoted by subscripted characters and can take on values from the set* $\mathcal{B}$.

Definition 2.2 *The* Boolean operators *disjunction (Boolean OR), conjunction (Boolean AND), and inversion (Boolean NOT) are represented by the symbols* $+$, $\cdot$ *(or whitespace), and an appended apostrophe (or an overbar), respectively (e.g.* $(x_1 \cdot y_1 + x_j)' = \overline{x_1 y_1 + x_j}$ *).*

Definition 2.3 *Using Shannon's Theorem (Shannon, 1938; Shannon, 1949) a* Boolean function $\mathcal{F}(x_1, \ldots, x_n)$ *is a composition of a finite number* Boolean operators *and variables. A single-output* Boolean function *is a function* $\mathcal{F} : \mathcal{B}^n \rightarrow \mathcal{B}$. *An m-output* Boolean function *is a function* $\mathcal{F} : \mathcal{B}^n \rightarrow \mathcal{B}^m$.

The *phase* of a *Boolean variable* x_i indicates whether the value of x_i is to be used directly or complemented (inverted). A *Boolean function* $\mathcal{F}$ is said to be *unate* in *Boolean variable* x_i if x_i appears always in only one phase in the expression of $\mathcal{F}$. $\mathcal{F}$ is said to be *positive* (or *negative*) *unate* in x_i if only x_i (or x_i') appears in the expression of $\mathcal{F}$. $\mathcal{F}$ is *binate* in x_i if the variable appears in both phases in the expression of $\mathcal{F}$.

Definition 2.4 *The* input vector $\mathbf{x}$ *and* output vector $\mathbf{y}$ *of the Boolean function* $\mathcal{F} : \mathcal{B}^n \rightarrow \mathcal{B}^m$ *are each one element of the domains* $\mathcal{B}^n$ *and* $\mathcal{B}^m$, *respectively.*

In general, the vector variables associated with the inputs of the *Boolean function* $\mathcal{F}_i(x_1, \ldots, x_n)$ are denoted as $\mathbf{x}_i$. When necessary, the jth input of $\mathcal{F}_i$, x_j, $1 \leq j \leq n$ is denoted as $\mathbf{x}_{i,j}$. Similarly, the vector variable associated with the output of the *Boolean function* $\mathcal{F}_i : \mathcal{B}^n \rightarrow \mathcal{B}^m$ is denoted as $\mathbf{y}_i$. When necessary, the jth output of $\mathcal{F}_i$, y_j, $1 \leq j \leq m$ is denoted as $\mathbf{y}_{i,j}$. The output may be denoted simply as y_i when $\mathcal{F}_i$ is a single-output function.

Definition 2.5 *The* image *of* A *under* $\mathcal{F} : \mathcal{B}^n \rightarrow \mathcal{B}^m$ *is the subset of* $\mathcal{B}^m$, *denoted by* $\mathcal{F}(A)$, *where* A *is a subset of* $\mathcal{B}^n$. *The* range *of* $\mathcal{F}$ *is the image of* $\mathcal{B}^n$ *under* $\mathcal{F}$ $(A = \mathcal{B}^n)$.

2.1.2 Application-specific terminology

It is important to be able to discuss the subset of Boolean vector space (*n-space*) that represents the sets of *input vectors* for which $\mathcal{F}$ takes the value 1.

Definition 2.6 *The* satisfying-set *(or* on-set*)* $S_\mathcal{F}$ *of a single-output Boolean function* $\mathcal{F}(x_1, \ldots, x_n)$ *is the set of input vectors* $\{\mathbf{x}_j, j = 1, \ldots, |S_\mathcal{F}|\}$ *of* $\mathcal{F}$ *for which* $\mathcal{F}(\mathbf{x}) = 1$. *The input vectors of the* on-set *are also called* minterms. *The* off-set *of a Boolean function* $\mathcal{F}$ *is defined as the set of input vectors* $\{\mathbf{x}_j, j = 1, \ldots, 2^n - |S_\mathcal{F}|\}$ *of* $\mathcal{F}$ *for which* $\mathcal{F}(\mathbf{x}) = 0$.

Definition 2.7 *The* restriction *of* $\mathcal{F}(\mathbf{x})$ *with respect to* x_i, *denoted as* $\mathcal{F}\,|_{x_i=c}$, *is*

$\mathcal{F}(x_1, \ldots, x_{i-1}, c, x_{i+1}, \ldots, x_n)$. *The restriction of* $\mathcal{F}(\mathbf{x})$ *is referred to as the* positive cofactor *of* $\mathcal{F}$ *when* $c = 1$ *and as the* negative cofactor *of* $\mathcal{F}$ *when* $c = 0$.

Definition 2.8 *The* Shannon expansion *(Shannon, 1938) of* $\mathcal{F}$ *around* x_i *represents the equality* $\mathcal{F} = x_i \cdot \mathcal{F}|_{x_i=1} + \overline{x_i} \cdot \mathcal{F}|_{x_i=0}$.

Definition 2.9 *The* smoothing *of* $\mathcal{F}(x_1, \ldots, x_n)$ *with respect to a variable* x_i *is* $\mathcal{S}_{x_i} = \mathcal{F}|_{x_i=1} + \mathcal{F}|_{x_i=0}$.

The smoothing of a function with respect to a variable corresponds to dropping that variable from further consideration. Informally, it corresponds to deleting all appearances of that variable (De Micheli, 1994).

Definition 2.10 *The* true support *or* dependence set *of a Boolean function* $\mathcal{F}(x_1, \ldots, x_n)$, *denoted* $I_{\mathcal{F}}$, *is the subset* $\{x_i | \mathcal{F}|_{x_i=0} \neq \mathcal{F}|_{x_i=1}\}$ *of the set of variables* $\{x_1, \ldots, x_n\}$ *used in the expression of* $\mathcal{F}$. $\mathcal{F}$ depends *on a variable* x_i *if and only if* x_i *is a* support *(*$x \in I_{\mathcal{F}}$*) of* $\mathcal{F}$.

Definition 2.11 *The function* $\mathcal{F}|_{x_i=\mathcal{G}}$ *is defined to be the* composition *of Boolean functions* $\mathcal{F} : \mathcal{B}^n \rightarrow \mathcal{B}$ *and* $\mathcal{G} : \mathcal{B}^n \rightarrow \mathcal{B}$, $\mathcal{F}(x_1, \ldots, x_{i-1}, \mathcal{G}(x_1, \ldots, x_n), x_{i+1}, \ldots, x_n)$.

2.1.3 Incomplete Boolean functions

Boolean functions can be *completely* or *incompletely specified.* Completely specified functions follow the previous definitions. Incompletely specified functions have their domain and range extended to the *augmented Boolean domain.*

Definition 2.12 $\tilde{\mathcal{B}} = \{0, 1, d, u\}$ *is the* augmented Boolean domain, *where d means either 0 or 1 (a* don't care*) and u represents unknown.*

Definition 2.13 *The assignment of the value d to the Boolean variable x_i in an incompletely specified Boolean function implies that the Boolean value of x_i does not affect the output of the function. The value of x_i may be either 0 or 1; we say that x_i is a* don't care.

Definition 2.14 Don't care sets *represent the conditions under which an incompletely specified Boolean function takes the value d.*

Don't care conditions occur in Boolean logic either because some combination of inputs of a Boolean function never occur (the domain of $\mathcal{F}(x_1, \ldots, x_n)$ is smaller than $\mathcal{B}^n$), or because some outputs of $\mathcal{F}$ are not observed. The first class of *don't care* conditions is called *controllability don't cares*, and the second class is called *observability don't cares.*

Definition 2.15 *The assignment of the value u to the Boolean variable y_i in an incompletely specified Boolean function implies that the value of y_i is* unknown. *Although the value of y_i is not specified, it is* **not** don't care. *Neither 0 or 1 can be assigned to y_i unless there is additional information to determine the appropriate assignment.*

2.1.4 Representing combinational logic

Definition 2.16 *A* Boolean network $\mathcal{N}$ *is an ensemble of Boolean functions. A Boolean network is represented by a set of N Boolean variables $\mathcal{V} = \{y_1, \ldots, y_N\}$*

and a set of Boolean functions $\{\mathcal{F}_1, ..., \mathcal{F}_N\}$ *such that* $\mathcal{N} = \{y_i = \mathcal{F}_i, i = 1, \ldots, N\}$, *where* $y_i = \mathcal{F}_i$ *represents an assignment of a single-output Boolean function for every Boolean variable. Functions* $\mathcal{F}_i, i = 1, \ldots, N$, *have* $K_i \leq N$ *inputs (*i.e., $\mathcal{F}_i : \mathcal{B}^{K_i} \rightarrow \mathcal{B}$*), with each input corresponding to a Boolean variable of* $\mathcal{V}$.

Subsets of Boolean networks are also Boolean networks. Boolean networks are represented by graphs $G(V, E)$ where the vertex set V is in one-to-one correspondence with the set of Boolean variables $\mathcal{V} = \{y_1, \ldots, y_N\}$ and where E is the set of edges $\{e_{ij} | i, j \in \{1, \ldots, N\}\}$ such that e_{ij} is a member of the set E if $y_i \in \text{support}(\mathcal{F}_j)$. Such networks are acyclic by definition.

Definition 2.17 *The in-degree of a vertex in a Boolean network is referred to as the variable's* fanin. *Similarly, the out-degree of a vertex in a Boolean network is referred to as the variable's* fanout. *The set of all variables represented by vertices that are reachable from the vertex representing variable* y *is* y*'s* transitive fanout. y*'s* transitive fanin *is the set of all variables which contain* y *in their transitive fanout.*

Definition 2.18 Primary inputs *are Boolean variables of a Boolean network that depend on no other variables (i.e.,* $\mathcal{F}_i$ *is the identity function for* primary inputs*).* Primary outputs *are Boolean variables on which no other variable depends.*

Definition 2.19 *The* Boolean behavior *of an* n*-input,* m*-output Boolean network is the Boolean function* $\mathcal{F} : \mathcal{B}^n \rightarrow \tilde{\mathcal{B}}^m$ *that corresponds to the Boolean network.*

A Boolean network with Boolean behavior represented by a set of completely specified Boolean functions is *fully specified.* Conversely, a Boolean network whose

Boolean behavior is represented by one or more incompletely specified functions and that contains one or more unknowns is only *partially specified.*

A combinational circuit describes a fully specified Boolean network and, therefore, has a set of associated Boolean functions that represent its Boolean behavior. A partially specified combinational circuit describes a partially specified Boolean network that contains partially specified Boolean behavior. A function which represents the behavior of any structure in a combinational circuit is dependent upon the values assigned to variables in its fanin. Thus the only variables upon which the value of a variable representing a structure in a combinational circuit depends on are those in its transitive fanin.

2.2 Decision diagrams

A number of forms have been suggested for representing Boolean functions. The truth-table is one such representation. Another is the *decision diagram*, introduced by Lee (1959).

Decision diagrams represent a function as a directed acyclic graphic (DAG). There are two types of nodes: terminal nodes and decision nodes. *Terminal nodes* have no children and contain values corresponding to a possible output of the function. Terminal nodes are generally represented graphically as rectangles or simply as their value label. *Decision nodes* are labeled by a variable identifier and have one outgoing labeled arc for each possible value that the variable may be assigned. Decision nodes are generally represented graphically as circles containing their variable identifier. A

decision node with no incoming arc is called a *root node.*

The value of the function for some variable assignment can be determined by traversing the graph beginning at the root node for the expression in question. At each decision node, the arc whose label corresponds to the value of the variable assigned to that node is followed. The terminal node reached in this way contains the value of the represented expression under that variable assignment.

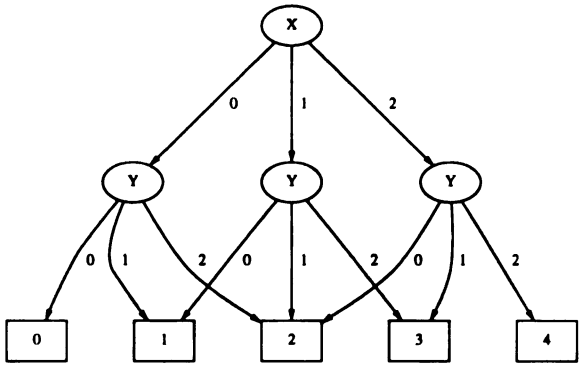


Figure 2.1: **A simple decision diagram.** This decision diagram represents the algebraic function $x + y$ where $x, y \in \{0, 1, 2\}$.

Decision diagrams, such as that depicted in Figure 2.1, are generally referred to as multi-valued, multi-terminal decision diagrams (MDDs or MTDDs) (Minato, 1996). MDDs can deal with arbitrary functions, but the size of the graph quickly becomes unmanageable for complicated functions. Because most design automation problems deal only with Boolean functions, this general data structure can be optimized for the manipulation of Boolean symbols.

2.2.1 Binary decision diagrams

The binary decision diagram (BDD) was introduced by Akers (1978) as a decision diagram form to represent Boolean functions. BDDs are decision diagrams in which there are exactly two terminal nodes labeled "0" (the *0-terminal*) and "1" (the *1-terminal*) and in which each decision node has exactly two outgoing edges: an arc labeled "0" (a *0-edge*), and an arc labeled "1" (a *1-edge*). Therefore each decision node N has exactly two children: $N.0$ (the child along N's 0-edge) and $N.1$ (the child along N's 1-edge).

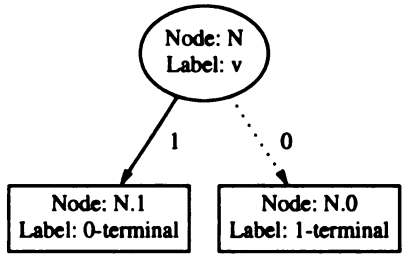


Figure 2.2: **Simple BDD.** A BDD G representing the function: $f_N = v \cdot 0 + \overline{v} \cdot 1$

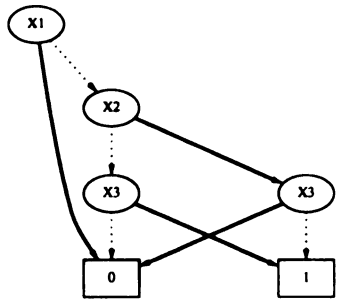
Consider a BDD G consisting of a root decision node N, labeled with the variable v, and N's two children. Let $N.0$ be the 1-terminal and $N.1$ be the 0-terminal. The BDD G (show as Figure 2.2) then represents the function:

$$\begin{aligned} f_N &= v \cdot 0 + \overline{v} \cdot 1 \\ &= \overline{v}. \end{aligned}$$

In general, the function of any node N, labeled with the variable v, can be defined recursively in terms of its two children, $N.0$ and $N.1$, by using its *Shannon expansion* (Definition 2.8) as follows:

$$f_N = v \cdot f_{N.1} + \overline{v} \cdot f_{N.0}. \tag{2.1}$$

Figure 2.3 illustrates a BDD along with the function and truth-table that the BDD represents. Each node has two outgoing edges, a 1-edge (shown as a solid arc) and a 0-edge (shown as a dotted arc). The functional value for any variable assignment is determined by traversing the path from the root node to a terminal node by following the appropriate branch at each decision node.



(a)

$$F_2 = \overline{X_1}(\overline{X_2}X_3 + X_2\overline{X_3})$$

X_1	X_2	X_3	F_2
1	d	d	0
0	1	1	0
0	1	0	1
0	0	1	1
0	0	0	0

(b)

Figure 2.3: **BDD and functional truth-table.** A solid (dotted) arc represents the branch taken when the decision variable is 1 (0).

2.2.2 Reduced ordered binary decision diagrams

BDDs were not widely used until a set of efficient algorithms for BDD implementation was introduced by Bryant (1985) . Shannon (1949) asserted that most Boolean functions require exponential size when represented graphically. Bryant demonstrated, however, that under reasonable restrictions, BDDs could efficiently represent a wide range of Boolean functions.

Bryant's BDD implementation requires that the BDD be *ordered* and *reduced.* Each decision variable in an ordered BDD (OBDD) must obey an ordering restriction that requires it to appear once, at most, on any path from the root node to a terminal node and in a sequence defined by a total ordering of the variables. Furthermore, as a result of eliminating and sharing isomorphic subgraphs within an OBDD, the BDD is reduced to a compact form.

Following Bryant (1986) we now formally define reduced, ordered BDDs:

Definition 2.20 *A binary decision diagram (BDD) is a directed acyclic graph consisting of two types of nodes. A* nonterminal *node v (also referred to as a* decision *node) is represented by a 3-tuple $\langle index(v), child_l(v), child_r(v)\rangle$, where $index(v) \in \{0, 1, \ldots, n-1\}$, and $child_l(v)$ and $child_r(v)$ are themselves nodes of the BDD. A* terminal *node v is represented by a 2-tuple $\langle index(v), value(v)\rangle$, where $index(v) = n$ and $value(v) \in \{0, 1\}$.*

Definition 2.21 *A BDD is* ordered *if for every nonterminal node v, $index(v) < index(child_l(v))$ and $index(v) < index(child_r(v))$.*

Definition 2.22 *A BDD is* reduced *if there is no nonterminal node v such that $child_l(v) = child_r(v)$ (redundant nodes) and if there are no two nonterminal nodes u and v such that $child_l(u) = child_l(v)$ and $child_r(u) = child_r(v)$ (isomorphic nodes).*

The ordered, reduced form is canonical and identical for equivalent functions (Bryant, 1985). Although properly referred to as a reduced ordered binary decision diagram (ROBDD), the term BDD has generally come to imply a ROBDD implementation. The BDD presented in Figure 2.3(a) is an ROBDD for the function presented

in Figure2.3(b). All BDDs discussed in the remainder of this dissertation will be implemented as ROBDDs.

Boolean operations such as logical AND, OR, and equality can be applied to functions represented as BDDs by using BDD manipulations. Bryant presented a suite of algorithms implementing these operations which manipulate the BDD representations of two functions and return the BDD representing the resulting function. Most BDD implementations manipulate BDDs by using the conventional BDD **Apply** algorithm (Figure 2.4) Bryant introduced. Through the use of dynamic programming techniques (used to cache intermediate results), these operations have an average time complexity almost proportional to the size of the BDD (Bryant, 1985).

Figure 2.4 provides the details of the conventional depth-first BDD **Apply** algorithm. Step 1 illustrates the base case of the recursive algorithm. If both BDD inputs are terminal nodes, then the operation is performed on their values, and the terminal node corresponding to the result of that operation is returned. Step 2 is a dynamic programming step: if the routine has already processed the two input BDDs, it looks up the result calculated and returns that BDD. Steps 3-6 create the BDD result of the **Apply** operation through recursion. Step 7 removes the root of the result if the value of its decision variable has no effect on the functional output of the BDD; that is, it removes irrelevant nodes. Step 8 checks to see if the computed BDD is isomorphic to any BDD created earlier in the recursive process. If it is, the operation uses the previously calculated BDD (removing isomorphisms). If it is not isomorphic, the operation returns the computed BDD after storing it for future use in Step 9. This algorithm has a worse case time complexity of $O(|F||G|)$, but the expected time

complexity is $O(|F| + |G| + |Result|)$ (Bryant, 1986).

```
Apply (binary_operation op, bdd F, bdd G)
begin
  1. if (is_terminal(F) and is_terminal(G)) then
     return terminal(F op G).
  2. if (previously_done(F, G)) then
     return table_lookup(F, G).
  3. v = top_variable (F, G);
  4. result.0 = Apply(op, F|v=0, G|v=0);
  5. result.1 = Apply(op, F|v=1, G|v=1);
  6. result = make_bdd(v,result.0,result.1);
  7. if (result.0 = result.1) then
     return result.0.
  8. if previously_created(result) then
     return previous_bdd(result).
  9. save_in_bdd_table(result);
  return result.
end.
```

Figure 2.4: **Recursive BDD Apply algorithm (Ranjan et al., 1996).**

The BDD has been of significant interest in the design automation field, and results of ongoing research regularly improve the efficiency of this representation. Researchers in parallel and distributed systems have recently developed breadth-first algorithms to implement BDDs of 15×10^6 nodes and more (Ranjan et al., 1996). Likewise, *multi-rooted* or *shared* BDDs have been proposed that allow multiple BDDs representing multiple functions to share subgraphs (Minato et al., 1990). Figure 2.5 illustrates a *shared* BDD for the function of the sum and carry outputs of a two-bit

full adder.

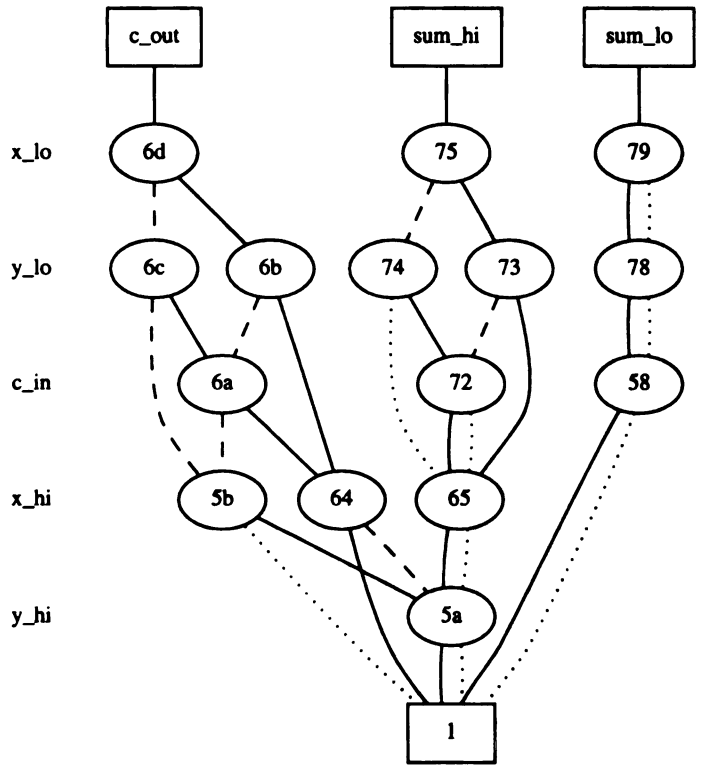


Figure 2.5: **Shared binary decision diagram for two-bit adder.** This multi-rooted BDD represents the function performed by a two-bit adder. In this representation (Somenzi, 1997), the label of each node is a unique random name. All nodes of the same level correspond to the same variable, whose name is shown at the left of the diagram. Solid lines indicate *then* arcs (i.e., 1-edges). Dashed lines indicate *else* arcs (i.e., 0-edges). Dotted lines indicate *complemented else arcs* and negate the value of the terminal.

2.3 Applications of binary decision diagrams

Although the basic idea of the BDD has been around for more than 30 years, its was only recently that canonical representation and efficient implementation made

BDDs a useful tool for CAD (Brace et al., 1990). BDD researchers have shown that polynomial size BDDs exist for a wide class of practical circuits (Fujita et al., 1988; Madre and Billon, 1988; Malik et al., 1988; Brace et al., 1990). The success of BDDs as tools for solving seemingly intractable problems has motivated research into variants of the basic data structure which aid in solving a variety of heretofore impractical applications. An important subset of BDD research areas are listed in Table 2.1.

Research Topic	Reference
Algebraic decision diagrams	Bahar et al., FMSD'97
Applications to polynomial algebra	Minato, IWLS'95
Asynchronous circuit synthesis	Lin et al., ICCAD'94
Binate covering problem solver	Jeong et al., ICCAD'92
Breadth-first manipulations	Ashar et al., ICCAD'94
Dynamic variable reordering	Rudell, ICCAD'93
Exact and approximate FSM traversal techniques	Coudert et al., ICCD'90
Efficient equivalence checking	Matsunaga, DAC'96
Formal verification of arithmetic circuits	Bryant et al., DAC'95
ILP solver based on edge-valued BDDs	Pedram et al., ICCAD'93
Implicit prime generation and two-level minimization	Coudert et al., DAC'93
Implicit set representation in combinatorial problems	Minato, DAC'93
Matrix representation using MTBDDs	Clarke et al., IWLS'93
Multi-valued decision diagrams	T. Kam's M.S. Thesis, Berkeley'90
Parallel algorithm for BDD construction	Kimura et al., ICCD'90
Symbolic synthesis techniques	B. Lin's, Ph.D. Thesis, Berkeley'91
Timed binary decision diagrams	Li et al., ICCD'97

Table 2.1: **BDD research areas.**

Many automated approaches to hardware verification, such as model-checking,

rely on BDDs or some variant as the underlying representation. Because of their compactness and efficiency, the use of BDD forms has become essential in approaches to a variety of CAD problems, especially those using formal methods for design and verification. The design automation community is using BDDs ubiquitously in formal verification, logic syntheses, test generation, and simulation (Sentovich, 1996; Bryant, 1995). It is for these reasons that we chose to use BDD-based techniques in the research presented in this dissertation.

2.4 Decision diagrams as proposition testers

A BDD can be viewed as describing two complementary subsets of an n-dimensional Boolean search space: one in which the decision variable assignments produce a 0 (false) function value, and one in which the variable arguments produce a 1 (true) function value. By creating a function $\mathcal{F}$ whose *satisfying set* $S_{\mathcal{F}}$ (Def. 2.6, p. 28) contains all assignments of the variables' arguments that satisfy some property, BDDs can be used to test any proposition.

Figure 2.6a shows the BDD representation of the Boolean proposition $((a \vee b) \wedge (c \vee d))$. Logical conjunction/disjunction representation is used to emphasize that this is a proposition, not simply a Boolean function. Each element of the satisfying set describes a unique path beginning at the root node and ending at the 1-terminal. As long as at least one such *1-path* exists, the proposition is satisfiable. In this example, the satisfying set of 1-paths (a,b,c,d) is: $S_{\mathcal{F}} = \{(1, d, 1, d), (1, d, 0, 1), (0, 1, 1, d), (0, 1, 0, 1)\}$.

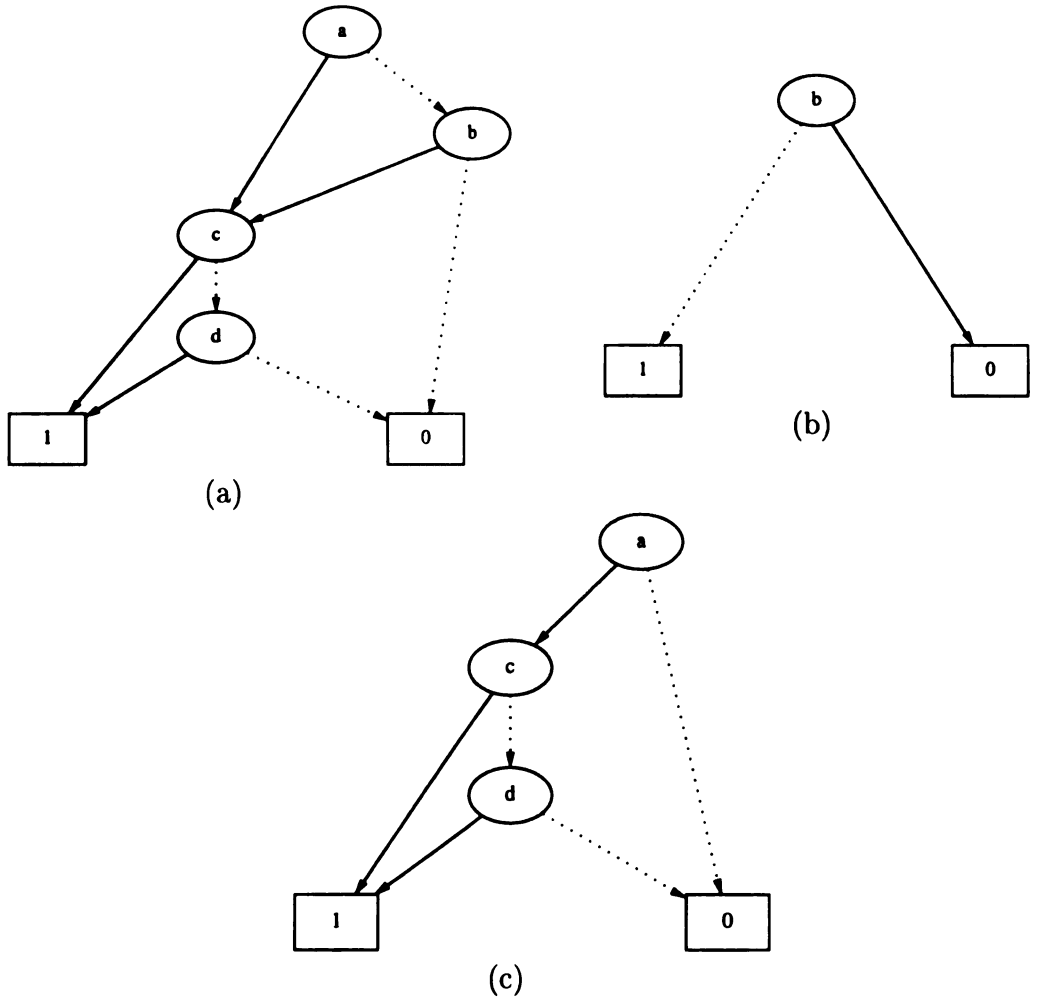


Figure 2.6: **Using BDDs to represent constraints.** BDDs representing (a) the proposition $[(a \vee b) \wedge (c \vee d)]$, (b) the proposition $\bar{b}$, and (c) the proposition $\{[(a \vee b) \wedge (c \vee d)] \wedge \bar{b}\}$, constructed by composing BDDs (a) and (b).

BDDs allow us to test a proposition by applying constraints to the proposition's BDD representation (Baldwin, 1994). For example, let us test our proposition under the constraint that $b = 0$. By creating a BDD for the proposition $\bar{b}$ (Figure 2.6(b)) and using the **Apply** algorithm (Figure 2.4) with the Boolean AND operation to compose the two BDDs, we create a new BDD (Figure 2.6(c)) that represents the original proposition with the constraint that $b = 0$. In this new BDD, in all 1-paths, $a = 1$, a necessary condition when $b = 0$. By examining the 1-paths (a, b=0, c, d) in this new BDD, we can determine each unique variable assignment that results in the proposition being satisfied under this constraint, namely, $S_{\mathcal{F}} = \{(1, 0, 1, d), (1, 0, 0, 1)\}$.

Chapter 3

Representation of structural relationships

To effectively approach the reverse engineering problem, one must be able to formally represent and reason about Boolean relationships encapsulating known and deduced system information. As shown in Chapter 2, a BDD representation of a circuit's function provides a complete and compact representation of the circuit's behavior. It does not, however, encode the relationships among internal structures and circuit outputs that are vital to design recovery.

To successfully reverse engineer a digital component, one may need to make use of relationships defined at the structural level of design. Since a straightforward representation of the circuit's function fails to represent the structures defined in the implementation, a new function that more completely represents the structure as well as the functionality of the circuit needs to be defined.

This chapter presents the *structural binary decision diagram* (SBDD). SBDDs

allow for efficient discovery and compact representation of structural relationships and *don't care* conditions for multiple output functions.

3.1 Motivation for the structural binary decision diagram

The key to representing and integrating full, and partial, functionality of digital components lies in providing an efficient, canonical representation for available system knowledge that can uniformly represent partial information from different levels of design. Such a representation allows the transformation of existing design data obtained from various sources into a representation useful for design automation. Such a uniform representation also simplifies the tasks of recreating the functionality of a partially specified system and detecting conflicting design information.

The problems faced by an approach to such a representation include the following: representing full, and partial, functionality of digital components and integrating these representations; recreating the functionality performed by a system for which only incomplete design information is available; identifying the high-level functionality of known or recovered components; resolving conflicting information about the design; and handling both combinational and sequential circuits/devices.

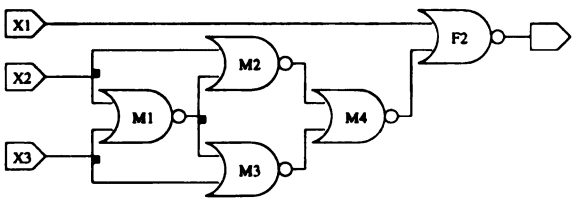
3.1.1 Limitations of traditional representation

As we have seen, the BDD is a key tool for representing functionality. Traditionally, BDDs are used to encode the behavior of a combinational circuit by representing the function of each circuit output in terms of the primary circuit inputs (as shown in Chapter 2). Terminal nodes of a BDD graph represent the logic values that a function takes on for all possible input variable values. Since BDDs are concerned only with functionality, all structure of the circuit is lost. Indeed, it is the canonical form inherent in reduced BDDs that make them most attractive (Lai et al., 1992).

BDD-based techniques constitute a powerful approach to many problems in the design automation field. Although serious efficiency issues are not uncommon in BDD-based techniques, these issues are relatively well known (Bryant, 1995). The traditional BDD representation of a combinational circuit, however, is not an adequate tool for reverse engineering.

Figure 3.1 presents a simple single-output combinational circuit that we will refer to as **simplecircuit**. This circuit is used throughout this dissertation to illustrate basic concepts. Consider the schematic for **simplecircuit** (Figure 3.1(a)) and its corresponding BDD (Figure 2.3, page 35). This BDD representation is the traditional way in which BDDs are used to represent circuits.

This traditional BDD representation provides a complete and canonical representation of the circuit behavior. This representation is of great value, for example, in the field of formal circuit verification. In particular, because BDDs are canonical representations, checking a reference circuit for equivalence with a modified version

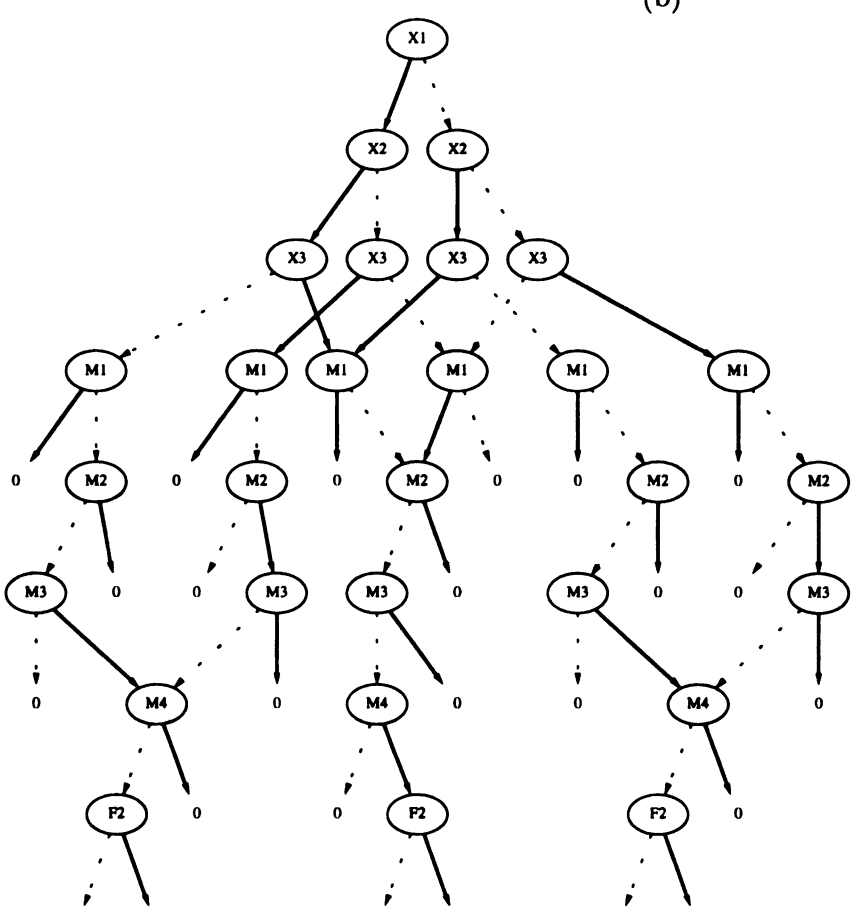


$$F_2 = \overline{X_1}(\overline{X_2}X_3 + X_2\overline{X_3})$$

(a)

X_1	X_2	X_3	M_1	M_2	M_3	M_4	F_2	$\mathcal{X}^S$
1	1	1	0	0	0	1	0	1
1	1	0	0	0	1	0	0	1
1	0	1	0	1	0	0	0	1
1	0	0	1	0	0	1	0	1
0	1	1	0	0	0	1	0	1
0	1	0	0	0	1	0	1	1
0	0	1	0	1	0	0	1	1
0	0	0	1	0	0	1	0	1
			otherwise					0

(b)



(c)

Figure 3.1: **Schematic, structural truth-table, and structural BDD for `simplecircuit`.** (a) Complete schematic and functional description of `simplecircuit`; (b) Truth table for the structure function (Definition 3.3) of `simplecircuit`; (c) BDD representing `simplecircuit's` complete structure function. 1-edges (0-edges) are represented as solid (dotted) arcs.

of the circuit is easily accomplished. However, this representation does not encode the structure or topology of the actual circuit, nor does it provide information about the relationships among the internal gate values and the circuit outputs. A number of combinational circuit implementations can satisfy the behavior specified by the BDD in Figure 2.3. Although this BDD describes the functionality of the circuit with respect to its output, it does not contain any information regarding the configuration of gates within the circuit that provides this functionality. This representation corresponds to a specification from the behavioral level of design (Figure 3.2).

3.1.2 Requirements for new interpretation

In reverse engineering applications, the function of a combinational circuit may not be completely specified. In this case, any available information, including the relationships specified by the topology of the circuit structures, may be of value in determining the effective function of the unspecified component. Therefore, one must depart from the traditional mechanism by which a BDD is used to represent a circuit and define a different method by which the structure as well as the functionality of the circuit may be encoded.

To recover design intent from an incomplete circuit implementation, we must be able to perform deductions on whatever information is made available to us, regardless of the level of abstraction of the information. Thus, recognizing the functionality in an incomplete circuit implementation description requires techniques that close the gap among different levels of abstraction. It also requires algorithms that can

```
ENTITY simplecircuit IS
  PORT( X1,X2,X3: in bit; F2: out bit );
END simplecircuit;

ARCHITECTURE behavioral OF simplecircuit IS
BEGIN
  F2 <= (not X1) and (((not X2) and X3) or (X2 and (not X3))) after 10 ns;
END behavioral;

ARCHITECTURE structural OF simplecircuit IS
  SIGNAL M1, M2, M3, M4 : bit ;
  FOR ALL : nor2 USE ENTITY trace.nor2(behav);
  FOR ALL : probe USE ENTITY trace.probe(behav);
  BEGIN
    gate0 : nor2 PORT MAP ( O => M1, a => X2, b => X3 );
    gate1 : nor2 PORT MAP ( O => M2, a => X2, b => M1 );
    gate2 : nor2 PORT MAP ( O => M3, a => M1, b => X3 );
    gate3 : nor2 PORT MAP ( O => M4, a => M2, b => M3 );
    gate4 : nor2 PORT MAP ( O => F2, a => X1, b => M4 );
    output_F2 : probe;
  GENERIC MAP( "F2", "sim_results/F2" );
  PORT MAP( F2 );
END structural;
```

Figure 3.2: **Behavioral and structural VHDL code for `simplecircuit`.** The behavioral description corresponds to the circuit specification, which is efficiently represented by a BDD. The structural description corresponds to the circuit implementation, which the SBDD (Chapter 3) efficiently represents.

deduce "new" information about function from the partial information available. The result of these techniques should be a compact representation of complete component knowledge for use by redesign tools. Such tools can take advantage of the information so represented to discover or redesign the missing components of the implementation description.

A new interpretation of standard BDDs has been developed that encapsulates the relationships present in the structural description of a combinational logic circuit. Structural binary decision diagrams (SBDDs) allow partial information related to the function and logic structure of a module to be represented in a simple, yet powerful, characteristic function. SBDDs not only retain the desirable qualities of standard BDDs but also include necessary structural information needed to represent a partially specified digital component. Thus, SBDDs correspond to a specification from the structural level of design (Figure 3.2). SBDDs allow for canonical representation of design data and also provide a data structure that allows the partial information represented by the SBDD to be refined as the reengineering process progresses.

3.2 Structural binary decision diagrams

When attempting to reengineer a circuit from partial knowledge, discovering the overall function of the circuit is the primary goal. Achieving that goal, however, may require reconstructing the functionality of missing components within the circuit by making use of structural context. The standard interpretation of BDDs encodes the function of a circuit only in terms of the primary inputs and the outputs of the circuit;

all structural context is lost. However, instead of defining a new data structure, we will define a different *interpretation* of standard BDDs that will encapsulate the relationships present in the structural description of the circuit.

3.2.1 Overview of the SBDD

The primary difference between the SBDD interpretation of a circuit and the standard BDD representation of a circuit is in the interpretation of the terminal nodes. In a traditional BDD interpretation, the terminal labels represent the *value* of the function for the input conditions described by the *path* from the root to the terminal. In an SBDD interpretation, the 1-terminal indicates that the path from the root to the 1-terminal *may* be a legal assignment in the circuit.

A *legal assignment* of variables is defined as an assignment of values to variables that could occur in a circuit that does not deviate from its specified behavior (i.e., a *fault-free* circuit). To illustrate such an assignment, consider a circuit consisting of an OR gate with inputs a and b and an output c. The following assignments (a,b,c) are examples of legal assignments: (1,1,1), (1,0,1), (0,0,0). The following assignments (a,b,c) are examples of illegal assignments: (1,1,0), (1,0,0), (0,0,1). (0,0,1) is an illegal assignment because the structure of the circuit requires that $c = 0$ when both a and b are equal to 0.

SBDDs may contain a variable representing the output of any gate, line, or other logic structure within a circuit description. A 1-path in an SBDD is an assignment of inputs, outputs, and internal structure values that do not contradict anything that

is known about the functional behavior of the circuit. This does not mean that any path to the 1-terminal is a legal variable assignment; it only means that there is no information which indicates such an assignment is illegal. Any path from the root to the 0-terminal (a *0-path*) represents an illegal assignment that contradicts available knowledge about the relationships between the variables and thus cannot occur in a fault-free circuit.

Informally, an SBDD for a digital circuit is a reduced, ordered, binary decision diagram defined as follows:

- Each decision node variable may represent a circuit input, a circuit output, or the value (output) of any structure performing a known or unknown function within the circuit.

- A 1-edge represents a state in which the structure represented by the decision node variable has a value of 1.

- A 0-edge represents a state in which the structure represented by the decision node variable has a value of 0.

- Any variable assignment that is a 0-path (ending in the 0-terminal) is illegal and cannot occur in a fault-free implementation.

- Any variable assignment that is a 1-path (ending in the 1-terminal) does not contradict any fact known about the function or structure of the circuit and *may* occur.

Because of the large number of 0-paths in most SBDDs, graphical representations quickly become cluttered with arcs to the 0-terminal. In this dissertation, all SBDD terminal nodes are represented multiple times and without boxes. In figures, all 1-edges are shown as solid arcs, and 0-edges are shown as dotted arcs. For clarity, arcs leading directly to the 0-terminal may not be represented in figures; any node for which only one outgoing arc exists may be assumed to have the 0-terminal as the target for the missing complementary arc.

3.2.2 Representing structure

The function of a combinational circuit is generally represented as a set of Boolean functions, each of which describes the logical behavior of one of the circuit's outputs. This notation concisely represents the behavioral functionality of the circuit, representing the function only in terms of the circuit's primary inputs and primary outputs. Any number of circuit implementations exist which satisfy this behavioral functionality. We now consider a description that can represent relationships between structural components of a particular implementation.

Definition 3.1 *Consider a combinational circuit A consisting of k internal components. The variables representing the inputs or outputs to any internal component are* net variables *and $\mathcal{V}$ denotes the set of all net variables. The circuit's primary inputs and primary outputs are special net variables and are included in $\mathcal{V}$. In the appropriate context, $\mathcal{V}$ is used to represent an assignment of Boolean values to the set of net variables.*

Definition 3.2 *Each component i in a combinational circuit defines the relationship between the component's n_i inputs and its m_i outputs. Denote component i's* input variable vector *as* $\mathbf{x_i}$ *and its* output variable vector *as* $\mathbf{y_i}$ *where each element in* $\mathbf{x_i}$ *and* $\mathbf{y_i}$ *represents the value of a net variable in* $\mathcal{V}$. *The* function of component i *can therefore be represented as* $\mathcal{F}_i : \mathcal{B}^{n_i} \rightarrow \mathcal{B}^{m_i}, \mathcal{F}_i(\mathbf{x_i}) = \mathbf{y_i}$.

The characteristic function (De Micheli, 1994) of component i, $\mathcal{X}_i^C : \{0,1\}^{|\mathbf{x_i}|+|\mathbf{y_i}|} \rightarrow \{0,1\}$, *is defined to be:*

$$\mathcal{X}_i^C(\mathbf{x_i}, \mathbf{y_i}) = \mathbf{1} \text{ iff } \mathcal{F}_i(\mathbf{x_i}) = \mathbf{y_i}. \tag{3.1}$$

Consider a component labeled j that consists of a single AND gate: $y_{j,1}$ = AND($x_{j,1}$,$x_{j,2}$). Figure 3.3 presents the function f_j and characteristic function $\mathcal{X}_j^C$ of component j.

$x_{j,1}$	$x_{j,2}$	f_j
1	1	1
1	0	0
0	1	0
0	0	0

(a)

$x_{j,1}$	$x_{j,2}$	$y_{j,1}$	$\mathcal{X}_j^C$
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	1
otherwise			0

(b)

Figure 3.3: **Function and characteristic function of a two-input AND gate.** (a) Output function; (b) Characteristic function.

Definition 3.3 *The* structure function *of a k component combinational circuit over the net variables* $\mathcal{V}$ *is defined as* $\mathcal{X}^S : \{0,1\}^{|\mathcal{V}|} \rightarrow \{0,1\}$ *where:*

$$\mathcal{X}^S(\mathcal{V}) = \prod_{i=1}^{k} \mathcal{X}_i^C(\mathbf{x_i}, \mathbf{y_i}). \tag{3.2}$$

The structure function, is a Boolean function in $|\mathcal{V}|$ variables whose value is false only for those assignments of Boolean values to net variables that contradict the functional constraints imposed by the circuit structures. Hence, *the value of the complete structure function is false for net variable assignments that could not be observed in the correctly functioning circuit.*

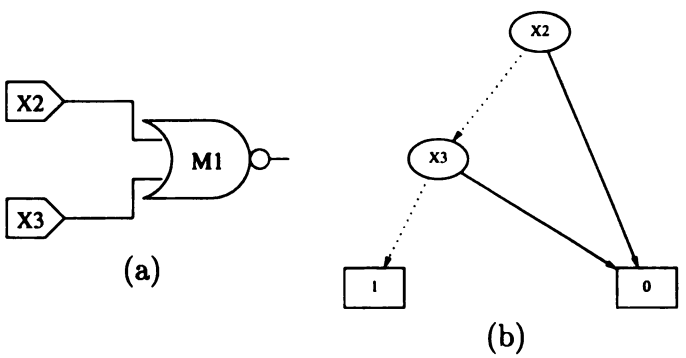
Definition 3.4 *A structural-level circuit implementation can be viewed as a directed, acyclic graph (DAG) in which each vertex represents a structure (usually a gate) and each arc (structure$_1$, structure$_2$) indicates that the output of structure$_1$ is an input to structure$_2$. This DAG forms a partial order on the set of net variables (Definition 3.1) representing the circuit's components. A* structural BDD (SBDD) *is a ROBDD representation of the combinational circuit's structure function (Definition 3.3) with a BDD variable order (Definition 2.21) that does not violate this partial order.*

Consider the combinational circuit `simplecircuit` and its functional specification presented in Figure 3.1(a) (page 47). This behavioral-level specification describes the function of the circuit in terms of its primary inputs and outputs. Figure 3.1(b) presents the truth-table for the implementation's structural function, which describes the circuit in terms of its complete set of net variables. Figure 3.1(c) presents a BDD which represents the structural function for `simplecircuit`; such a BDD is referred to as the *structural BDD* (SBDD) for the circuit. In a completely specified circuit, the values assigned to the primary inputs completely specify the necessary value of all

other net variables. Therefore, for each assignment of input variables in a completely specified circuit, exactly one 1-path exists in its structural BDD.

3.2.3 Single gate example

The schematic shown in Figure 3.4(a) illustrates a subcircuit of **simplecircuit** (Figure 3.1(a)) that consists of a single NOR gate M_1. Figure 3.4(b) illustrates the traditional BDD representation of this circuit. The BDD represents the value of the output M_1. Observe that the output $M_1 = 0$ in this circuit when $X_2 = 1$. This is represented in the BDD by the path $X_2 \xrightarrow{1} 0$. In this traditional representation, the terminal labels of the BDD represent the output value of the function, in this case the gate M_1. Although this representation is sufficient for a circuit consisting of a single gate, there is no means for representing the output value of any additional gates.



$M_1 = \overline{X_2 + X_3}$

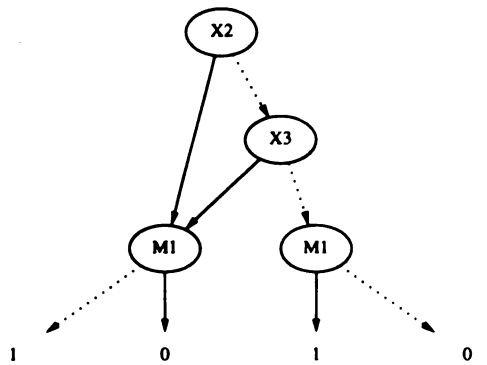
X_2	X_3	M_1
1	d	0
0	1	0
0	0	1

(c)

Figure 3.4: **ROBDD and functional behavior of a NOR gate.**

Consider the SBDD representation of this same one-gate circuit (Figure 3.5(a)). The SBDD representation contains all of the information present in the BDD repre-

sentation. For example, observe that when $X_2 = 1$ in the circuit, the output $M_1 = 0$. This is represented in the SBDD by considering two paths. The path $X_2 \xrightarrow{1} M_1 \xrightarrow{0} 1$, represents that if $X_2 = 1$, then $M_1 = 0$ is a legal assignment. Conversely, the path $X_2 \xrightarrow{1} M_1 \xrightarrow{1} 0$ represents that if $X_2 = 1$, then $M_1 = 1$ is an illegal assignment. Since exactly one legal assignment exists for M_1 when $X_2 = 1$, it can be deduced that the gate $M_1 = 0$ when $X_2 = 1$.



(a)

X_2	X_3	M_1	F'
1	d	0	1
0	1	0	1
0	0	1	1

(b)

Figure 3.5: **SBDD and structural description of a NOR gate.** For clarity, terminal nodes in SBDDs are represented multiple times and without boxes.

Observe that the SBDD interpretation represents the same information present in the BDD representation but that it does so by representing M_1 as a decision node, rather than as the overall function of the BDD. This representation allows us to represent the values of any number of structure outputs within a single BDD representing the circuit's structure function. Figure 3.1(c) presents a SBDD constructed in this way.

3.3 Representing unknown structures

In many cases, as discussed in Chapter 1, complete information regarding the functionality of one or more circuit components is unavailable. Any component whose functionality is not fully specified is called a *blackbox* structure (Wojcik et al., 1997). The outputs of such a structure are referred to as *blackbox net variables.*

Definition 3.5 *Let the behavior of an output $b \in \mathcal{V}$ of a blackbox component i with inputs $\mathbf{x_i}$ be defined by the partial function $\mathcal{F} : \tilde{\mathcal{B}}^{n_i} \rightarrow \tilde{\mathcal{B}}^{m_i}, \mathcal{F}(\mathbf{x_i})$ where $\tilde{\mathcal{B}}$ is used as defined in Definition 2.12. Then the* characteristic function for blackbox output *b is defined to be:*

$$\mathcal{X}^C_{i,b}(\mathbf{x_i}, \mathbf{b}) = \begin{cases} 0, & \text{if } \mathcal{F}(\mathbf{x_i}) \neq \mathbf{b} \\ 1, & \text{if } \mathcal{F}(\mathbf{x_i}) = \mathbf{b} \\ 1, & \text{if } \mathcal{F}(\mathbf{x_i}) = \textit{ undefined} \end{cases} \tag{3.3}$$

By defining the characteristic function of a partially specified component in this way, one can state the following property of the circuit's structure function.

Theorem 3.1 *Let $\mathcal{X}^S$ be the structure function for a combinational circuit A as given in Definition 3.3, where $\mathcal{X}^C$ represents a characteristic function for any structure, including structures specified by a partial function, as defined in Definition 3.5. Then $\mathcal{X}^S(\mathcal{V}) = 0$ only if the assignment of Boolean values to net variables $\mathcal{V}$ is never observable in the functioning circuit A.*

Proof: Assume that A's structure function $\mathcal{X}^S(\mathcal{V}) = 0$ for some assignment of Boolean values to net variables $\mathcal{V}$ that was observable in the func-

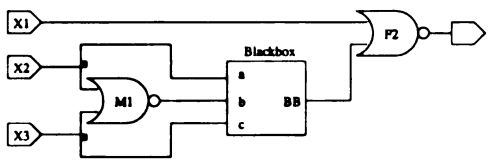
tioning circuit. According to Equation 3.2, $\mathcal{X}^S(\mathcal{V}) = \prod_{i=1}^{k} \mathcal{X}_i^C(\mathbf{x_i}, \mathbf{y_i}) = \mathbf{0}$. Thus, there must exist a component j in A for which $\mathcal{X}_j^C(\mathbf{x_j}, \mathbf{y_j}) = \mathbf{0}$. Therefore, according to Equation 3.3, $f_j(\mathbf{x_j}) \neq \mathbf{y_j}$. Since $\mathbf{x_j}, \mathbf{y_j} \subseteq \mathcal{V}$, this conclusion contradicts the assumption that $\mathcal{V}$ is observable in the functioning circuit, and thus the assumption must be false.

Definition 3.5 provides a mechanism for representing partial knowledge within the SBDD framework. A structure function that includes the characteristic function of a blackbox output may only partially specify the functionality of the combinational circuit it represents. Theorem 3.1 provides a basis from which it is possible to deduce the set of possible relationships that the blackbox may represent. In a structure function including partial knowledge, an assignment of net variables for which $\mathcal{X}^S(\mathcal{V}) = 1$ does not contradict any known relationship, but it is not guaranteed to be the single relationship that represents the observable functionality of the blackbox in the functioning circuit. The single relationship that represents the blackbox can be determined only by eliminating all other relationships from consideration. Formally:

Definition 3.6 *The structure function* $\mathcal{X}^S$ fully specifies *the functionality of the combinational circuit A it represents if for any assignment of A's primary input variables* $\mathbf{x_A} \subseteq \mathcal{V}$, *there exists exactly one assignment of values to the net variables in* $\mathcal{V} - \mathbf{x_A}$ *for which* $\mathcal{X}^S(\mathcal{V}) = 1$. *Otherwise,* $\mathcal{X}^S$ partially specifies *the functionality.*

Figure 3.6(a) presents a partial specification for `simplecircuit`, representing a portion of the implementation as a blackbox. In the SBDD (Figure 3.6(c)) that represents the partial specification's structure function (Figure 3.6(b)), there is no

longer a unique 1-path for every input variable assignment. Thus, this structure function only partially specifies the functionality of the circuit.

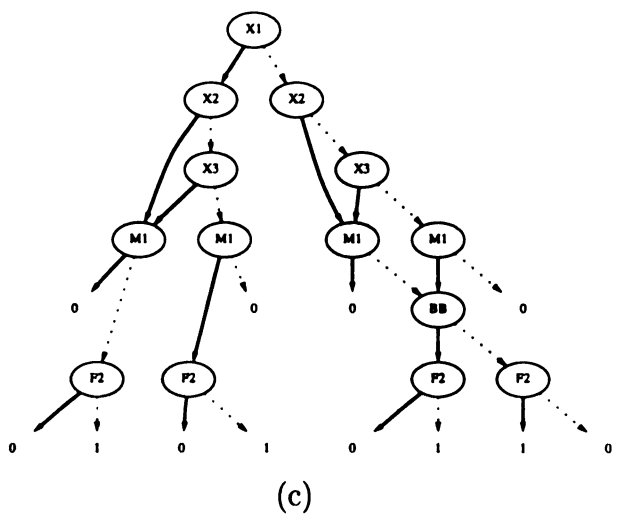


$$F_2 = \overline{X_1} \cdot \overline{BB(X_2, M_1, X_3)}$$

(a)

X_1	X_2	X_3	M_1	BB	F_2	$\mathcal{X}^S$
1	1	*d*	0	*d*	0	1
1	0	1	0	*d*	0	1
1	0	0	1	*d*	0	1
0	1	*d*	0	1	0	1
0	1	*d*	0	0	1	1
0	0	1	0	1	0	1
0	0	1	0	0	1	1
0	0	0	1	1	0	1
0	0	0	1	0	1	1
		otherwise				0

(b)



(c)

Figure 3.6: **Schematic, structural truth-table, and SBDD for partial `simplecircuit`.** (a) Partial schematic and functional description of **`simplecircuit`**; (b) Truth table for the structure function (Definition 3.3) of **`partial simplecircuit`**; (c) SBDD representing **`simplecircuit`**'s structure function. The blackbox net variable BB is used to represent the value of the unspecified structure output M_4.

Observe that all primary input variable assignments in which $X_1 = 1$ have a single 1-path, and that the value of the blackbox has no effect on the circuit output. This fact can be verified though observation of the partial circuit schematic (Figure 3.6(a)). The representation has encapsulated the fact that the value of the blackbox is a *don't care* for the value of F_2 when $X_1 = 1$. The representation also clearly shows that the

function of the blackbox structure and the value of its output BB defines the value of the output variable F_2 when $X_1 \neq 1$. In such situations, we say that F_2 *depends on* or is *sensitized to* BB. Formally:

Definition 3.7 *Let the partial specification of a combinational circuit A with primary inputs* $\mathbf{i}$, *primary outputs* $\mathbf{o}$, *and an unspecified component output represented by the blackbox net variable* b *be represented by the structure function* $\mathcal{X}^S$ *over the net variables* $\mathcal{V} = \{\mathbf{i}, \ldots, b, \ldots, \mathbf{o}\}$.

Let $\mathcal{F}_{\mathbf{i_j},\mathbf{b}}(\mathcal{V}')$ *represent a Boolean function that is true for only those assignments of net variables for which* $\mathcal{X}(\mathbf{i}_j, \ldots, b, \ldots, \mathbf{o}) = 1, \mathbf{o} \subseteq \mathcal{V}' \subseteq \mathcal{V} - \mathbf{i_j} - \{\mathbf{b}\}$.

If $\forall \mathbf{i_j} \in \{\mathbf{0},\mathbf{1}\}^{|\mathbf{i}|}, \mathcal{F}_{\mathbf{i_j},\mathbf{0}}(\mathcal{V}') = \mathcal{F}_{\mathbf{i_j},\mathbf{1}}(\mathcal{V}')$, *the value of* b *has no effect on the other net variables in* $\mathcal{V}'$, *and* b *is a* don't care under $\mathcal{V}'$.

Conversely, if $\exists \mathbf{i_j} \in \{\mathbf{0},\mathbf{1}\}^{|\mathbf{i}|}, \mathcal{F}_{\mathbf{i_j},\mathbf{0}}(\mathcal{V}') \neq \mathcal{F}_{\mathbf{i_j},\mathbf{1}}(\mathcal{V}')$, *the value of* b *has an effect on the other net variables in* $\mathcal{V}'$, *and the net variables whose value depends on* b *are* sensitized to b.

The BDD representation of the structural function allows one to efficiently identify input conditions under which output variables are sensitized to a blackbox net variable. In Figure 3.6, one can see that the value of the primary output net variable F_2 is independent of the value of the blackbox net variable BB when $X_1 = 1$. However, when $X_1 = 0$, the value of F_2 clearly depends upon the value of BB. For the input vector $X_1 = 1, X_2 = 0, X_3 = 1$, for example, if $BB = 1$, then $F_2 = 0$. If, however, $BB = 0$ under the same input vector, then $F_2 = 1$. Therefore, F_2 is sensitized to the blackbox net variable BB. This situation leads to the following result:

Theorem 3.2 *Let G be an SBDD representation (Definition 3.4) of a structural function of a combinational circuit containing a blackbox net variable b. Consider each 1-path in G.*

1. *If the node corresponding to the net variable b does not appear on the 1-path, then b is a don't care under the variable assignment described by the 1-path.*

2. *If the node corresponding to the net variable b does appear on the 1-path, then at least one net variable representing a structure in the transitive fan-out of the structure represented by b is sensitized to b's value. Furthermore, the sensitized variable will appear between b and the 1-terminal.*

Proof:

(1) Assume that for a blackbox net variable b that does not appear on some 1-path in G, that b is not a *don't care* under the variable assignment described by the 1-path. If b is not a *don't care*, then according to Definition 3.7, the value of at least one variable represented in G depends on (is sensitized to) the value of b.

Furthermore, the variables form a Boolean network (Definition 2.16) representing a combinational circuit. Thus the sensitized variable must appear in b's transitive fan-out. Since G is an SBDD, its variables are ordered based upon the partial order implicit in the DAG representation of the Boolean network (Definition 3.4). Therefore the sensitized variable has a higher index (Definition 2.21) than b in G's BDD variable ordering

Since b does not appear on the 1-path, it must be reduced, and thus, $child_r(b) = child_r(b)$ by Definition 2.22. It has already been determined, however, that a variable whose index is greater than b's exists and that since the variable is sensitized to b, its value depends upon b under the input vector of the 1-path under observation. Therefore the value of that variable depends on whether it is the left ($b = 0$) or right ($b = 1$) child of b. It has already been shown, however, that the right and left children of b are equivalent. This is a contradiction, and therefore our initial assumption must be false. Any blackbox net variable b that does not appear on some 1-path in G is a *don't care.*

(2) Since the variables form a Boolean network (Definition 2.16) representing a combinational circuit, a sensitized variable (if one exists) must appear in b's transitive fan-out and therefore has a higher index (Definition 2.20) than b in the BDD variable ordering (Definition 3.4). Thus, it can only appear between b and the 1-terminal in any path, including the one under observation.

Assume that for a blackbox net variable b that appears on some 1-path in G, that no net variable representing a structure in the transitive fan-out of the structure represented by b is sensitized to b's value. Furthermore, since the variables form a Boolean network (Definition 2.16) representing a combinational circuit, only variables that represent a structure which appears in the transitive fan-out of the structure represented by b can

be sensitized to b. Thus, no variables on the 1-path are sensitized to b. Therefore b is a *don't care* along that 1-path and b's right and left children are identical. It is a property of ROBDDs that nodes whose children are isomorphic subgraphs are removed from the diagram. This contradicts the fact that a decision node representing the variable b appears on the 1-path, and therefore our initial assumption must be false. If b appears on some 1-path in G, a variable which is sensitized to b appears along that 1-path.

Theorem 3.2 allows one to determine when a blackbox variable is fully specified by simply examining of the structure of the SBDD graph. For example, note that BB is on at least one 1-path (e.g., any 1-path for which $X_1 = 1$) in Figure 3.6. Therefore some net variable (in this case the net variable F_2) must be sensitized to BB. Thus multiple 1-paths exist under the primary input vector for which the net variable F_2 is sensitized to BB. Therefore, according to Definition 3.6, BB is not fully specified.

3.4 SBDD efficiency

Although SBDDs encode the mathematical relationships between input, output, and internal structure values implicit in the hierarchy of the circuit, the actual topology of the circuit is not represented. To be able to introduce information from any level of design abstraction, one cannot assume that the specification has the same topology as the implementation or even that corresponding blocks in the specification and implementation are equivalent. An assumption of topological equivalence may

be invalidated by optimization techniques applied to the implementation; thus, it must be avoided. SBDDs encode structural context only via representation of the mathematical relationship between structures.

The representation of these numerous relationships is not without cost. The structure function (Definition 3.3) which encapsulates these relationships may become exceptionally complex. The next section introduces issues related to the efficiency of the BDD representation of the structure function.

3.4.1 Operations on and properties of the SBDD

Since a BDD's size is sensitive to the order of its variables, it is necessary to order the SBDD decision variables appropriately. These variables include the input variables and the additional variables for internal structures in the design, including the design outputs. As presented in Definition 3.4, a structural-level circuit implementation can be viewed as a directed, acyclic graph (DAG) in which each vertex represents a structure (usually a gate) and each arc (structure_1, structure_2) indicates that the output of structure_1 is an input to structure_2. This DAG forms a partial order on the set of gates. We use this order as a basis for selecting an SBDD variable total ordering heuristic.

This ordering exploits the structure of the circuit to obtain a suitable input ordering (although it may not be optimal). Efficiency issues are discussed in Chapter 6. Similar heuristic orderings are used by Bryant (Bryant, 1986). Dynamic reordering (Rudell, 1993) and other advanced BDD reduction techniques can be applied to the

SBDD structure if necessary.

An SBDD can be produced for any circuit by creating an SBDD for each structure (as shown for a NOR gate in Figure 3.5) and then successively AND-ing these SBDDs together by using the BDD recursive **Apply** operation (Figure 2.4). Each structure acts as a constraint on the behavior of the circuit and marks illegal argument assignments as 0-paths.

For example, consider **simplecircuit** (Figure 3.1). An SBDD for each gate in **simplecircuit** can be constructed, illustrated for the gate M_1 discussed previously in Figure 3.4. When these SBDDs are combined with the BDD AND operation, the result is the complete SBDD for the circuit. Figure 3.1(c) presents the complete SBDD for **simplecircuit** and demonstrates that the SBDD encodes the value of every gate-level structure in the circuit under every possible assignment of primary inputs.

In a completely specified combinational circuit, the value of each internal structure is defined by the values of the primary inputs. Therefore, exactly one arc leading from each SBDD node representing an internal structure is illegal and should lead to the 0-terminal. The example SBDD shown in Figure 3.1(c) illustrates this property.

3.4.2 Increasing SBDD efficiency though reduction

A key aspect of the efficiency of the BDD representation is the sharing of isomorphic subgraphs in reduced BDDs, as described in Definition 2.22. In a complete, fully specified SBDD, however, there may be very little sharing of isomorphic subgraphs

and very little accompanying reduction in the number of irrelevant decision nodes. Such an SBDD has a worst-case size of $O(2^n k)$, where n is the number of input variables plus blackbox nodes and k is the number of non-blackbox structures represented in the SBDD. Since each behavior of the non-blackbox structure is fully defined by the value of the input variables and blackbox structures, exactly one arc from any decision node representing such a variable leads directly to the 0-terminal, and exactly one arc does not. There is no real "branching" at such a node, and thus such a variable increases the maximum size of the SBDD linearly rather than exponentially.

Worst-case size may by observed if 1-paths exist that are only slightly different from each other (perhaps differing in only a single gate). Such a difference forces the SBDD representation to create two separate 1-paths until they merge after the portion of each path in which they differ. Reducing the number of internal structures represented by an SBDD not only reduces the graph size linearly, but also increases the likelihood of isomorphic subgraph sharing.

Reducing such a decision variable is straightforward. Because exactly one arc leads to the 0-terminal, each decision node that refers to the variable can be removed by pointing the incoming arc directly to the node referred to by the outgoing arc that does not point to the 0-terminal. For example, if a decision node variable representing $M1$ in the SBDD for `simplecircuit` (Figure 3.7(a)) is removed, an SBDD (Figure 3.7(b)) that still represents the effect of that gate, but that no longer has any node containing that decision variable, is obtained.

The functional constraints imposed by such a structure (whose decision variable has been reduced) are still represented in the SBDD. However, the ability to directly

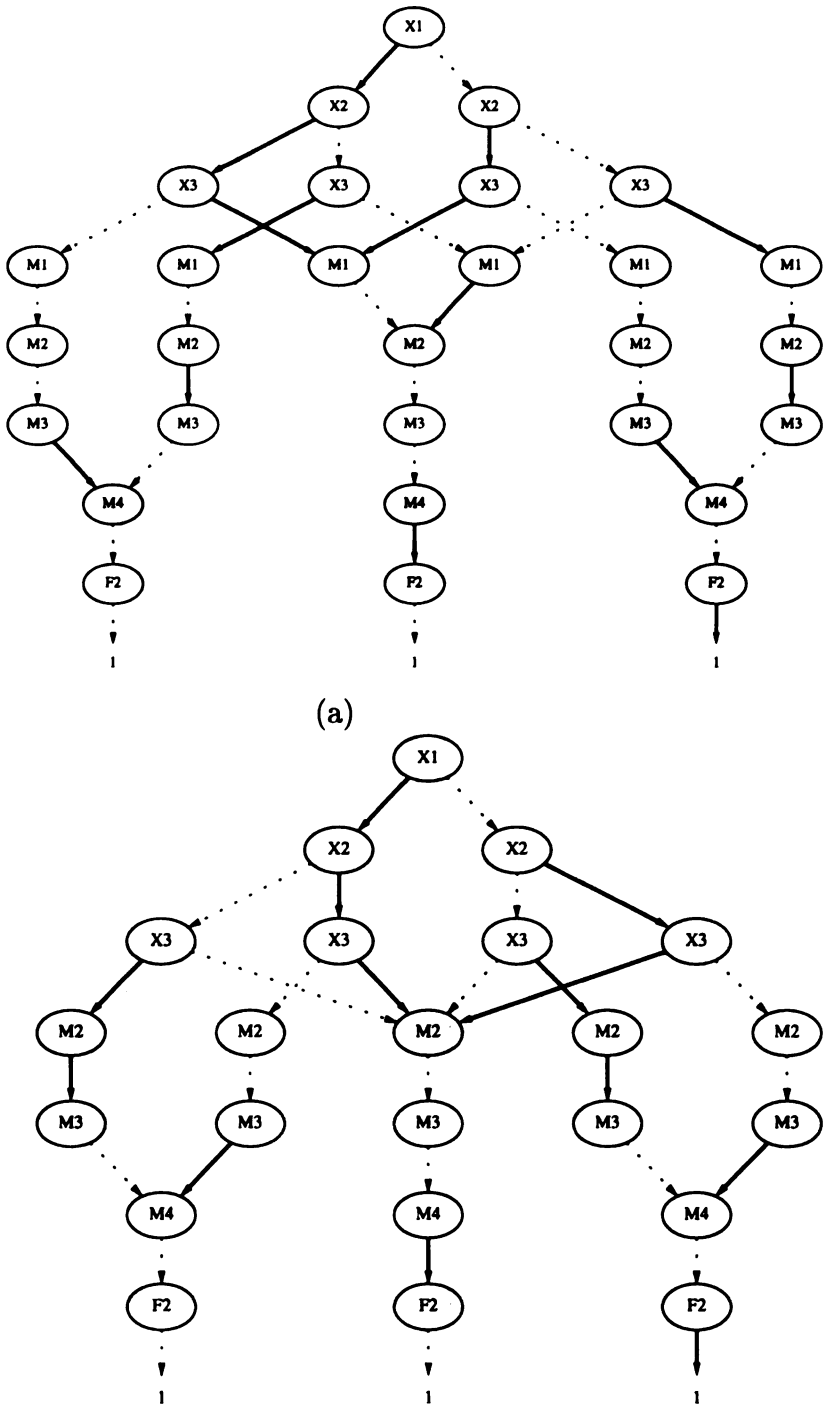


Figure 3.7: **Example reduced SBDD for `simplecircuit` (Figure 3.1).** (a) The unreduced SBDD for **`simplecircuit`**. (b) The SBDD for **`simplecircuit`** with the variable $M1$ reduced. For clarity, arcs leading directly to the 0-terminal are not shown in the graphical representation.

reference the reduced variable in any way is lost. Deduced relationships involving the variable that have already been introduced are not lost, but the ability to introduce new relationships that directly involve such a reduced variable is lost. If some set of decision node variables that does not need to be referenced directly in future constraints can be determined, reducing these variables can greatly reduce the size of the SBDD representation.

To work on very large circuits efficiently, the number of intermediate gate variables should be limited to only those relevant to a particular problem. If, however, the goal is to represent the unreduced structure of an entire circuit, then the SBDD is no less efficient than any other encoding.

If the goal is restricted to representing only the knowledge pertaining to the overall function output and determining the functionality of a set of blackbox components, one can reduce the SBDD's size by limiting the decision variables as follows:

- Decision variables representing primary inputs may not be removed, since they are necessary to specify the state of the overall circuit.
- Decision variables representing circuit output may not be removed, since their values capture the functionality of the circuit.
- Decision variables representing a structure that is a direct input to a blackbox may not be removed, since they specify the state of the blackbox under various input conditions.
- Decision variables representing blackbox outputs may not be removed, since they encode the relationships which we are attempting to recover.

- Any remaining decision variable may be removed once all the variables representing the outputs of any structures to which it is a direct input have been introduced into the SBDD, since the decision variable will not be directly referenced once all such relationships have been introduced.

Consider the circuit shown in Figure 3.8 and the corresponding complete, fully specified SBDD shown in Figure 3.9. This complete SBDD representation has few subgraph isomorphisms and therefore approaches worst-case size. However, since the representation of the SBDD is limited to only those variables that are needed to solve a particular problem (such as determining the function of the blackbox $BB1$ in the **schneider1** circuit presented in Figure 3.10), the size of the SBDD can be greatly reduced by removing unnecessary net variables. Figure 3.11 presents the reduced SBDD for the **schneider1** circuit.

The reduced representation clearly shows that the output of the blackbox $BB1$ has no effect on the output Z and can be considered a *don't care* under every assignment of the input variables except for the assignments $(1,1,1,1)$, $(1,0,0,d)$, and $(0,0,0,1)$. This representation does not, however, show the relationships between the removed net variables and the value of the blackbox variable. Thus, the ability to further specify the behavior of the blackbox variable with any information that defines relationships between the blackbox variable and the variables that have been removed is lost. Chapter 4 discusses how to take advantage of relationships between the net variables represented in the SBDD to help complete the specification of a partially specified blackbox function.

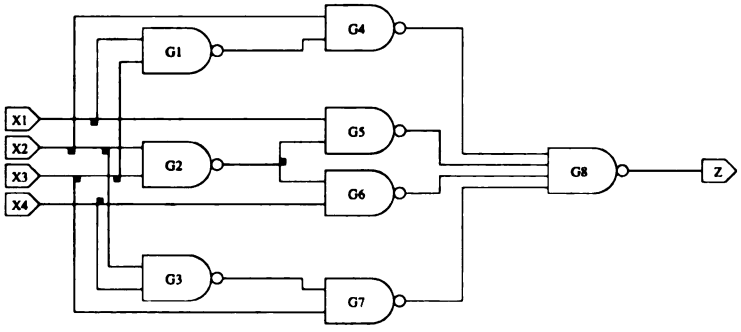


Figure 3.8: **Schneider circuit.**

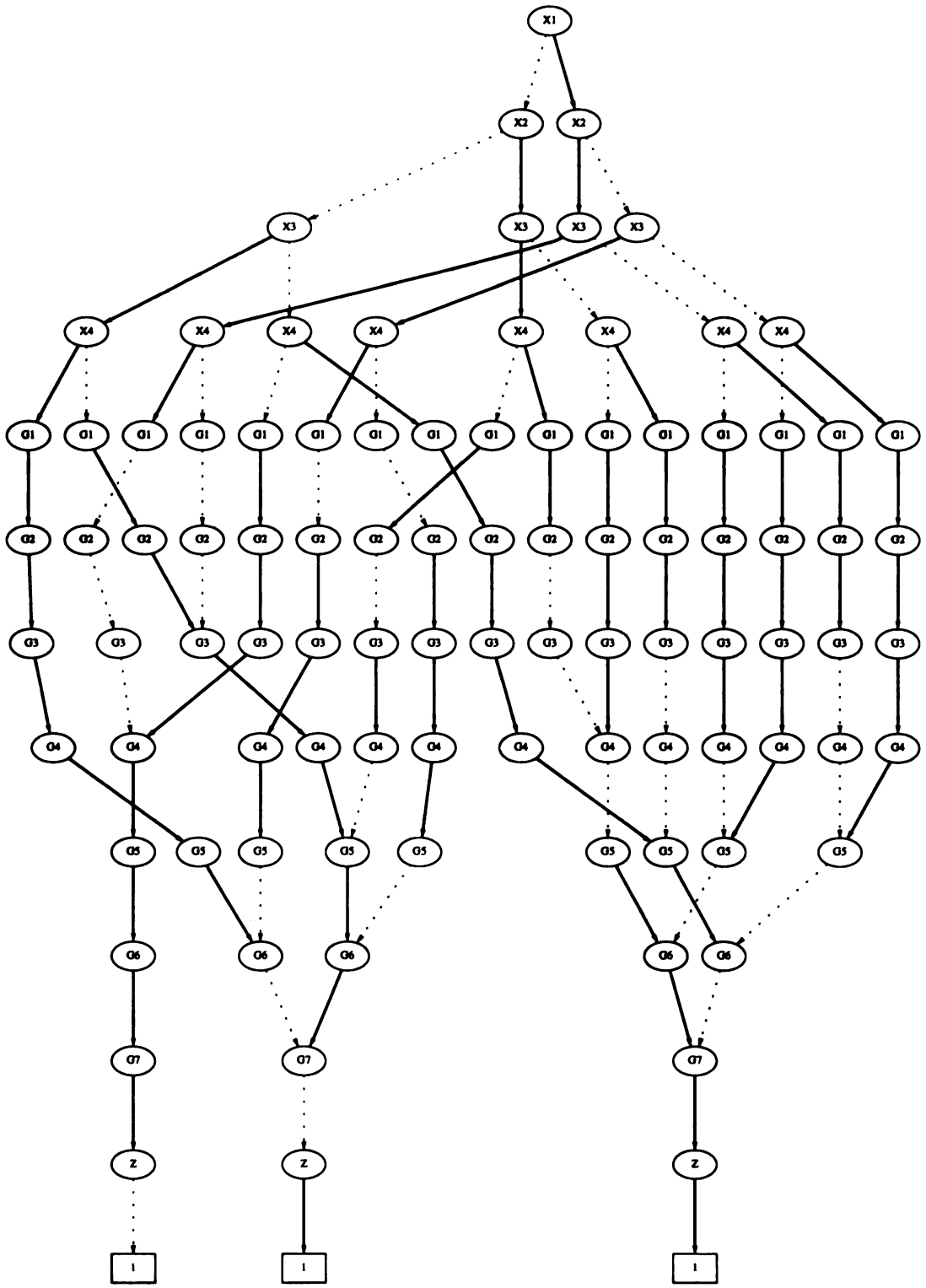


Figure 3.9: **Unreduced SBDD for the schneider circuit.** For clarity, arc leading directly to the 0-terminal are not shown in the graphical representation.

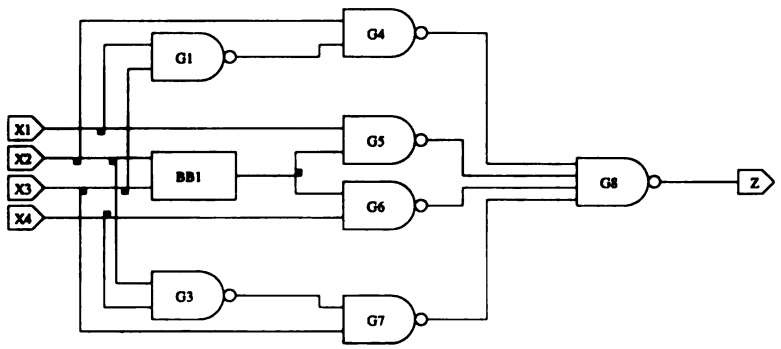


Figure 3.10: **Schneider1 circuit.** This figure presents a partial schematic for the **schneider** circuit (Figure 3.8). In this example, the function of the portion of the circuit represented by the blackbox device *bb*1 is unknown.

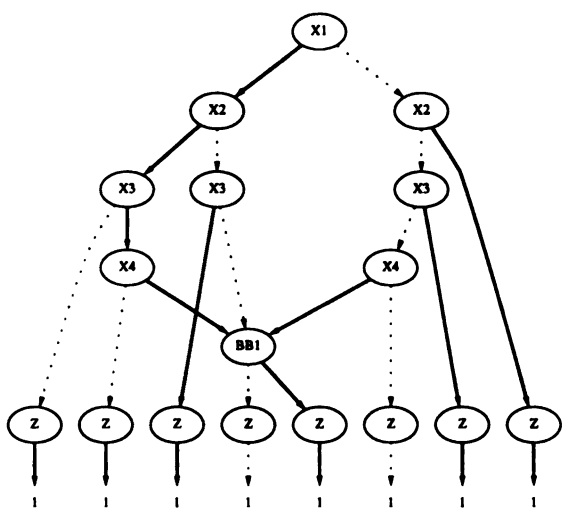


Figure 3.11: **Reduced SBDD for the schneider1 circuit.** For clarity, arcs leading directly to the 0-terminal are not shown in the graphical representation.

Chapter 4

Representation and recovery of partial knowledge

As described in Chapter 1, the goal of digital design recovery is to reconstruct the behavioral-level description of an existing electronic part, board, or system from available information. This task is complicated by the fact that often only partial or conflicting design information is available.

Our overall approach, therefore, relies primarily on information provided by the analysis of an existing system (Section 1.2.3). This process, however, is not necessarily 100% accurate. Errors in etching or imaging, or an inability to identify the functionality of particularly complex transistor layouts may cause the recovered design implementation to contain areas with unknown functionality (e.g., blackboxes). Even when an adequate description of the implementation is available, it may be important to be able to verify the accuracy of existing documentation by testing this description against all available information.

A key aspect of the design recovery problem is dealing with systems for which the original design information is incomplete or conflicting. Existing reengineering approaches do not address this issue. We believe that a complete redesign process requires a methodology to infer the missing or incomplete specification, if possible.

This chapter presents a methodology for uniformly representing any available system information that describes functional relationships among system components. It then describes a new way to use this representation to discover complete circuit functionality in many cases. Although the ability to specify an unknown portion of a circuit implementation depends on the coverage provided by the available system, the approach presented in this dissertation will generally allow the efficient recovery of blackbox functionality if this functionality is deducible from the available information.

4.1 Representing partial knowledge

Don't care information is used for optimization in many digital system design applications. This information, usually in the form of incompletely specified Boolean functions, is traditionally represented as a *ternary-valued function* $f : \{0,1\}^n \rightarrow \{0,1,d\}$ (Minato, 1996). Ternary-valued functions are manipulated by the extended logic operations presented in Figure 4.1.

There are two common ways in which decision diagrams are used to represent ternary-valued functions. The first method is to represent the ternary values 0, d, and 1 as terminal nodes in a multiple-terminal BDD (Matsunaga and Fugita, 1989).

NOT

f	$\overline{f}$
0	1
d	d
1	0

AND

$f \cdot g$	0	d	1
0	0	0	0
d	0	d	d
1	0	d	1

OR

$f + g$	0	d	1
0	0	d	1
d	d	d	1
1	1	1	1

EXOR

$f \oplus g$	0	d	1
0	0	d	1
d	d	d	d
1	1	d	0

Figure 4.1: **Extended logic operations for ternary functions.**

The second way is to represent the function as a pair of BDDs in which each ternary value is represented by an encoded pair of Boolean values (Minato et al., 1990).

These representations of incompletely specified Boolean functions are highly efficient for calculating and representing the *don't care* sets and are often used for optimization as part of the technology mapping process. In these traditional representations, however, the decision variables cannot assume the value d; only the function value can take on the d value. Furthermore, neither the decision variables nor the overall function can be assigned an unknown value that is not a *don't care*. Thus, these BDDs do not represent the entire augmented Boolean domain (Definition 2.12).

An obvious approach to representing functions $f : \tilde{\mathcal{B}}^n \rightarrow \tilde{\mathcal{B}}$ is to define a set of extended logic operations for the augmented Boolean domain and to extend the multiway decision diagram such that each decision node contains a 0-edge, 1-edge, d-edge, and a u-edge. A representation of this sort accurately (if not necessarily efficiently) represents functions in the augmented Boolean domain. Although this representation allows the specification of the effect of a variable having an unknown

value, it does not provide a mechanism for determining that a variable must logically take on the value *don't care* or *unknown* along any path. Hence, this representation is not particularly amenable to automated deduction.

The SBDD representation of partial functions presented in Section 3.3 encapsulates the representation of both the *don't care* and the *unknown* (undefined) relationships that are present in a partially specified Boolean network (Section 2.1.3). Furthermore, simple examination of the graph structure allows the identification of *don't cares* and unknowns which exist in the system. Finally, even though this information about additional variable and functional values is included in the SBDD representation, no new algorithms for creating SBDDs or operating on them need to be developed. The available (and developing) BDD algorithms can be used directly. Therefore, we will use SBDDs to represent our system of functions in the augmented Boolean domain. Additionally, we will present SBDD algorithms that allow the introduction of new information that may specify, in part or full, the unknown relationships represented.

As defined in Chapter 3, an SBDD decision node variable may represent any structure in a circuit description regardless of whether its function is completely specified. It is this ability to efficiently represent partial knowledge that allows SBDDs to be an appropriate tool for reengineering systems in which only partial knowledge is available.

Consider any gate, or collection of gates, that defines the functional relationship between any arbitrary set of internal or primary input lines and an arbitrary set of internal or primary output lines. Based on Section 3.3, such a gate or collection of

gates for which a complete functional specification is unavailable is referred to as a blackbox structure. Since SBDDs represent one output line per decision variable, a blackbox decision node variable must be created for each output of such an unknown structure.

Because no information on which to base the functionality of the blackbox is available, there is no information that allows the overall functionality of the circuit to be constrained: a 1-path must exist for either value of the blackbox decision variable (Definition 3.5). Exactly one of these paths is legal in the fully specified circuit function, but allowing 1-paths along both branches enables the value of this component to be represented as unknown. If the blackbox has no effect on decision variables subsequent to it in the variable ordering, its value is effectively a *don't care* and does not appear on that SBDD 1-path. Any SBDD decision variable representing a structure that is fully specified will have one branch leading immediately to a 0-terminal, and the other branch will be part of at least one 1-path. An SBDD node representing a blackbox variable may have 1-paths along both branches; such nodes are called *unknown nodes.*

Figure 4.2(a) shows a completely unspecified circuit that computes the two-output Boolean function $\mathcal{F}(a, b) = (x, y)$. Without additional knowledge, the SBDD for such a circuit would simply be the 1-terminal, because *no* information regarding the circuit's functionality would be available. Although this example provides no structural context that might allow one to deduce additional information, assume that a partial description of the circuit behavior is available that specifies that $\mathcal{F}(1, 1) = (1, 1)$ and $\mathcal{F}(1, 0) = (0, u)$. Figure 4.2(b) shows the SBDD for this circuit that encodes this

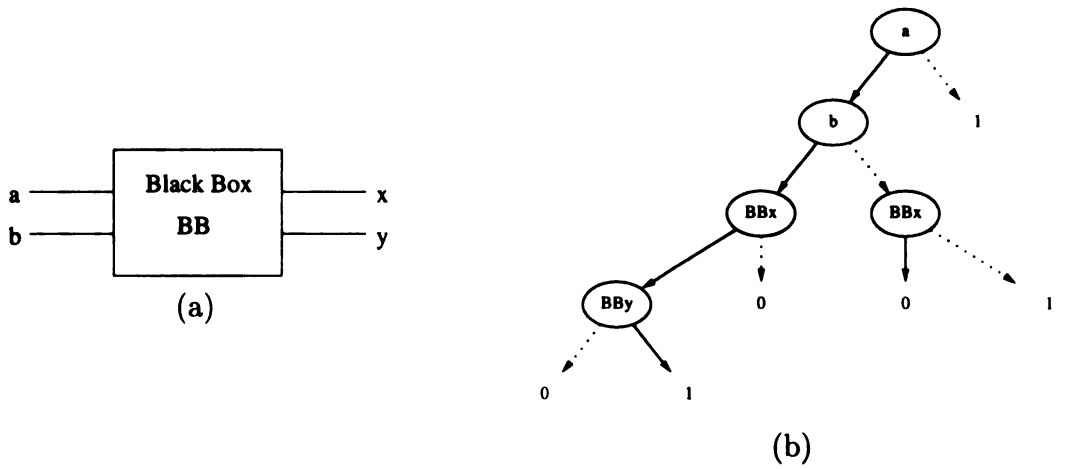


Figure 4.2: **Unknown circuit implementation.** The SBDD representation of partial knowledge. This SBDD represents the partial knowledge $\mathcal{F}(1,1) = (1,1)$ and $\mathcal{F}(1,0) = (0,u)$.

partial knowledge. Note that this figure contains a blackbox decision node variable for each of the two outputs of BB and fully encodes the partial knowledge introduced. The SBDD can be interpreted as follows:

- *If $a = 0$, then both blackbox outputs are unknown.* Consider the path $a \xrightarrow{0} 1$ in the SBDD for this circuit (Figure 4.2(b)). Recall that any path that ends at the 1-terminal in an SBDD *may* be legal in the system. Therefore, any assignment of values to BB_x and BB_y may be legal when $a = 0$, and thus their behavior is not determinable from the information provided.

- *If $a = 1$ and $b = 1$, then $BB_x = 1$ and $BB_y = 1$.* Consider the path $a \xrightarrow{1} b \xrightarrow{1} BB_x \xrightarrow{0} 0$ in the SBDD for this circuit (Figure 4.2(b)). Recall that any path that ends at the 0-terminal in an SBDD is illegal in the system (i.e., it contradicts a known relationship). Therefore, $BB_x \neq 0$ when $a = 1$ and $b = 1$.

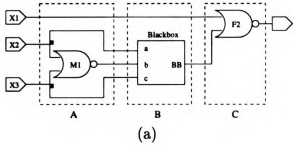
Hence, $BB_x = 1$. Similarly, the path $a \xrightarrow{1} b \xrightarrow{1} BB_x \xrightarrow{1} BB_y \xrightarrow{0} 0$ indicates that $BB_y \neq 0$ when $a = 1$, $b = 1$, and $BB_x = 1$. Hence, $BB_y = 1$.

- *If* $a = 1$ *and* $b = 0$, *then* $BB_x = 0$ *and* BB_y *is unknown.* Consider the path $a \xrightarrow{1} b \xrightarrow{0} BB_x \xrightarrow{1} 0$ in the SBDD for this circuit (Figure 4.2(b)). Therefore, $BB_x \neq 0$ when $a = 1$ and $b = 1$ Hence, $BB_x = 1$. Furthermore, the path $a \xrightarrow{1} b \xrightarrow{0} BB_x \xrightarrow{0} 1$ indicates that no assignment of remaining variables is illegal when $a = 1$, $b = 0$, and $BB_x = 0$. Thus, the actual value of BB_y cannot be determined under the input conditions $a = 1$, $b = 0$ and BB_y is considered unknown.

It should be stressed that this representation is based entirely upon effective functionality of the blackbox or blackboxes in the context of the overall system functionality. The number or position of the blackboxes within the system is not an issue with regards to this representation although it may play a factor in the difficulty of recovering the missing functionality.

Example: partial specification of `simplecircuit`

Figure 4.3(a) presents a partial specification for **`simplecircuit`** (Figure 3.1(a)), representing a portion of the implementation as a blackbox. Consider the symbolic solution space to this partial specification (Figure 4.3(b). This table clearly represents that the value of the output F_2 is 0 for all input variable assignments with $X_1 = 1$. Therefore the behavior of the unspecified portion of the circuit is irrelevant and thus a *don't care.*



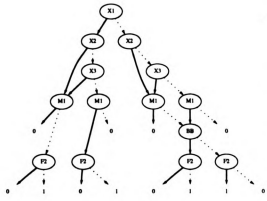
(a)

$$F_2 = \overline{X_1} \cdot \overline{BB(X_2, M_1, X_3)}$$

X_1	X_2	X_3	$M1$	BB	F_2
1	d	d	d	d	0
0	1	1	0	u	$\overline{BB(1,0,1)}$
0	1	0	0	u	$\overline{BB(1,0,0)}$
0	0	1	0	u	$\overline{BB(0,0,1)}$
0	0	0	1	u	$\overline{BB(0,1,0)}$

(b)

Figure 4.3: **A partial specification of `simplecircuit`.** (a) Circuit with unspecified subcircuit: Area A represents the known input logic, area B represents the unknown blackbox subcircuit, and area C represents the known output logic. (b) The symbolic solution space to this circuit: The blackbox representing M_4 is denoted symbolically as BB.



(a)

X_1	X_2	X_3	$M1$	BB	F_2
1	1	d	0	d	0
1	0	1	0	d	0
1	0	0	1	d	0
0	1	d	0	**if** $BB = 1$	**then** $F_2 = 0$
				if $BB = 0$	**then** $F_2 = 1$
0	0	1	0	**if** $BB = 1$	**then** $F_2 = 0$
				if $BB = 0$	**then** $F_2 = 1$
0	0	0	1	**if** $BB = 1$	**then** $F_2 = 0$
				if $BB = 0$	**then** $F_2 = 1$

(b)

Figure 4.4: **The SBDD for a partial specification of `simplecircuit`.** The truth-table representation of the SBDD contains unknowns but still encapsulates what the state of the circuit is under either of the possible values of BB under all variable assignments.

Figure 4.4(a) presents the SBDD for this partial specification. All input variable assignments for which $X_1 = 1$ have a single 1-path, and the value of the blackbox has no effect upon the circuit output. The representation has encapsulated the fact that the value of the blackbox is a *don't care* for the overall function when $X_1 = 1$.

The symbolic solution space to any such problem is contained within the SBDD representation and can be extracted without difficulty. Figure 4.4 illustrates the extraction of the truth-table representation of the blackbox BB from its SBDD representation. This truth-table representation is equivalent to the symbolic solution space presented in Figure 4.3(b).

Consider the truth-table row in Figure 4.3(b) corresponding to the assignment $X_1 = 0$, $X_2 = 1$, $X_3 = 0$. The symbolic truth-table specifies that $M_1 = 0$, that the value of the blackbox BB is unknown, and that the value of the output F_2 is $\overline{BB}$ when the inputs to BB are $X_2 = 1$, $X_3 = 0$, and $M_1 = 0$. This information is represented in the circuit's SBDD (4.4(a)) as follows.

Consider the path $X_1 \xrightarrow{0} X_2 \xrightarrow{1} M_1 \xrightarrow{0} BB$. Since neither of BB's children are the 0-terminal, the value of BB is unknown (Definition 3.6). The paths $X_1 \xrightarrow{0} X_2 \xrightarrow{1} M_1 \xrightarrow{0} BB \xrightarrow{1} F_2 \xrightarrow{1} 0$ and $X_1 \xrightarrow{0} X_2 \xrightarrow{1} M_1 \xrightarrow{0} BB \xrightarrow{1} F_2 \xrightarrow{0} 1$ allow one to deduce that when $X_1 = 0$ and $X_2 = 1$ (regardless of the value of X_3) $M_1 = 0$, and that if $BB = 1$ that the only legal assignment for F_2 is 0. Likewise, the paths $X_1 \xrightarrow{0} X_2 \xrightarrow{1} M_1 \xrightarrow{0} BB \xrightarrow{0} F_2 \xrightarrow{1} 1$ and $X_1 \xrightarrow{0} X_2 \xrightarrow{1} M_1 \xrightarrow{0} BB \xrightarrow{0} F_2 \xrightarrow{0} 0$ allow one to deduce that when $X_1 = 0$ and $X_2 = 1$ that (regardless of the value of X_3) $M_1 = 0$, and that if $BB = 0$ that the only legal assignment for F_2 is 1. This information is contained in the truth-table row in Figure 4.4(b) corresponding to the assignment

$X_1 = 0$, $X_2 = 1$, $X_3 = d$, $M_1 = 0$. Note that the output $F_2 = \overline{BB}$ in this truth-table corresponds to the known information available in Figure 4.3(b).

Therefore, using only the SBDD representing the partial specification, one can deduce that $M_1 = 0$, the value of the blackbox BB is unknown, and that the value of the output F_2 is $\overline{BB}$ for the the input assignment $X_1 = 0$, $X_2 = 1$. Similar deductions allows the construction of a truth-table representing the information encapsulated by the SBDD (Figure 4.4(b)).

4.2 Utilizing additional relationships

As discussed previously, various forms of information concerning the functionality and structure may be available for any digital system. To make full use of all this information, a uniform way to represent the Boolean relationships between the net variables that the information describes must be determined. This section describes a methodology for representing full or partial relationships between net variables and shows how this information can be used to more completely specify the behavior of a partially specified digital device.

Definition 4.1 *For any Boolean relationship* $\mathcal{R}(\mathbf{x}, \mathbf{y})$ *between net variable vectors* $\mathbf{x}, \mathbf{y} \in \mathcal{N}$, *a* constraint characteristic function *is defined as:*

$$\mathcal{X}^R(\mathbf{x}, \mathbf{y}) = 0 \text{ iff } \mathbf{x}\overline{\mathcal{R}}\mathbf{y}. \tag{4.1}$$

Consider, for example, a device that includes four net variables: a, b, c, and d. Suppose that available information states that a and b are control variables and that if a and b are both 1, then $d = \bar{c}$. This relationship would be represented by the constraint characteristic function shown in Figure 4.5.

a	b	c	d	$\mathcal{X}^R$
0	d	d	d	1
d	0	d	d	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Figure 4.5: **Example constraint characteristic function.** This truth-table represents the constraint characteristic function $\mathcal{X}^R(< a, b >, < c, d >)$ for a relationship that states that if $a = 1$ and $b = 1$, then $d = \bar{c}$.

Applying a constraint characteristic function (Definition 4.1) to the structure function (Definition 3.3) for a circuit introduces additional knowledge while assuring the validity of Theorems 3.1 and 3.2. The introduction of relationships created from available device information may allow the behavior of a partially specified circuit to be fully specified, as stated below.

Theorem 4.1 *Let $\mathcal{X}^S(\mathcal{N})$ represent the structure function (Definition 3.3) for a partial specification of a combinational circuit A in terms of net variables $\mathcal{N}$. Furthermore, let $\mathcal{X}^R(\mathbf{x}, \mathbf{y})$ describe a relationship between two sets of net variables in $\mathcal{N}$. Let A contain a partially specified component B, whose inputs $\mathbf{B_{in}}$ depend on some set of*

variables in $\mathbf{x}$ *and a variable* u*, upon whose value some set of variables in* $\mathbf{y}$ *depends. If* u *is sensitized to an output* B_{out} *of* B*, then* $\mathcal{X}^R$ *implies a partial specification of* B*.*

Proof: As defined above, the value of the net variables $\mathbf{B_{in}}$ and $\mathbf{y}$ depend on the assignment of net variables in $\mathbf{x}$. Likewise, since the values of the net variables in $\mathbf{y}$ are dependent upon u, the value of u is determined by the necessary value required to produce the appropriate value in $\mathbf{y}$ for the input assignment $\mathbf{x}$. Therefore the value of u is also dependent on $\mathbf{x}$.

Furthermore, since the variable u is sensitized to the blackbox output B_{out}, the value of u determines the necessary value of B_{out} in the sensitive assignment(s). Thus, the value of B_{out} can be determined for the blackbox inputs $\mathbf{B_{in}}$ under any conditions specified in $\mathcal{X}^R$ for which u's value can be determined for an assignment of $\mathbf{x}$ in which the u is also sensitized to B_{out}. Therefore an output function for B has been partially specified.

In essence, this theorem states that when the characteristic function for any relationship is applied to the structural function for the circuit, previously unspecified behavior of blackbox components may become specified. As mentioned in Section 1.2.1, a great deal of information regarding a design is often available. Regardless of the level of design at which information is presented (Section 1.1.2), if the available information describes a Boolean relationship between net variables, its corresponding constraint characteristic function can be determined and used to try to specify the behavior of the blackbox. Thus, we have described a way in which information from any level of design can be used to help determine the behavior of an unspecified black-

box structure. In the remainder of this section we illustrate the use of this theorem by demonstrating how the relationships present in test vectors can help specify the functionality of a partial schematic.

4.2.1 Utilizing test vectors

A number of possible sources for design information exist, but the actual format of the information is not particularly relevant as long as the partial relationships between the net variables that the information describes can be examined. The text that follows considers a set of test vectors as an additional source of information and demonstrates how this information can be represented as a set of constraint characteristic functions (Definition 4.1). To more clearly demonstrate the application of the proposed methodology, we focus on instances of the following problem:

> **Reengineering from partial specifications (RFPS) problem:** Given a partial representation for a combinational circuit and a set (partial or complete) of test vectors for the functioning circuit, discover the global functionality of the circuit and the effective functionality of the partially specified circuit components.

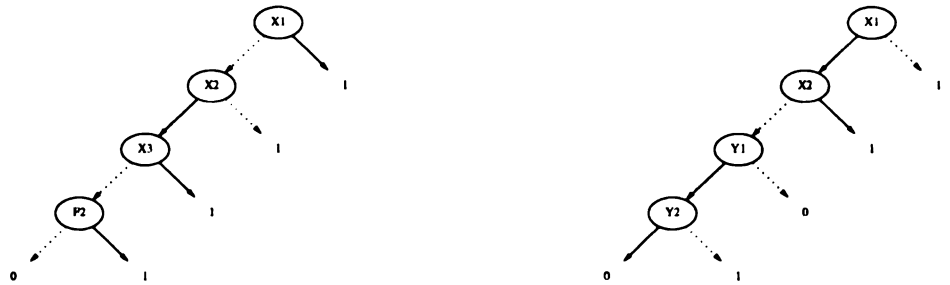
For the purpose of this problem, circuit functionality refers to the underlying basic logic functions created by connecting gates, library constructs, or other digital building blocks. This functionality does not include hold times, wire and propagation delays, or other timing issues. The goal is simply to identify the basic function of the complete combinational circuit. Once functionality has been determined, traditional

techniques can be used to re-synthesize missing portion of the circuit.

Sets of test vectors that are used for simulation testing and post-silicon verification are commonly available for digital designs. For combinational circuits, *test vectors* consist of a set of input vectors and their corresponding output vectors that can be used to verify correctness of function. Therefore, test vectors provide partial information about the input/output relationship of the circuit, which can be described by constraint characteristic functions and therefore represented by an SBDD.

The relationships specified by a test vector may be represented as an SBDD in which all paths are 1-paths except for the paths that correspond to the input variable state specified by the test vector and the *illegal* output states. Consider the test vector $< 0,1,0 >=< 1 >$ for **simplecircuit** (Figure 3.1) that has inputs X_1, X_2, and X_3 and a single output F_2. This test vector describes a partial relationship that states that F_2 cannot have the value 0 if $X_1 = 0, X_2 = 1$, and $X_3 = 0$. This relationship can be represented by a constraint characteristic function and the corresponding SBDD shown in Figure 4.6(a). Likewise, a test vector $< 1,0 >=< 1,0 >$ for a circuit with inputs X_1, X_2 and outputs Y_1, Y_2 describes a relationship that states that the only legal value for Y_1 is 1 and the only legal value for Y_2 is 0 when $X_1 = 1$ and $X_2 = 0$ (Figure 4.6(b)).

Figure 4.6(a) shows the SBDD constraint for one of the automatic test pattern generation (ATPG) vectors for **simplecircuit** generated by using the U.C. Berkeley SIS package (Sentovich et al., 1992). Such a constraint can be used to invalidate all 1-paths in the SBDD that represent an assignment of values to net variables that was proven illegal by the test vector. This SBDD constraint can be composed with the



(a) (b)

Figure 4.6: **SBDDs for test vectors.** These SBDDs represent the test vectors for: (a) $f : B^3 \rightarrow B^1; f(0,1,0) = 1$ and (b) $g : B^2 \rightarrow B^2; g(1,0) = (1,0)$.

SBDD for a partial specification of the circuit (Figure 4.4) to produce a new SBDD encapsulating this additional knowledge (Figure 4.7). This constraint invalidates all paths of Figure 4.4 along which $X_1 = 0, X_2 = 1, X_3 = 0$, and $F_2 \neq 1$. All other paths in the SBDD are unaffected by this constraint. See Appendix A for a detailed description of this process.

Introducing new knowledge via constraint may have one of several effects on the SBDD:

- It may "discover behavior" and reduce "unknown" nodes (that is, blackbox nodes in which neither arc leads to a terminal-0) into "known nodes", thereby achieving the same effect as traditional backtraceing.

  This effect can be seen in the previous example. BB is an unknown node along the path $X_1 = 0, X_2 = 1, X_3 = 0$ in Figure 4.4. After the application of the SBDD constraint for the test vector, however, the value of BB is "known" along this path, as shown in Figure 4.7.

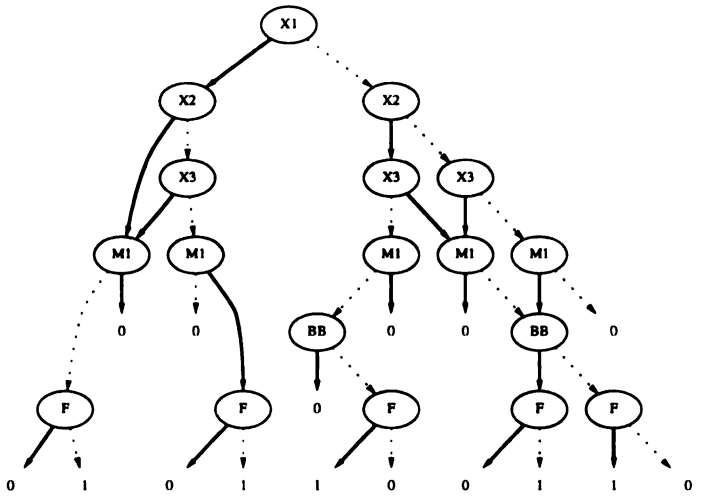


Figure 4.7: **SBDD for the partial description of `simplecircuit`.** This SBDD demonstrates the effect of applying the constraint $f(0,1,0) = 1$. Contrast this with Figure 4.4(c), which shows the SBDD before the constraint was applied.

- It may instantiate an instance of a decision variable node along a path in which that variable was previously a *don't care.* This occurs when the value of an unknown blackbox node becomes known somewhere along the path as a result of the introduction of new information.

  A trivial example of this effect can be seen in the previous example as well. In Figure 4.4, there is no decision node for X_3 along the path $X_1 = 0, X_2 = 1$; X_3 is a *don't care.* The new information represented in Figure 4.7 shows that the value of X_3 is no longer considered a don't care along that path, because its value has been determined to have an effect on the value of F and therefore BB.

- It may allow the deduction of a *secondary constraint.* (See Section 4.2.3.)

4.2.2 Using BDDs to discover specifications

Consider an example RFPS problem (Section 4.2.1) consisting of the partial schematic for **simplecircuit** presented in Figure 4.3 and the SIS-generated (Sentovich et al., 1992) ATPG test vectors for **simplecircuit** presented in Figure 4.8(a). Each test vector defines a simple relationship between a particular primary input assignment and a primary output assignment. The characteristic function representing this relationship (Definition 3.2) is false only for variable assignments for which the inputs take on the value specified by the test vector but for which the outputs do not. Discovering the specification of a blackbox component can be simplified though the use of the BDD representation of the structure and characteristic functions (as discussed in Chapter 3 and Section 4.2.1).

Constraint characteristic functions can be represented by BDDs such as the one in Figure 4.8(c), where the BDD represents the characteristic function of the relationship defined by the test vector $X_1 = 0, X_2 = 1, X_3 = 0 \rightarrow F_2 = 1$. The addition of each constraint that defines a previously unspecified relationship modifies the BDD representation of the structure function (Theorem 4.1). Figure 4.8(d) presents the result of applying the BDDs representing the test vectors in Figure 4.8(a) to the structural BDD of the partial implementation of **simplecircuit** (Figure 4.3(a)). Appendix A provides the details for this example.

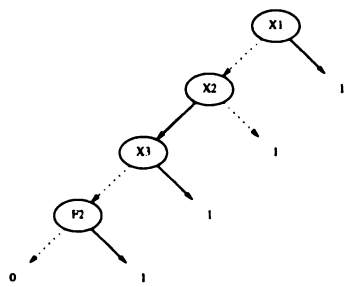
The goal of the RFPS problem is to determine the functionality of the missing component(s) so that the system can be redesigned. After a structure function is constructed and all relationships between internal variables are represented, there are

X_1	X_2	X_3	F_2
0	0	0	0
0	0	1	1
0	1	1	0
0	1	0	1
1	1	0	0

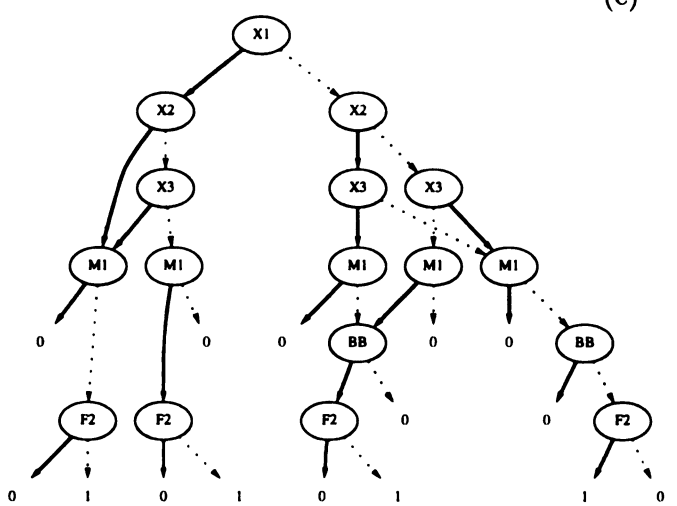
(a)

X_2	X_3	M_1	BB
1	1	0	1
0	0	1	1
1	0	0	0
0	1	0	0
otherwise			*d*

(b)



(c)



(d)

Figure 4.8: **The RFPS solution for `simplecircuit`.** (a) The ATPG test vectors for the **`simplecircuit`** RFPS Problem; (b) BB specified to within *don't care* conditions; (c) The BDD representing the characteristic function of the test vector $X_1 = 0, X_2 = 1, X_3 = 0 \rightarrow F_2 = 1$; (d) The BDD representing the structure function of **`simplecircuit`** after including the relationships defined in the test vectors.

four cases described below. Section 4.3 presents the algorithm that automates the identification of the appropriate case and the extraction of the blackbox specification from a structural BDD.

- *Case 1: The structure function represents the complete functionality of the circuit as well as that of the blackbox.* In this case, the exact functionality of the blackbox can be extracted from the BDD representation of the structure function.

- *Case 2: The structure function represents the complete functionality of the circuit but not of the blackbox.* The circuit functionality is completely specified if all paths though blackboxes have one arc leading to the 0-terminal and the other arc containing exactly one 1-path. In this case, the exact functionality of the blackbox is not necessarily defined under all of its input conditions. However, as the functionality of the circuit is fully defined, the unknown specifications of the blackbox are *don't cares.* Thus, the blackbox is specifiable to within *don't care* conditions, which is sufficient for reengineering.

  In other words, one identifies for which inputs the value of the blackbox affects the output, and what the value of the blackbox is under these inputs. For all other inputs (in which the blackbox output does not have an effect upon the circuit output), one does not necessarily know the exact value of the blackbox but does know that it is a *don't care* in terms of the overall circuit function. Examination of the SBDD in Figure 4.8(d) provides a specification for the functionality of the blackbox structure to within *don't care* conditions (Figure

4.8(b)).

- *Case 3: Conflicting information is detected.* The third case occurs when no value of a blackbox output can satisfy all constraints. If this situation occurs, one can identify the conflicting relationships but must resolve the situation externally.

- *Case 4: The solution is incomplete.* In the final case, neither the circuit nor the blackbox component(s) are fully specified. If, after constraints representing all available information has been applied, any blackbox node remains *unknown*, then the function in only partially specified. The SBDD represents the sum of all applicable knowledge about the circuit, but additional information is necessary to produce a full functional description.

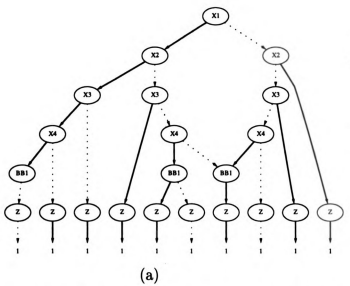
  The SBDD representation can be used to extract the set of necessary relationships that must be determined to complete the specification. If these relationships can be determined (perhaps through the use of a operational device or simulation of the circuit's behavior), the blackbox component can be specified to within *don't care* conditions without resorting to exhaustive testing.

4.2.3 Deduction of secondary constraints

When accumulated partial information dictates the necessary output of a blackbox under some input assignment, that knowledge may be able to be applied to resolve other unknown blackbox nodes. Since the behavior of the blackbox structure is consistent, the blackbox will produce the same output whenever its inputs have a particular

assignment. Whenever a blackbox node becomes known, a secondary constraint that represents the assignment of the direct inputs into the blackbox structure and the expected output of the blackbox is created. Application of this secondary constraint forces other unknown nodes representing the same blackbox structure to produce the correct output if their direct inputs have the appropriate assignment.

Figure 4.9 presents the SBDD for **schneider1** (Figure 3.10) after the SIS-generated test vectors for the circuit have been applied, as discussed in Section 4.2.1. The circuit is not fully specified since one node remains unknown. To fully specify the circuit, one of the two 1-paths beginning $X_1 \xrightarrow{1} X_2 \xrightarrow{0} X_3 \xrightarrow{0} X_4 \xrightarrow{1} BB_1$ must be determined illegal.



(a)

X_1	X_2	X_3	X_4	Z
1	0	1	1	1
1	0	0	0	1
1	1	1	1	0
1	1	0	1	1
0	0	0	1	1
1	1	1	0	1
0	0	0	0	0
0	1	1	1	1

(b)

Figure 4.9: **Reduced SBDD for the schneider1 circuit with constraints.** The SBDD (a) is the result of applying the SIS-generated ATPG test vectors (b) for the **schneider** circuit to the SBDD for **schneider1** (Figure 3.11). For clarity, arcs leading directly to the 0-terminal are not shown in the graphical representation.

Figure 3.10 shows that BB_1's primary inputs are X_2 and X_3. It can be determined

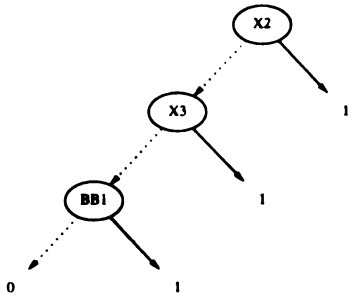
from the known 1-path $X_1 \xrightarrow{0} X_2 \xrightarrow{0} X_3 \xrightarrow{0} X_4 \xrightarrow{1} BB_1 \xrightarrow{1} Z \xrightarrow{1} 1$ that BB_1 has a value of 1 when its inputs (X_2, X_3) are $(0,0)$. Since the structure represented by BB_1 is combinational, it must take that same value along the unknown path $X_1 \xrightarrow{1} X_2 \xrightarrow{0} X_3 \xrightarrow{0} X_4 \xrightarrow{1} BB_1$ because that path also has $(X_2, X_3) = (0,0)$.

Applying this knowledge as a system-wide constraint allows one to determine the 1-path for the last unknown node. The function is now fully specified (Figure 4.10(b)). Furthermore, the structure BB_1 can be specified to within *don't care conditions* (*d*-conditions) (Figure 4.10(c)). That is, the necessary functionality of BB_1 in the circuit can be exactly specified, and the behavior of BB_1 under all of its input assignments in which it is not a *don't care* can be exactly identified. Finally, a specification of assignments under which BB_1 is a *don't care* can be determined, thereby allowing identification or reengineering of the previously unspecified component (Figure 4.10(c)).

The identification of such deduced constraints requires an examination of the entire SBDD. Its complexity therefore increases with the size of the graph. The complexity of the application of the deduced constraint likewise depends on the number of nodes in the SBDD.

4.2.4 Extensions to multiple blackboxes

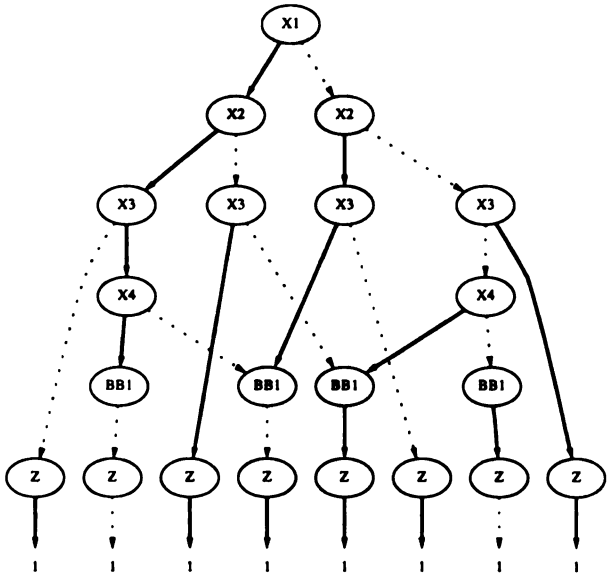
For clarity, the examples discussed in this thesis have been limited to circuits that contain no more than a single blackbox. The definitions, however, have been general and have not restricted the number of blackboxes that may exist in a device whose



(a)

X_2	X_3	BB_1
0	0	1
0	1	d
1	0	d
1	1	0

(c)



(b)

Figure 4.10: **Reduced SBDD for the `schneider1` circuit after deduction.** (a) the secondary constraint to be applied to the SBDD for **`schneider1`**; (b) a completely specified SBDD deduced by applying test vectors and the secondary constraint; (c) The blackbox BB_1 is specified to within d-conditions.

design is to be recovered. We now present an example that contains an unknown portion which must be represented by multiple blackbox variables. In this particular example, the blackbox variables are part of the same unknown structure and share a common set of inputs.

Figure 4.11 presents a partial schematic of an implementation of the **schneider** circuit (Figure 3.8) in which a the unknown portion has two outputs and thus each must be represented by a blackbox variable. Figure 4.12 presents the SBDD that represents the structure function and constraint functions generated from **schneider**'s SIS-generated set of ATPG test vectors.

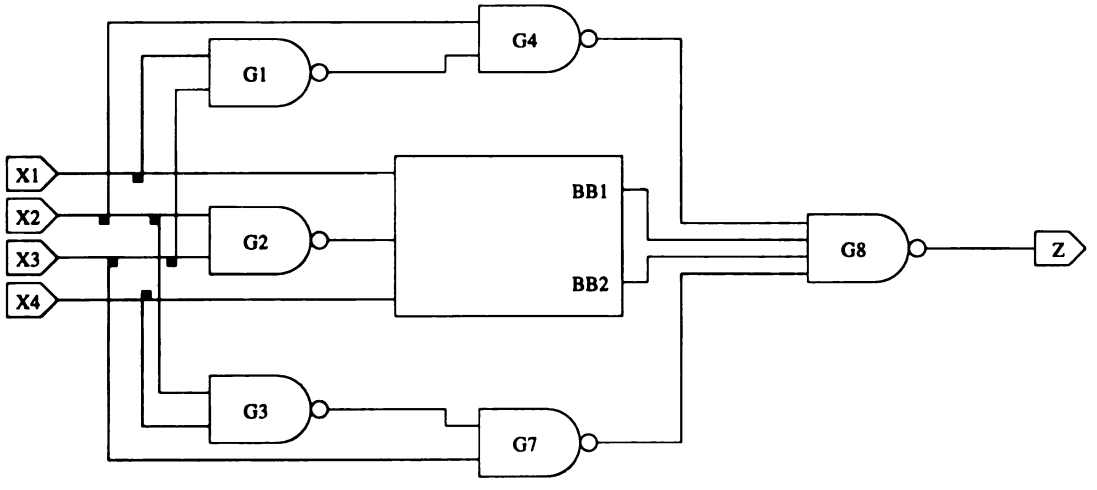


Figure 4.11: **Schematic for schneider2 RFPS problem.** This schematic presents a partial implementation of the schneider circuit (Figure 3.8). The functionality of a portion of the circuit is unknown, and thus multiple blackbox variables are necessary to represent the unknown structure.

In a circuit with multiple blackboxes, the deduction of secondary constraints (Section 4.2.3) is not necessarily straightforward. For each unknown node in an SBDD,

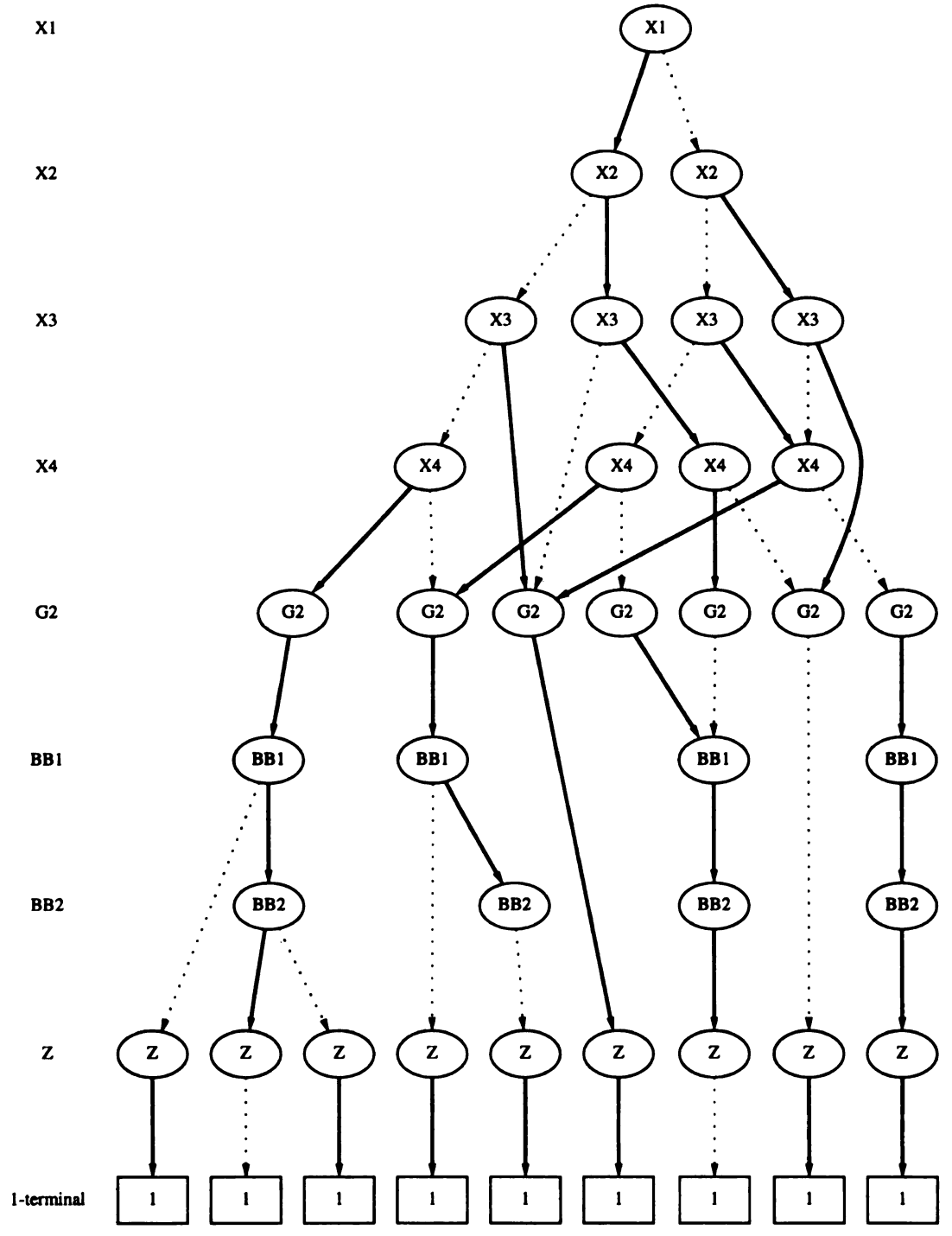


Figure 4.12: **SBDD for `schneider2` multiple-blackbox RFPS problem.** This SBDD representation specifies the deduced functionality of `schneider2` and both blackboxes after constraints (Figure 4.9(b)) and deduction are applied. The behavior of the blackboxes (and thus the circuit) remain unspecified for only three input assignments (1001, 1000, and 0001).

two subgraphs exists which represent the behavior of the rest of the system under the two possible behaviors of the blackbox node. Exactly one of the two paths leading from the node represents behavior that is legal in the actual circuit. When multiple blackboxes exist, seemingly "known" blackbox nodes (i.e., exactly one of the nodes children is the 0-terminal) may exist along a path which is not provably legal due to the presence an another (unknown) blackbox node along the path.

For example, consider the behavior of the blackbox variables $BB1$ and $BB2$ in Figure 4.12. Under the input assignment 1000 (i.e., $X_1 = 1$, $X_2 = 0$, $X_3 = 0$, $X_4 = 0$), G_2 is constrained to take the value 1 because the 0-edge from G_2 along that path leads to the 0-terminal. However, neither arc of the node representing the blackbox variable $BB1$ leads to the 0-terminal (i.e., the node is *unknown*). If, however, $BB1 = 1$ under this input assignment, the SBDD shows that $BB2 = 0$. For the input assignment 1000, $BB2$'s inputs (Figure 4.11) have the values $X_1 = 1, G_2 = 1, X_4 = 0$. We cannot, however, apply the relationship $BB2(1, 1, 0) = 0$ to the entire SBDD as a secondary constraint since the "truth" of that relationship depends upon the behavior of $BB1$, which remains unknown.

A secondary constraint can only be deduced from known blackbox nodes which belong to exactly one 1-path. Note that in systems with multiple blackboxes, a secondary constraint may itself generate "new" information, creating a tertiary constraint, and so on.

Overall, the design recovery of systems which contain multiple blackboxes proceeds in an identical fashion to those consisting of a single blackbox. Additional blackboxes do, however, increase the degrees of freedom available to the structure function. Each

additional blackbox can cause an exponential increase in the number of possible paths which must be proven legal or illegal. In some circumstances, a correspondingly greater amount of available information may be necessary to fully specify the behavior of the circuit.

4.3 Implementation and results

The techniques discussed in Section 4.2 were implemented in C using Carnegie Mellon University's BDD package (Long, 1996). This section presents an overview of a structural BDD implementation and initial results.

4.3.1 Implementation

This section discusses the algorithmic details of the initial implementation of a design recovery tool based on the SBDD methodology. This algorithm demonstrates one way in which to approach the RFPS problem presented in Section 4.2.1 using SBDD-based techniques.

As input, the algorithm requires only the mathematical relationships defined by the structure of the circuit (Definition 3.3) and available external information (Definition 4.1) (in this case, a set of test vectors). Our initial implementation automates the extraction of these relationships through examination of the partial system schematic and the ATPG test set. The algorithm uses SBDDs as representations of these mathematical relationships.

The possible outputs of the algorithm correspond to the the cases discussed in Sec-

tion 4.2.2. If possible, the system will specify the behavior or any blackbox structures to within *don't care* conditions. Otherwise, the algorithm will identify the deduced behavior as well as a minimal set of test vectors that, if determined, would allow the complete specification to be deduced. If conflicting information is detected during processing, the algorithm will halt with output describing the conflicting relationship. We present this algorithm as an example of one way in which SBDD-based techniques may be leveraged in design recovery solutions.

The SBDD Algorithm

Step 1: Initialization. Initialize the structural BDD G to logical one. Assign the primary inputs as BDD variables using some reasonable total order.

Step 2: Apply legal component or relationship constraint. A constraint is applied to G by constructing a BDD that represents its characteristic function (as defined in Definitions 3.2, 3.5, and 4.1) and by using the BDD `Apply` function with the **And** operator (Bryant, 1986). If the decision variable representing the output of a legal component constraint is not represented in G, it is introduced into G.

For building the structural BDD, component and relationship characteristic functions are considered identical; both represent relationships between net variables in the BDD. A *relationship constraint is legal* if all of the variables on which the relationship depends exist as BDD variables. A *component constraint is legal* if the variables that represent the component's inputs exist in the BDD. Because of the structure of combinational circuits, all constraints eventually become legal. Since each constraint must be constructed as an SBDD and composed with the structural BDD G using

the BDD **Apply** operation, the introduction of each constraint is an $O(|G|)$ operation (Bryant, 1985).

Step 3: Reduce BDD size. If all the constraints related to a particular variable have been applied, remove the variable from the structural BDD by using the smoothing operator (Definition 2.9). Variables that represent inputs to or outputs from a blackbox structure are not subject to this reduction. Variables representing the primary inputs and outputs of the structure are also nonreducible.

Although the functional constraints imposed by a structure whose variable has been removed from the BDD in this way are still represented in the BDD, the ability to directly reference or introduce new relationships which directly involve the removed variable is lost. Such removal, however, greatly increases the efficiency of the representation (Section 3.4.2). The application of the smoothing operator requires examination of each node to determine if it is subject to remove and is therefore an $O(|G|)$ operation.

Step 4: Test for conflict and repeat. Verify that at least 1-path exists for each assignment of primary input values. This examination requires a recursive traversal of the BDD and is therefore an $O(|G|)$ operation. If an assignment of primary input values exists for which no 1-path exists, a conflicting relationship has been introduced. In this case, the relationships is identified and the algorithm halts. Otherwise, if any constraints remain unapplied, return to Step 2.

Step 5: Determine completeness of specification. If the resulting structural BDD has a unique 1-path for each input vector, it is fully specified; proceed to Step 7. This examination requires a recursive traversal of the BDD and is therefore an

$O(|G|)$ operation.

Step 6: Acquire necessary knowledge. Consider each primary input assignment for which multiple 1-paths exist. For each such assignment, determine the value of all blackbox inputs by examining the paths. These blackbox input assignments (which may be duplicates) represent blackbox input conditions for which the output of the blackbox is unknown and not a *don't care*. The discovery of the behavior of the blackbox in these conditions will fully specify the blackbox. For each unique blackbox input condition, select its associated primary input assignment. If the corresponding circuit outputs for this set of input assignments can be determined, then the circuit can be fully specified by applying these relationships as constraints (goto Step 2). This step requires a recursive search of the BDD graph and is therefore an $O(|G|)$ operation.

Consider Figure 4.7 (page 89). The behavior of the overall functionality of `simplecircuit` represented by this BDD is unknown for all primary input assignments for which multiple 1-paths exist (namely 000, 001, and 011). Since each of these input assignments presents a different input assignment to the blackbox function $BB(X_2, X_3, M_1)$ (respectively 001, 010, and 110), each of these input conditions represents an unresolved vector. Determination of all unresolved vectors (provided in Figure 4.8(a)) leads to a complete specification of the overall circuit functionality (Figure 4.8(d)).

Step 7: Specify blackbox structures. Consider each node n in G that represents the output of a blackbox structure BB. Consider the value of the primary inputs as defined along the path from the root to n. If exactly one 1-path exits, then

the behavior of output b from structure BB is defined for the values of $BB's$ inputs that were defined on the path from the root to n. If more than one 1-path exists, then the value of b is explicitly unspecified and is not a *don't care*. b is a *don't care* for any input assignment to BB that is not known or explicitly unspecified. After producing the blackbox specification, halt. This step also requires a complete traversal of the BDD and is therefore an $O(|G|)$ operation.

Consider Figure 4.8(d), the value of the blackbox structure BB can be specified to *d*on't care conditions using this step to produce the truth-table presented as Figure 4.8(b).

4.3.2 Validity and complexity

The algorithm has a well defined next-step function. Furthermore, the properties specified in Theorem 4.1 (page 84) hold. Thus, the resulting SBDD must correspond to exactly one of the four cases specified in Section 4.2.2 (page 90). Clearly defined halting conditions are identified for each case. For the cases in which the complete specification of the circuit is represented (Cases 1 and 2), the algorithm halts in Step 7. For the case in which conflicting information is present (Case 3), the algorithm halts in Step 4. Otherwise (Case 4), the algorithm terminates in Step 6.

It remains to be shown that each step is well-defined and correct. Since SBDDs are simply a particularly efficient representation of possible solutions to a set of mathematical constraints and operations, the results are correct by construction. This section addresses the correctness of each step and thus the validity of the presented

SBDD algorithm.

Step 1: Initialization. In Step 1, the SBDD G is created. At initialization, the root node of G is the 1-terminal. This standard BDD operation takes $O(1)$ time using the standard BDD algorithm assignment operator and is guaranteed to complete (Bryant, 1985). The primary inputs $\mathbf{i}$ are added to the BDD manager as decision variables in some reasonable order (an $O(|\mathbf{i}|)$ operation). Any total ordering of the primary inputs is algorithmically correct, but the efficiency of the representation may be very sensitive to the variable order chosen (Bryant, 1985). Dynamic reordering techniques can be used at any time during the algorithm to modify the order of the primary inputs if necessary.

Step 2: Apply legal component or relationship constraint. In Step 2, constraint SBDDs are created and applied to the SBDD G. When applying constraints, variables for the inputs and outputs must be introduced into the BDD manager if they have not previously been included. Since the topology of the circuit itself forms a reasonable variable ordering (Section 3.4.1), such an ordering can be determined by performing a traversal of the Boolean network for the system (an $O(k)$ operation, where k is the number of structure output variables in the system).

For a known structure, an SBDD representing its characteristic equation (Definition 3.2) is constructed. For an unknown structure, an SBDD representing the characteristic function for the blackbox (Definition 3.5) is constructed. For any additional relationship specified, a constraint characteristic function (Definition 4.1) is constructed. The construction of the BDD representing these functions proceeds using standard BDD techniques for representing functions (Bryant, 1985). As no

relationship can involve more than k variables (where k is the number of structure output variables in the system), this operation has an average-case time complexity of $O(|k|)$ and a worst-case time complexity of $O(2^{|k|})$. This worst-case complexity is uncommon for most Boolean functions (Bryant, 1985).

Finally, the newly created SBDD constraint is applied to the SBDD G using the standard BDD **Apply** algorithm with the Boolean **And** operator. The composition is a $O(|G|) \cdot O(|F|)$ operation (Bryant, 1985). Since the definitions of the functions for each constraint are well defined, and only standard BDD operations are used, this step is correct and will complete.

Step 3: Reduce BDD size. In Step 3, the size of the SBDD G is reduced, if possible. The worst-case size for G is $O(2^{k+i})$, where k is the number of structure output variables in the system, and i is the number of primary inputs. Although the average size does not generally approach this worst-case size, no guarantees can be made. This reduction step is included to increase the efficiency of the representation, but is not a necessary component of the SBDD algorithm. The reduction of an SBDD by removing SBDD variables is discussed in Section 3.4.2. Once a variable has been removed, the introduction of a relationship which involves the variable will cause an error and the algorithm will halt. Therefore, removal of a decision node is not allowed until all relationships which directly involve its variable have been included in G. This step requires a well-defined graph traversal of G and is therefore an $O(|G|)$ operation.

Step 4: Test for conflict and repeat. In Step 4, the SBDD is examined to determine if any assignment of input variables leads directly to the 0-terminal. If such a case is found, the algorithm determines that the last relationship introduced has

caused a conflict with one or more relationships represented in the SBDD and halts. This test requires a well-defined graph traversal and is therefore an $O(|G|)$ operation.

After testing for conflict, the next available constraint is selected and the algorithm returns to Step 2. It is necessary to first apply the characteristic equations for all known and unknown systems structures in the partial order defined by the system topology. Only after introducing these relationships can additional information be introduced. By introducing relationships in this order, we guarantee that the "inputs" to each relationship are available and that their values are well defined prior to the introduction of additional information.

Step 5: Determine completeness of specification. In Step 5, the SBDD is examined for completeness. Since a fully specified SBDD has exactly one 1-path for each assignment of input variables, this determination can be made by a simple graph traversal. If any node in G representing a decision variable which is not a primary input does not have exactly one arc which leads to the 0-terminal, then the SBDD is not fully specified. This test requires a well-defined graph traversal and is therefore an $O(|G|)$ operation.

Step 6: Acquire necessary knowledge. In Step 6, the partially specified relationships of SBDD G are identified. This step is similar to the traversal performed in Step 5, except that the primary input assignment which appears in the path to each "unknown" node is recorded. The node is then marked to prevent redundant primary input assignments from being recorded. This test requires a well-defined graph traversal and is therefore an $O(|G|)$ operation. This set of primary input vectors represents input conditions for which the output behavior of the circuit is

unknown. Once the set of vectors has been determined, the algorithm produces the information, and halts.

If an external source can provide the behavior of the primary outputs under the input assignments represented by this set of vectors, this information can be used (in Step 2) to fully specify the SBDD. This claim is supported as follows. Each vector corresponds to an assignment of primary inputs which produces an assignment of blackbox inputs under which the behavior of the blackbox structure is unknown (this follows from the technique use to produce the set of vectors). If the circuit output assignment is unknown for the input assignment represented by the vector, then at least one output is sensitized to a blackbox variable which appears along the SBDD path. Therefore, by Theorem 4.1, the vector specifies a partial relationship upon the blackbox(es) to which it is sensitized.

The set of vectors can be reduced in size by examining the SBDD to determine the blackbox input conditions for the unknown node or nodes appearing on the path corresponding to each vector. For each path, the variable assignment for inputs to the unknown structure can be determined. Vectors in the set which share unknown structure input assignments are redundant since the generation of secondary constraints (Section 4.2.3) guarantees that information deduced regarding the output of any blackbox for some blackbox input assignment are uniformly represented throughout the SBDD. This reduction is done for efficiency only, and does not necessarily need to be included as part of the algorithm.

Step 7: Specify blackbox structures. In Step 7, a specification for the blackbox structure in a fully specified SBDD G is produced. This specification can be gen-

erated by recursively traversing G (which is an $O(|G|)$) operation). As each blackbox node is encountered, the blackbox input assignment to which it corresponds (this is specified by the path used to reach the node) is recorded. Furthermore, the output of the blackbox node is recorded. For all blackbox input assignments which are not recorded, the function of the blackbox is a *don't care.*

4.3.3 Results

The structural BDD-based approach has been applied to a number of benchmark combinational circuits (McElvain, 1993). Results are summarized in Table 4.1. The column labeled "Shared BDD Size" presents the size of the BDD(s) representing the functionality of the circuit, as implemented by an efficient complemented-edge, shared-BDD package (Somenzi, 1997). For each circuit, several blackbox scenarios were tested, including cases in which the subcircuit contains multiple blackboxes and multiple outputs. For each scenario, we present the size of the structural BDD and the CPU time necessary to construct it. Although the representation of the circuit's structure function is significantly larger than a traditional BDD representation, this complexity is the necessary cost of providing a framework for representing partial information.

Since net variables which represent nonessential circuit structures are reduced once it is no longer necessary to reference the variable (Step 3 of the algorithm), the size of the SBDD becomes relatively independent of the number of gates in the circuit. The limiting factors on its size are the number of primary inputs and outputs, and

Circuit Name	No. of Inputs	No. of Outputs	No. of Gates	Shared BDD Size
alu4	14	8	681	1453
f51m	8	8	43	73
pm1	16	13	39	42
t481	16	1	2072	202
z4ml	7	4	20	47

Circuit Name	Unknown Gates	Structural BDD Size	Unresolved Vectors	CPU Time (sec)
alu4	0	1663	0	4.6
	2	1774	0	7.61
	4	1779	7	8.8
f51m	0	765	0	17.1
	5	3057	0	20.9
	10	3898	12	20.9
pm1	0	1101	0	45.4
	4	1980	0	60.3
	8	3605	0	78.9
t481	0	204	0	294.78
	10	441	4	353.1
	50	5061	68	1738.3
z4ml	0	80	0	3.8
	3	173	0	4.5
	10	215	10	5.3

Table 4.1: **RFPS results.** Structural BDD sizes and run times to discover the functionality of partially specified components removed from benchmark circuits.

number of essential internal structures represented (Section 3.4.2).

The CPU time (indicated in CPU seconds on a SPARCStation 20) indicates the time needed to (1) build the SBDD from a partial netlist description, (2) apply the relationship constraints for the circuit's ATPG test vectors, and (3) provide a description of the blackbox functionality or indicate additional sufficient tests. In many of the cases, the additional information provided in the form of a circuit's ATPG test vectors was insufficient to allow complete deduction of the overall circuit functionality. The number of "Unresolved Vectors" column indicates the number of input/output relationships which must be determined in order to specify the behavior of the blackboxes to within don't care conditions and to therefore specify the functionality of the overall circuit. Note that the algorithm allows the exact determination of the needed additional input/output relationships.

It is necessary to mention that the number of unresolved vectors is significantly influenced by the selection of the unknown gates. The number of blackbox structures, the numbers of inputs and outputs of each structure, and the importance of each unknown structure's role in the overall circuit functionality are all factors which affect the difficultly of determining the complete functionality of the circuit. Since no existing techniques provide an effective solution to this problem, we provide these preliminary results simply to demonstrate the feasibility of this approach.

These preliminary experiments have demonstrated the feasibility of this technique. The blackbox specifications produced are provably correct by construction. Since this problem is inherently intractable, however, this approach must fail for problems of a certain size or complexity.

Chapter 5

Semantic equivalence checking

The problem of identifying meaningful components from a gate-level description has been identified as a critical research area in the design recovery process (Section 1.2.3). Of particular interest is identifying a cluster of connected low-level devices that form a high-level component. Previous approaches to this problem have relied on the identification of exact structural matching (syntactic matching) to identify subcircuits.

Semantic techniques can be used to identify subcircuits that are equivalent to a high-level component in many situations for which syntactic techniques fail (Eckmann and Chisholm, 1997). For example, the structural changes imposed by new implementations, design optimizations for area and power, or many other complicating factors can cause purely syntactic techniques to fail, but such structural changes are amenable to semantic techniques. Semantic matching consists of two primary tasks: the identification of subcircuits which may be block-level modules, and the determination of semantic equivalence between such a subcircuit and a block-level

module. The first of these tasks is addressed elsewhere (White et al., 1997). This chapter focuses upon the efficient determination of semantic equivalence.

Although semantic techniques are not limited to any particular level of circuit description or application, this dissertation considers only the identification of high-level components from gate-level netlists. The methods presented here are restricted to identifying the block-level functionality of synchronous combinational components with no loops or other timing issues. Since combinational circuits are the basis of various logic circuits, the transformation of combinational netlists to a higher level of design (a netlist of high-level components and glue-logic) will provide a basis for understanding sequential circuit functionality in the future.

5.1 Determining equivalence

Identifying the subcircuits in a detailed circuit description is a fundamental operation in both circuit validation and design recovery. Existing techniques for such identification in design recovery rely on finding an exact match for a subcircuit structure within the description (Section 5.2.1). These techniques fail to identify subcircuits that are functionally equivalent but have been obfuscated because a different technology has been used or because the design has been optimized.

Following the notation commonly used in Boolean Matching (Benini and De Micheli, 1997), consider some subcircuit (or *cluster*) of a combinational circuit. Such a subcircuit has $|\mathbf{i}|$ inputs such that, $\mathbf{i} = \langle i_1, \ldots, i_{|\mathbf{i}|} \rangle$, $|\mathbf{o}|$ outputs such that, $\mathbf{o} = \langle o_1, \ldots, o_{|\mathbf{o}|} \rangle$, and a vector of Boolean functions (the *cluster function*) that deter-

mines the relationships among them:

$$\mathcal{F}(\mathbf{i}) = \langle \mathcal{F}_1(\mathbf{i}), \ldots, \mathcal{F}_{|\mathbf{o}|}(\mathbf{i}) \rangle. \tag{5.1}$$

Likewise, for any high-level component module with inputs $\mathbf{x}$ and outputs $\mathbf{y}$, there exists a vector of Boolean functions (the *pattern function*) that describes its behavior:

$$\mathcal{G}(\mathbf{x}) = \langle \mathcal{G}_1(\mathbf{x}), \ldots, \mathcal{G}_{|\mathbf{y}|}\mathbf{x}) \rangle. \tag{5.2}$$

Two bijections, π_I, the *input permutation function*, and π_O, the *output permutation function* are defined as follows:

$$\pi_I : \{i_1, \ldots, i_{|\mathbf{i}|}\} \rightarrow \{x_1, \ldots, x_{|\mathbf{x}|}\} \tag{5.3}$$

$$\pi_O : \{\mathcal{F}_1, \ldots, \mathcal{F}_{|\mathbf{o}|}\} \rightarrow \{\mathcal{G}_1, \ldots, \mathcal{G}_{|\mathbf{y}|}\}. \tag{5.4}$$

Definition 5.1 *Two vectors of Boolean functions $\mathcal{F}$ and $\mathcal{G}$ are* input-permutation, output-permutation equivalent *($\mathcal{PP}$-*equivalent*) if bijections exist such that:*

$$\forall_k, 1 \leq k \leq |\mathbf{o}|, \mathcal{F}_k(\mathbf{i}) = \pi_O(\mathcal{F}_k)(\pi_I(\mathbf{i})). \tag{5.5}$$

We can now define semantic equivalence for combinational designs.

Definition 5.2 *Two combinational designs D_1 and D_2 with corresponding vectors of Boolean functions $\mathcal{F}$ and $\mathcal{G}$ are* semantically equivalent *if and only if $\mathcal{F}$ and $\mathcal{G}$ are $\mathcal{PP}$-equivalent. The input bijection π_I and the output bijection π_O under which $\mathcal{F}$ and*

$\mathcal{G}$ are $\mathcal{PP}$-equivalent describe the input and output correspondences between D_1 and D_2.

This definition of semantic equivalence provides a mechanism for identifying subcircuits that are functionally equivalent, irrespective of obfuscating details.

Definition 5.3 Semantic matching *is the process of identifying a block-level component which is semantically equivalent to a combinational subcircuit described at the gate-level (i.e., a netlist).*

Transforming detailed gate-level circuit descriptions into corresponding descriptions based on block-level modules depends on enumerating all of the candidate subcircuits within the original detailed description and determining semantic equivalence for each candidate (Eckmann and Chisholm, 1997; Doom et al., 1998). This chapter is primarily concerned with presenting efficiency issues and solutions related to determining semantic equivalence.

5.2 Historical perspective

This section of the dissertation covers some common terms, major issues, and published techniques related to the identification of high-level entities necessary for design recovery. It describes some existing algorithms that have been used to solve some instances of the equivalence problem.

5.2.1 Syntactic matching

Previous approaches to this problem have relied on the discovery of subgraph isomorphisms to identify meaningful subcircuits (Bochner, 1988; Luellau et al., 1984; Ohlrich et al., 1993). In these approaches, the circuit is represented as a labeled graph. Likewise, a graph representing some subcircuit of interest is defined. These approaches then attempt to find all instances of the defined subcircuit in the graph representing the full circuit. In other words, these approaches attempt to identify clusters of gates that exactly match some search pattern.

Such syntactic techniques have been successfully used to identify isomorphisms between structures in a graph representing a circuit description and a particular implementation (or a set of implementations) of a block-level module. For any given block-level module, certain implementations are available in technology-specific libraries. By performing a syntactic match for every implementation of every module available in a technology library, all standard implementations of block-level modules implemented with that library can be identified.

The advantage of syntactic matching is that it is exceptionally efficient. Subgemini is an algorithm for isomorphic subgraph matching with complexity linear to the graph size (Ohlrich et al., 1993). The disadvantage of syntactic matching, in general, is that a syntactic algorithm can identify only the set of specific implementations of a functional component that are contained in its library. Nonstandard or intentionally obfuscated implementations will never be recognized. Furthermore, any optimization that modifies the implementation of the entity (such as optimizations for *don't care*

conditions) renders the entity unfit for recognition by structural techniques. Syntactic matching cannot reliably recognize all the functional components that exist in a circuit.

Although they are useful in applications such as converting a transistor netlist into a gate netlist, techniques that rely on exact structural matching have limited application to levels of design in which components have many valid implementations. Therefore semantic techniques which allow the identification of block-level modules based upon their functionality, rather than their structure, must be considered.

5.2.2 Factorial permutation

Although the equivalence of two single-output functions represented as ROBDDs can be tested in constant time (Bryant, 1985), such testing requires that the correspondences between the input variables be clearly identified. Because input and output variable correspondences are not generally available, the straightforward method for determining if two multiple-output functions are $\mathcal{PP}$-equivalent is to test for equivalence over the set of $|\vec{i}|! \cdot |\vec{o}|!$ possible pairs of bijection functions (*i.e.*, over all input and output permutations). When there are more than seven or eight inputs, the straightforward permutation technique is computationally intractable.

5.2.3 Logic verification

In logic verification, a *specification* describing some functional behavior is compared with a circuit *implementation* of that function to prove equivalence. Verification

techniques that can deal with problems involving large numbers of inputs, sequential behavior, and significant numbers of intermediate gates do exist. However, such techniques require that correspondences between the implementation and specification be known (Lai et al., 1992). Since one cannot assume knowledge of such correspondences when attempting to identify high-level components in a flat netlist, verification techniques are generally not applicable.

5.2.4 Boolean matching

Technology mapping (also known as *cell-library binding*) is part of the synthesis process that is commonly used to transform logic representations into interconnections within a set of implementation-dependent cells. Technology mapping is used to create cost-optimized implementations for some logic function or Boolean network in a particular implementation style that defines some library of building blocks (cells). Detecting the equivalence of these Boolean functions to cells, referred to as *Boolean matching*, is well studied (Benini and De Micheli, 1997).

In many ways, the problem of determining equivalence between a combinational circuit and a high-level entity library is similar to the problem of Boolean matching. Boolean matching algorithms are designed to efficiently match small (fewer than six inputs), single-output clusters with a component of their cell libraries that implements the function at the least cost. However, although a general solution to the equivalence problem must be able to efficiently match functions with any number of inputs and outputs, it also needs to be concerned with only a single (although possibly multiple-

output) pattern function rather than attempting to find a "least cost" match from an entire library of matching functions. The goal of semantic matching is not to find the best implementation of a function from a set of possible implementations but only to identify equivalence and variable correspondences between a particular subcircuit and a particular high-level component. It appears that no suitable solution to this problem has been proposed in the literature.

5.2.5 Boolean signatures and filters

A *signature* of a Boolean function is a unique characteristic representation of some property of the function. Two otherwise unrelated functions can have the same signature. Having equal signatures is a necessary condition for equivalence matching. Functions that share a signature are said to share a *signature class.*

A *signature function* is a function that takes an arbitrary function as an input and returns a characteristic signature for that function. The value of a signature function must be determined only by the behavior of the generic function; variable order, variable labels, and random elements may not be used as part of the determination.

Boolean signatures have been used successfully to increase the efficiency of Boolean matching algorithms (Mailhot and De Micheli, 1993). Since sharing a signature class is a necessary condition for equivalence, the matching of signature functions can be used to eliminate functions from equivalence consideration. Functions that do not have matching signature characteristics can be filtered from the search space since they cannot be equivalent. The primary limit to the effectiveness of such filtering is

the complexity cost of the signature function.

The use of filtering techniques in Boolean matching (Mailhot and De Micheli, 1993) has resulted in the discovery of a wide variety of signature tests by various researchers. Using signatures as filters to eliminate some permutations from consideration can appreciably reduce the complexity of a $\mathcal{PP}$-equivalence check (Definition 5.1). There are two classes of signatures: those that provide information regarding the behavior of input variables (input signatures), and those that provide information regarding the behavior of output variables (output signatures). Some of these signatures are discussed briefly here because they provide directions for future research. Lai et al. (1992) contains additional details.

For any function $f(x_1, \ldots, x_n)$, represented by BDD G of size $|G|$, the following signatures are defined:

- Cardinality of dependence set: The *dependence set* of a function consists of only those input variables that have an effect on its value. Thus,

$$Dep(f) = \{x_i | f(\ldots, x_{i-1}, 0, x_{i+1}, \ldots) \neq f(\ldots, x_{i-1}, 1, x_{i+1}, \ldots)\}. \qquad (5.6)$$

  The output signature $F_{dep}(f) = |Dep(f)|$ can be computed in $O(|G|)$ time by using a BDD-based algorithm (Lai et al., 1992). This signature is particularly useful when output-permutation equivalence is being determined. Outputs can be permuted only with outputs of the same dependence set cardinality. Functions that do not have the same number of outputs in each cardinality class cannot be equivalent.

- Cardinality of on-set: The *o*n-set of a function consists of all input assignments that produce a true (on) output.

$$F_{on} = |\{\mathbf{x}|f(\mathbf{x}) = 1\}|. \tag{5.7}$$

Since the range of F_{on} is quite large ($2^{|\mathbf{x}|}$), the cardinality of the on-set is one of the more effective Boolean signature functions. This signature can be computed in $O(|G|)$ time by using existing algorithms (Lai et al., 1992). This signature can be used to reduce both the number of output matchings between multiple-output functions and the number of input permutations.

- Unateness of input variables: A *binate* variable is present in both its direct and complemented phase in the minterm expression for a function. A *unate* variable is present in either its direct or complemented form, but not both. Thus the unateness of each input variable can be used as a signature $F_{unate}(f, x) =$ {*binate, positive unate, negative unate*}. For two functions to be equivalent, corresponding input variables must have similar unateness properties.

  The unateness of input variables can also be used as an output signature. For each output, count the number of binate, positive unate, and negative unate input variables that occur in the function's minterm expression. Its matching function must share these same sums. Computing the unateness of each input variable for each output function is a $O(|G|^2)$ operation (Lai et al., 1992), which may be too expensive for semantic matching.

- Symmetry class of input variables: Two variables are *symmetric* if they can be interchanged without changing the value of the function. Thus x_i and x_j are symmetric if and only if $f(\ldots, x_i, \ldots, x_j, \ldots) = f(\ldots, x_j, \ldots, x_i, \ldots)$. The input variables can be partitioned into symmetry classes that act as a signature for each output function. In addition, input variables can be matched only to input variables that have equivalent symmetry classes over all output functions. Symmetry computation requires an $O(|G|^2)$ operation (Lai et al., 1992) for each pair of inputs for each function and is probably too expensive for semantic matching. Although they can be effective for small cells in Boolean matching problems, symmetry classes are not always an effective signature on high-level entities, many of which have few symmetries.

  High-level entities, however, often possess *group symmetries*. When some group of input variables can be interchanged with a disjoint group of input variables without changing the value of the function, the groups of variables are said to be *group symmetric*. Group symmetries are also signature functions that might be particularly effective on high-level entities representing arithmetic functions. However, no algorithm is currently known for efficiently calculating group symmetries.

5.3 Determining semantic equivalence efficiently

This section describes an algorithm for determining if a semantic match exists between a subcircuit and a high-level component. A general solution to the equivalence

problem requires the identification of high-level components that are more complex then those dealt with in Boolean matching but that lack the input/output correspondences between the logic design and the library components that verification techniques require. Since the functionality of the high-level component may be represented in any number of structural forms, the subcircuit must be identified by proving semantic equivalence (Eckmann and Chisholm, 1997). Since semantic equivalence is defined as the process of determining equivalence between any pair of Boolean functions, these techniques are not limited to any particular level of circuit description or application. This thesis, however, considers semantic equivalence only in the context of the identification of high-level components in gate-level netlists.

5.3.1 Input signatures and suspect sets

This approach to the semantic matching problem makes use of signature information to reduce the number of input correspondences that must be considered. This reduction is accomplished through the use of suspect sets.

As discussed in Section 5.2.5, a signature of a Boolean function is a unique, characteristic representation of some property of the function. The class of signatures that provide information regarding the behavior of a function's input variables are referred to as *input signatures.*

Definition 5.4 *The signature values for any input signature function can be used to partition the function inputs into classes corresponding to their signature. Such a list of inputs is a* signature class.

The following result is clear.

Theorem 5.1 *A pattern function and a cluster function cannot be semantically equivalent under any input correspondence in which any pair of corresponding inputs are members of differing signature classes.*

Proof: Consider a cluster function and a pattern function which are semantically equivalent. Assume that some pair of corresponding inputs (say y and x, respectively) exist which are not members of signature classes with equal signature value. Since y and x do not share a signature class, some input signature function f exists such that $f(y) \neq f(x)$. This contradicts that fact that y and x are corresponding inputs of equivalent functions and must therefore have the same signature for all input signature functions. Therefore, our assumption was incorrect; pairs of corresponding inputs must be members of signature classes with equal signature value.

Definition 5.5 *A cluster input variable i_k's* suspect set, *S_{i_k} is the subset of pattern function $\mathcal{G}$'s inputs, $x_1, \ldots, x_{|\mathbf{x}|}$, that share a signature class with i_k under every input signature for which information is available.*

The use of suspects sets allows one to significantly reduce the factorial search space associated with determining semantic equivalence.

5.3.2 Vector input signature

A new signature function that has proven to be an adequate initial filter for many problems is introduced here. It takes advantage of the fact that the vector functions under consideration consist of multiple functions, each corresponding to a single output.

Definition 5.6 *A positive (negative) Boolean* unit vector *is defined as a vector in which exactly one element has the value 1 (0) and in which all other elements have the value 0 (1).*

Definition 5.7 *For any vector of Boolean functions* $\mathcal{F}(\mathbf{i}) = \mathbf{o}$, i_j*'s positive unit* vector input signature *is the sum of the function outputs (i.e., the cardinality of the on-set) when the positive unit vector with input* i_j *equal to 1 is applied.*

$$\mathcal{F}_{+vec}(i_j) = \sum_{n=1}^{|\mathbf{i}|} \mathcal{F}_n(\mathbf{u}), \text{ where } u_k = 1 \text{ if and only if } k = j. \tag{5.8}$$

The negative unit vector input signature is defined similarly.

Definition 5.8 *For any vector of Boolean functions* $\mathcal{F}(\mathbf{i}) = \mathbf{o}$, *the function's* vector signature *is an ordered set of* $|\mathbf{i}|$ (x, y) *pairs, in which each such pair corresponds to an input* i_j *of* $\mathcal{F}$ *and* x *(*y*) represents the positive (negative) unit vector input signature.*

Table 5.1 shows the results of applying the vector signature to the vector function of a 4-bit ALU. The resulting vector signature is $\{2 \times (1,7), 1 \times (2,2), 1 \times (2,5), 6 \times (2,7), 3 \times (3,5), 1 \times (6,5)\}$. The details of computing this vector signature are presented in Appendix B.

Input Name	Vector Input Signature	
	Positive	Negative
sel1, Cn'	1	7
sel3	2	2
a0	2	5
sel0, sel2, b0, b1, b2, b3	2	7
a1, a2, a3	3	5
m	6	5

Table 5.1: **Vector input signature for the TI 54181 4-bit ALU.** The positive and negative unit vector input signatures are shown for a 4-bit ALU with selection inputs sel0-3, mode input m, carry input Cn', and data inputs a0-3 and b0-3. The vector signature partitions the function inputs into six signature classes.

The positive and negative signature functions each return a signature which ranges in value from 0 to $|o|$ (where $|o|$ is the number of function outputs). These functions alone can be used to partition inputs into $|o| \cdot |o| = |o|^2$ suspect sets. Therefore these signatures become potentially more effective as the number of function outputs increases and are particularly well-suited to the multi-output functions which represent block-level modules in semantic matching.

Non-unit vector input signatures

When any signature class contains a single member (that is, no other input shares its (x, y) signature value), a correspondence is clearly identifiable. The vector signature for the four-bit ALU shown in Table 5.1 has three signature classes with only a single member (the signature classes for *sel*3, *a*0, and *m*). Recognizing correspondences for such variables is straightforward. A signature class with multiple members, however,

does not differentiate among the inputs sharing the signature class. Such differentiation may sometimes be achieved through the use of additional non-unit vector input signatures.

In the case of the four-bit ALU, six inputs are members of the suspect set $< 2, 7 >$. Let us consider other input vectors which can be applied to provide information to further differentiate these six inputs. The correspondence between $sel3$ and the pattern function input to which it correspondences is clear, since only one input is a member of the appropriate suspect set. Similarly, for each suspect set, a corresponding number of pattern function inputs exist which have the same signature values. This ability to partition both the inputs in the pattern function, as well as the inputs in the cluster function, provides a way for us to create new vectors to use to produce signatures.

Let us consider how we might attempt to create a new signature which would help differentiate the cluster input variable $sel0$ from the rest of the variables in suspect set $< 2, 7 >$. Consider the vector corresponding to the assignment $sel1 = Cn' = 1$, $sel3 = 0$, $a0 = 0$, $a1 = a2 = a3 = 1$, $m = 0$, and $sel0 = sel2 = b0 = b1 = b2 = b3 = 0$ to the cluster function inputs. Since the corresponding groups of input variables in corresponding suspect sets are clearly identified in the pattern function, it is possible to create an assignment of values to pattern function inputs which exactly corresponds to this vector. Consider now, the effect of changing the value of one of the inputs in the suspect set $< 2, 7 >$ to 1 and noting the output sum. The output-sum may differ as different inputs are assigned the value 1. Thus, this technique can be used as a signature to differentiate the inputs from each other. This same technique can

be used with the inputs in the corresponding pattern function suspect set. If, for example, the cluster functions sum-of-outputs for the vector in which $sel0 = 1$ does not equal pattern function's sum-of-outputs for the vector in which $b0 = 1$, then $sel0$ and $b0$ do not correspond and can be placed in different suspect sets.

The ability to distinguish inputs in different suspect sets from one another provides a means for the creation of additional vector input signatures. Like the positive and negative unit vectors, these additional non-unit vectors can be used to in a sum-of-outputs signature function to provide additional information which may define new suspect sets. These effective use of non-unit vectors has not been fully explored and remains open.

5.4 Implementation and results

This section presents an overview of an algorithm for determining semantic equivalence between two gate-level structures and preliminary results.

5.4.1 Implementation

This approach to the semantic matching problem makes use of signature information to reduce the number of input correspondences that must be considered. This reduction is accomplished though the use of suspect sets (Definition 5.5).

Let $\mathcal{F}(\mathbf{i}) = \mathbf{o}$ be the vector of Boolean functions for some subcircuit. Let $\mathcal{G}(\mathbf{x}) = \mathbf{y}$ be the vector of Boolean functions for a high-level component. Semantic equivalence and input/output correspondences between the subcircuit and the high-level compo-

nent can be determined as described by the following algorithm.

Semantic Equivalence Algorithm

Step 1: Create binary decision diagrams. Create a BDD for each "output" o_j of the cluster function $\mathcal{F}$. Likewise, create a BDD for each "output" y_j of the pattern function $\mathcal{G}$.

Step 2: Determine signature classes. Determine the vector signatures for the inputs of $\mathcal{F}$ and $\mathcal{G}$. Partition the set of each function's input variables into equivalence classes defined by these signatures. Compare the number of inputs from each function which return the same signature. If this number differs for any signature value, the functions cannot be equivalent; the algorithm returns a negative result and halts.

Step 3: Determine suspect sets. For each input i_j of the cluster function $\mathcal{F}$, create a suspect set S_j which contains the subset of inputs of pattern function $\mathcal{G}$ that have the same signature as the input i_j. Apply additional input signatures (Sections 5.2.5 and 5.3.2) to reduce suspect set size below a predetermined threshold, if possible.

Step 4: Iterate though legal input correspondences. Eliminate all matchings that include a correspondence between a cluster function input i_j and any pattern function input which is not in S_j. For each remaining input correspondence, attempt to identify a output correspondence under which the functions are equivalent (Step 5). After all input correspondences have been considered, any correspondences accepted in Step 5 are produced as output, and the algorithm halts. If no correspondences under which the functions are equivalent has been identified, the algorithm returns a

negative result and halts.

Step 5: Determine legal output correspondences. Compare each pair of BDDs representing a substituted cluster function output and a pattern function output. If a unique output matching for each pair is determined, a legal correspondence has been identified; the algorithm accepts the identified correspondence, and returns a positive result.

5.4.2 Validity and complexity

We now present a proof of validity for the semantic equivalence checking algorithm.

Theorem 5.2 *The Semantic Equivalence Algorithm (Section 5.4.1) will determine an input correspondence under which function $\mathcal{F}$ and function $\mathcal{G}$ are semantically equivalent if and only if such a correspondence exists.*

Proof:

Two functions are semantically equivalent if any only if a correspondence between their respective inputs and outputs can be found under which the functions are $\mathcal{PP}$-equivalent (Definition 5.2). Obviously, if every possible correspondence is checked (as in the factorial permutation approach described in Section 5.2.2), a correspondence under which function $\mathcal{F}$ and function $\mathcal{G}$ are equivalent will be found if (and only if) such a correspondence exists. Thus, we need only prove the following:

A. The Semantic Equivalence Algorithm does not discard any correspondence under which functions $\mathcal{F}$ and $\mathcal{G}$ are equivalent (i.e., no

false negatives).

B. The Semantic Equivalence Algorithm does not accept a correspondence under which $\mathcal{F}$ and $\mathcal{G}$ are not equivalent (i.e., no false positives).

Proof of A: Assume that $\mathcal{F}$ and $\mathcal{G}$ are equivalent functions under one or more input and output correspondences. Clearly, if all input correspondences and output correspondences are tested (as is done in the factorial permutation approach discussed in Section 5.2.2), then each such correspondence will be identified (and only such correspondences). As the semantic equivalence algorithm attempts to perform a more efficient search of the input and output correspondence space by removing some correspondences from consideration, we need only show that the correct correspondence is not removed from consideration.

The algorithm removes from consideration only those correspondences which take place between input variables which are members of non-corresponding suspects sets and thus possess differing input signatures for at least one signature function. Theorem 5.1 states that two functions can be semantically equivalent only under input correspondences which take place between members of their respective signature classes that have equal signature value. Therefore, the functions $\mathcal{F}$ and $\mathcal{G}$ cannot be equivalent under the correspondences that are removed from consideration.

Proof of B: Assume that $\mathcal{F}$ and $\mathcal{G}$ are not equivalent functions under any

input and output correspondences. If all input correspondences and output correspondences are tested (as is done in the factorial permutation approach), then the functions will fail to be equivalent under every correspondence tested. Thus, it will have been proven that the functions are not equivalent.

The semantic equivalence algorithm is only capable of discarding correspondences which would normally be checked during the factorial permutation approach. No additional correspondences are added. Thus, non-equivalent functions $\mathcal{F}$ and $\mathcal{G}$ will always be identified as such.

Complexity

The factorial permutation technique presented in Section 5.2.2 requires $|\mathbf{i}|!|\mathbf{o}|!$ comparisons. The Semantic Equivalence Algorithm achieves significant improvement. Let n represent the cardinality of the largest input suspect set determined in Step 3 of the algorithm. An upper bound on the number of legal input correspondences is $n!^{|\mathbf{i}|/n}$. The Semantic Equivalence Algorithm, however, constrains the value of n to be less than a predetermined threshold. Reasonably small values of n can be achieved for many problems through pruning suspect set sizes by applying multiple signature values until all suspect set sizes fall below some threshold (say, less than seven). Since n has an upper limit (the threshold) n can be treated as a constant value c, and the input correspondence selection will be exponential in complexity: $O(c^{|\mathbf{i}|})$.

Pruning of suspect sets to reduce n below some constant threshold is effective in most components except those with large numbers of symmetric inputs (which

are indistinguishable under Boolean signatures). In such cases, however, *any* input matching will succeed for the symmetric inputs, which actually simplifies the process of proving semantic equivalence because a correspondence will be identified very early in the execution of the algorithm.

Although BDDs are an efficient mechanism for representing the functionality of most components, their size may become intractably large for certain functions under some (or all) variable orderings (Bryant, 1985). Since a "good" variable ordering for the pattern function library can be obtained *a priori*, most BDD-based concerns can be eliminated. If the BDD for any cluster function output exceeds the size of the largest BDD representing a pattern function output, that input matching can be immediately discarded and BDD generation can be discontinued since no legal correspondence can exist between functions that have BDDs of different sizes under the same variable ordering. Pathological functions (such as multipliers) that have no efficient BDD representation remain an open issue.

Since each cluster output BDD is tested against each pattern output BDD exactly once in Step 5 of the algorithm, the complexity of determining legal output correspondence is only $O(|\mathbf{o}|^2)$. Therefore, the overall complexity of this approach is $O(c^{|\mathbf{i}|}|\mathbf{o}|^2)$. This exponential algorithm is a significant improvement over factorial methods and makes semantic matching feasible for many block-level modules.

5.4.3 Results

The algorithm for semantic matching was implemented in C using the University of Colorado's decision diagram library (Somenzi, 1997). Experiments were conducted on a Sun Ultra Enterprise 3000 running Solaris 2.5.1 with 256 MB of main memory and 879 MB of virtual memory. Experimental circuits were taken from the LGSynth93 benchmark suite (McElvain, 1993).

Table 5.2 compares the Semantic Equivalence Algorithm with the factorial approach. For each component, the table shows the size of the subcircuit, size (number of decision nodes) of the BDD representation of the component's pattern function (under some reasonable variable ordering), number of input matchings, and total number of BDD equivalence checks made during the program's run time. Since some functions contain symmetries, it is not sufficient, in general, to only identify a single correspondence when multiple correspondences exists. In design recovery, for example, it is necessary to identify all possible correspondences so that the "best" correspondence can be used in the recovered design description. Therefore, the algorithm always performs a complete search of the correspondence space and this is reflected in the run time presented.

The **z4ml** circuit (a 3-bit adder) shows a case in which the inputs are indistinguishable from their vector signature, and thus the number of input matchings is 7!. Note that because the algorithm prunes (i.e., reduces) the output search space, the number of comparisons is only 20,304, an order of magnitude less then the number of comparisons necessary in a 120,160 (7!4!) nonpruned search.

Circuit Name	No. of Inputs	No. of Outputs	BDD Size
C1908	33	25	127349
alu2	10	6	231
cc	21	20	57
f51m	8	8	73
pm1	16	13	42
sct	19	15	102
t481	16	1	202
z4ml	7	4	47

Circuit Name	Input Matchings		Correspondences Checked		Run Time (sec)
	Method 1	Method 2	Method 1	Method 2	
C1908	8.7e+36	7.9e+12	1.3e+62	NA	NA
alu2	3.6e+06	2.0e+00	2.9e+10	3.2e+01	0.2
alu4	8.7e+10	8.6e+03	3.5e+15	6.9e+04	232.4
cc	5.1e+19	1.4e+07	1.2e+38	1.5e+09	37675.5
f51m	4.0e+04	4.8e+01	1.6e+09	4.3e+02	0.1
pm1	2.1e+13	2.0e+05	1.3e+23	2.8e+06	273.4
sct	1.2e+17	4.0e+07	1.6e+29	6.0e+08	75647.4
t481	2.1e+13	2.3e+07	2.1e+13	2.3e+07	88354.5
z4ml	5.0e+03	5.0e+03	1.2e+05	2.0e+04	4.55

Table 5.2: **Experimental results.** The circuits included in this table are a subset of the LGSynth93 benchmark suite. The results listed for Method 1 are calculated for the Factorial Permutation approach (Section 5.2.2). The results presented for Method 2 are experimental results for a single vector signature implementation of the Semantic Equivalence Algorithm presented in Section 5.4.1.

The **alu4** circuit (a 4-bit ALU) is sufficiently complex to have fairly well-distributed vector signatures and thus is able to take advantage of vector signature information to recognize that only 8,640 of the greater than 87 billion possible input matchings can possibly produce a legal correspondence. The use of vector signatures has made this intractable comparison feasible. Furthermore, note that of the 3.5 x 10^{15} total correspondences (14!8!) possible, only 69,411 comparisons are necessary.

Obviously, circuits (such as C1908) exist for which a single vector signature does not adequately prune the matching space. A single vector signature is capable of reducing the number of input matchings for the 173 input LGSynth93 **pair** circuit from 173! to approximately 73!. While this reduction of search space is certainly significant, the suspects sets must be significantly reduced in size (by using additional input signatures) to permit semantic matching within a reasonable execution time.

Chapter 6

Reengineering methodology

As discussed in Section 1.1.1, reengineering consists of some form of reverse engineering, followed by possible alteration, and finally the more traditional forward engineering process. The rationale for the reengineering of legacy digital systems (Section 1.2.2) is to reduce the costs associated with the reimplementation of existing, tested devices in new fabrication technologies. Thus, the first step in the reengineering of legacy digital systems is the recovery of the original design to a level appropriate for reimplementation.

The RTL description of a device is technology independent and therefore abstract enough to allow for implementation in any fabrication technology (De Micheli, 1994). Furthermore, for many systems, formal verification techniques exist which can prove the equivalence of a synthesized implementation and the RTL description.

The difficulty lies in constructing a RTL description of an existing device which is provably correct. If such a design can be recovered and if existing formal verification techniques can prove the synthesis (forward engineering) process correct, then we will

have reengineered a new device which is provably equivalent to the legacy system. Ideally, the testing cycle for the reengineered device will be significantly shortened by providing a formal proof of correctness.

6.1 Design recovery methodology

This section presents a possible approach to design recovery which takes advantage of the SBDD and semantic pattern matching techniques discussed previously (Chapters 3, 4, and 5).

Design recovery methodology

Step 1: Partitioning. Using syntactic matching techniques, identify all sequential system components. Use these devices as cut points to divide the system into subcircuits of combinational logic.

Step 2: Represent available information. Represent the structural relationships within each cluster of combinational logic as an SBDD (Chapter 4). Create constraint characteristic functions (Definition 4.1) for any available information which provides information regarding the relationship of structures within the cluster.

Step 3: Refine specification. If the information represented introduces a conflict, then flag the relationships for external resolution (Section 4.2.2, Case 3). If functionality remains unspecified, then determine a minimal set of information which is sufficient to specify behavior. Determine this information through testing, external deduction, or other techniques (Section 4.2.2, Case 2). If the SBDD representing

the combinational subcircuit is completely specified (Section 4.2.2, Case 1), then synthesize a gate-level implementation of the blackbox functionality and complete the structural description. If the functionality of the combinational cluster cannot be completely specified, then provably correct recovery of the design is not possible.

Step 4: Match functional behavior. For each subcircuit of the cluster, attempt to determine block-level functionality (if any) using semantic matching techniques (Chapter 5). Every subcircuit must be considered since any subcircuit may implement the functionality of a block-level module (Doom et al., 1998). Recall that semantic matching techniques determine equivalence though matching of functionality rather than matching implementation. Hence, the introduction of deduced gate-level implementations of blackbox components (in Step 3) does not pose additional complications for this matching.

Step 5: Produce RTL description. Produce a description of the circuit in terms of sequential components (if any) identified in Step 1, block-level modules identified for each combinational cluster in Step 4, and remaining combinational logic. In many cases, this description should be sufficient for effective redesign.

6.2 Capabilities and limitations

This methodology is presented as a preliminary approach to the design recovery of digital systems. The SBDD and semantic matching techniques used in this approach are currently only applicable to combinational circuits; therefore it is necessary that we partition the system (Step 1) into combinational subcircuits prior to the applica-

tion of these techniques. This methodology is particularly appropriate to the design recovery of obsolete legacy systems, as these systems are often composed of non-optimized, primarily combinational devices.

Furthermore, this methodology is capable of deducing the functional role of partially specified combinational components. However, the SBDD-based techniques discussed in Chapters 3 and 4 are not currently applicable to sequential devices and are therefore not yet applicable in the determination of the specifications of partially specified sequential components. Thus, it is necessary that any blackbox components be composed of fully combinational devices in order to utilize this approach in sequential systems for which only partial information exists.

This preliminary approach is primarily targeted towards non-optimized systems. The identification of library entities in modern systems are complicated by design optimizations which may obfuscate the function of the entity implementation.

Non-reversible optimizations

During the design and synthesis process, logic functions may be modeled from available block-level or gate-level components that "almost" fit the necessary function. Common techniques used to create implementations for functions which "almost" fit an existing function include bridging inputs, applying stuck-at values, and ignoring outputs (Mailhot and De Micheli, 1993).

When two (or more) inputs to a library cell are connected to the same input line, such cell inputs are *bridged.* When a cell input is tied to ground (power), the input is *stuck-at-0* (*stuck-at-1*). Furthermore, some outputs of the library entity may not

be used. Designs that incorporate these optimizations may be difficult to identify in their corresponding gate-level netlists. The actual components used depend on the specific modules available and the cost metrics associated with the binding processes.

When a block-level component is designed with bridged or stuck-at inputs, its gate-level implementation may be optimized to take advantage of this fact during synthesis. In cases involving stuck-at or bridged inputs, the number of observable inputs to a cluster of gates may actually differ from the number of inputs to the block-level module which it represents. Likewise, a design which contains a block-level module with an output which is ignored will generally not represent that output at the gate-level, causing the number of observable outputs to a gate cluster to differ from the number of outputs of the block-level which it represents. In these cases, existing semantic matching techniques are not sufficient, alone, to identify the block-level device which corresponds to such a subcircuit. Design recovery in this circumstance remains an open problem.

Reversible optimizations

Another common design optimization is the reduction of implementation area by causing the intermediate functionality of several distinct library units to occur in a single, shared cluster. During logic synthesis and technology mapping, each block-level module is reduced to primitive logic functions which must be mapped to available library cells and appropriately interconnected to provide the necessary high-level functionality of the block-level component. In situations where two or more block-level components share a logic function primitive, the gate-level implementation may be

optimized so that the same physical implementation is used in the logical function of both block-level modules (De Micheli, 1994).

Since both modules sharing a portion of combinational logic can be functionally identified through semantic comparison to the appropriate combinational cluster, optimizations of this sort are reversible. Such optimizations must be kept in mind, however, when designing a subgraph enumeration algorithm (Section 5.1). It is necessary in such cases to allow for the possibility that some cluster outputs are not outputs of a block-level device but are, instead, used in the implementation of another device. Such an approach seriously complicates the subgraph enumeration process, but is necessary to allow identification of block-level modules which have undergone such optimization.

Don't Care Sets

Consider a circuit with primary inputs $\mathbf{x}$, primary outputs $\mathbf{z}$, and the vector of functions $\mathcal{H}(\mathbf{x}) = \mathbf{z}$ (as defined in Equation 5.1), which determines the relationship between them. Also consider some cluster within this circuit with inputs $\mathbf{i}$, outputs $\mathbf{o}$, and vector of functions $\mathcal{F}(\mathbf{i}) = \mathbf{o}$, which similarly determines the behavior of the cluster circuit. In a completely specified circuit, it is possible to determine the vector of functions $\mathcal{P}(\mathbf{x}) = \mathbf{i}$ (which determines the cluster inputs for any given primary input set) and the function $\mathcal{Q}(\mathbf{x}, \mathbf{o}) = \mathbf{z}$ (which determines the value of the primary outputs based upon the value of the cluster outputs and the behavior of the rest of the circuit). These relationships fully describe the environment around the multi-output cluster.

The *input controllability don't care set* (CDC) for the cluster includes all input

conditions that are never produced by the environment (Benini and De Micheli, 1997). Thus the CDC is defined as follows:

$$CDC = \{\mathbf{i} | \mathbf{i} \text{ is not in } Range(\mathcal{P}(\mathbf{x}))\}. \tag{6.1}$$

The *output observability don't care set* (ODC) for each output of the cluster denotes all input patterns that produce situations in which the output of the cluster is not observed by the environment (Benini and De Micheli, 1997). Effectively, the ODC set contains all cluster inputs for which the values of the primary outputs do not depend upon the output(s) of the cluster. In mathematical terms:

$$ODC = \{\mathbf{i} | \forall \mathbf{x} \text{ such that } \mathcal{P}(\mathbf{x}) = \mathbf{i}, \forall \mathbf{o} \in Range(\mathcal{F}), \mathcal{Q}(\mathbf{x}, \mathbf{o}) = \mathcal{H}(\mathbf{x})\}. \tag{6.2}$$

These don't care conditions produce degrees of freedom available within the cluster function. Functions within these degrees of freedom will produce behavior which is indistinguishable with regards to the environment. Therefore, during synthesis, a function "effectively equivalent" but not necessarily "identically equivalent" to the logic function specified by the cluster may be chosen to implement it. That is, some vector function $\mathcal{F}'$ can be used such that:

$$\forall \mathbf{i} \in \{Range(\mathcal{P}) - \mathbf{CDC} - \mathbf{ODC}\}, \mathcal{F}(\mathbf{i}) = \mathcal{F}'(\mathbf{i}). \tag{6.3}$$

The actual function implemented will be one of the functions that obeys these condi-

tions and meets some design criteria such as cost. These kinds of *don't care* optimizations are common in sophisticated synthesis algorithms as well as in hand-optimized designs.

In order to recover design in systems which exploit *don't care* sets during optimization, we must be able to determine the *effective* high-level function implemented by a cluster. The preliminary results presented in this dissertation do not consider problems that contain *don't care* optimizations. We hypothesize, however, that this task can be effectively carried out by determining the effective functionality of a subcircuit by treating it as a blackbox, and identifying all blackbox input assignments for which it is not a *don't care* (as described in Chapter 4).

With this information, it is possible to define a new cluster function which has the behavior of the original cluster function under all input assignments which do not correspond to *don't care* assignments, and the value 0 otherwise. Intuitively, we apply a mask which sets the value of the function to 0 under any input conditions which have been identified as *don't care* conditions. Likewise, it is possible to define a new pattern function which has the behavior of the original pattern function under all inputs which do not correspond to *don't care* assignments, and the value 0 otherwise. The application of this mask effectively removes all degrees of freedom, and allows the semantic matching techniques described in Chapter 5 to determine if the effective role of the cluster is that of the module represented by the pattern function.

6.3 Formal correctness

It is of vital importance that the recovered design be functionally correct. That is, the recovered design need not necessarily be identical to the original design, but it should represent the same functionality (assuming that there were no errors in the original implementation).

Obviously, in a completely specified implementation, existing formal verification techniques can be applied to prove equivalence between the existing implementation and the recovered design. Designs recovered using our proposed methodology, however, are provably correct by construction. Intuitively, the subgraph enumeration and semantic matching techniques replace portions of the implementation-level design with equivalent block-level components. Since the functionality performed by the block-level components is exactly equivalent to that performed by the gates which they represent, the recovered design is correct. If semantic matching is performed on optimized circuits as discussed in the previous section, the block-level components are not necessarily exactly equivalent to the gate which they represent but are provably equivalent in the overall context of the function.

In situations for which only partial system knowledge is available, the deduction of effective functionality is also provable. SBDDs are used to represent characteristic functions of partial relationships within a system. Our approach only recovers effective functionality of systems represented by fully specified SBDDs. If the SBDD representing the system is not fully specified, the design is not recovered.

However, if the SBDD representation is fully specified, then the specification of

any blackbox structures is the collection of the unique input/output relations which satisfy the constraint characteristic functions. Furthermore, it is obvious that the assignment of output values is alway "correct" for input conditions under which the value of the blackbox has been determined as a *don't care* in terms of overall system function. Therefore, a formal proof of equivalence can be constructed for any design recovered by this approach.

6.4 Complexity issues

The problem of representing and proving equivalence of arbitrary Boolean functions is inherently intractable (Jain et al., 1997). Although the BDD representation has proven to be an effective representation for the functionality of many digital systems, the usefulness of this representation is dependent upon a number of factors. Most important among these factors are the number of variables represented, the BDD variable order, and the particular BDD variant (BMD, ZDD, et al. (Bryant, 1995)) which is used as the underling representation.

The SBDD approach to the deduction of functionality is dependent upon the the representation of the characteristic function that represents known relationships. Section 3.4.2 presented the worst-case SBDD size and a technique for size reduction. In some circumstances, of course, the functions represented by an SBDD can reach these worst-case sizes. This approach uses SBDDs only to represent the combinational portions of the digital system. Therefore the size of the SBDD is based upon the size of the partitions (Section 6.1, Step 6), not necessarily on the size of the overall circuit.

We are therefore confident that SBDD-based techniques are applicable to systems of the complexity currently undergoing reengineering.

On-going research efforts to extend the capabilities of the BDD representation are active in the design automation community. Since the SBDD is fundamentally a BDD which represents a Boolean function described in Chapters 3 and 4, this on-going research can be utilized to increase the maximum size of combinational subcircuits which may be represented by this methodology. In particular, research into parallel BDD implementations will have significant impact upon the size of combinational partitions which can be effectively represented and solved.

As our goal for this dissertation is to provide a preliminary approach to the problem of redesign for digital systems, as opposed to implementing a commercial redesign system, we are unable to include results which concern the size of problems which can be solved using this approach. It is our hope, however, that opportunities for such research will present themselves in the future.

Chapter 7

Conclusion and future directions

We have shown that considerable interest exists in the design automation community regarding formal techniques for the remanufacture and reengineering of obsolete digital systems. Furthermore, we have discussed substantial hindrances to implementing successful reengineering. Systems are often a blend of digital, analog, and software components. A variety of sources of system data might be available, such as the physical hardware, software source code, test program sets, manufacturing artwork, or paper documentation. Even with all of these potential sources of information, reengineering is complicated by the fact that some of the system data might be contradictory, incomplete, or out-of-date. It is our belief that formal design recovery in the reengineering process should be viewed as a process for providing error-free retroactive documentation of an existing digital system.

We have presented an introductory overview of the reengineering process, of the design process, and of the current role of formal verification in digital design. Furthermore, we have discussed the need for formal design recovery in existing critical

legacy systems. Most importantly, we have presented new techniques that facilitate the formal recovery of high-level design information from available low-level, and possibly incomplete, descriptions of digital systems. These techniques take advantage of the proven efficiency of BDDs to represent partially specified logic and to represent the Boolean relationships which determine the behavior of the represented device. Using these techniques, we have shown that we can identify block-level modules, detect conflicts, and deduce unspecified functional behavior from structural context and available additional information.

7.1 Semantic matching

The goal of reengineering digital systems (Section 1.2.2) is to reduce the costs associated with the remanufacture of existing, tested devices. In order for a device to be implemented in a new technology, it is first necessary to recover a RTL design of the system. The only source of trusted system information in some critical systems is the existing, fully tested, physical hardware. Other sources of information, if available, cannot necessarily be trusted to be correct. Furthermore, state of the art reverse engineering techniques are not always able to provide a complete gate-level description of an existing device. In some cases, the functionality of portions of the device are unknown and must be treated as blackboxes.

Previous approaches to this problem have attempted to transform gate-level descriptions of a device into RTL descriptions of a device through syntactic (structural) matching techniques. Syntactic matching, however, has limited application since

high-level components have many valid implementations. Design optimizations for area and power, for example, may obfuscate implementations, causing syntactic techniques to fail.

The function of an arbitrary combinational subcircuit is semantically (functionally) equivalent to the function of a high-level component if input and output correspondence exist under which the functions are equivalent. Previous approaches to this problem could not utilize semantic techniques since they required factorial exploration of the input and output correspondence search space.

We have met our goal of developing a method to determine semantic equivalence between a subcircuit and a high-level component in a tractable number of comparisons. We introduced the concept of using input signature functions to partition device inputs into equivalence classes called suspect sets. In particular, we introduced a new input signature function (the vector signature function) which takes advantage of the multiple-output nature of high-level modules and has proven to be particularly efficient in partitioning device inputs. Since input correspondences need only be considered between members of corresponding suspect sets, we significantly reduced the complexity of this problem, making it tractable for many common modules. Lastly, we have presented preliminary results which demonstrate the effectiveness of the technique using a single vector signature filter.

Future directions in semantic matching relate primarily to the introduction of additional filters to decrease the run time and increase the capabilities of the program. The vector signature alone is not an effective filter for several of the circuits tested. Additional function filters, such as those described in Section 5.2.5, are necessary if

this technique is going to be used effectively. The effectiveness and the costs of each filter should be explored and the identification of intractable problems, if any, should be considered.

Some functions are inherently difficult to describe and match when this technique is used. The multiplier and multiplexer are two such functions. The multiplier is quite sensitive to filtering, and the number of comparisons necessary is relatively small. Each such comparison, however, is exceptionally time consuming. Multipliers are well known to produce exponential graphs when represented as BDDs. Creating the BDD that represents the function of the multiplier under some variable ordering may be prohibitively time consuming.

The multiplexer function, on the other hand, is almost completely insensitive to the vector filter function. A multiplexer consists of n control inputs, 2^n data inputs, and a single output whose value is equal to that of the input selected by the control inputs. Although the n control inputs may be identified by the vector signature, all but two of the data inputs (the $\vec{1}$ and $\vec{0}$ lines) fall into the same equivalence class (since their behavior is never selected by the control inputs). If n is greater than four, there would be at least $2^4 - 2 = 16 - 2 = 14$ input variables in the same vector class, requiring at least 14! comparisons, which is intractable. The equivalence algorithm can "flag" such clusters as having an intractable number of comparisons, but some other method must be later used to consider these cases.

A technique for canonically ordering variables based on the recursive sorting of truth-tables by row and column sums is presented in (Wu et al., 1994). If such a technique can be implemented efficiently, it will be completely unnecessary to consider

searching the factorial matching space to determine PP-equivalence (Definition 5.1). The canonical ordering for the cluster and the canonical order for the entity must indicate an appropriate matching when such a matching exists. A tool based on this mechanism should be developed and tested for efficiency as well as maximum problem size. Although this canonicalization is of exponential complexity, this technique is quite promising.

This canonical ordering technique could be quite useful in performing (exact) equivalence matching, because we would no longer need to test equivalence under all input correspondences. We would merely need to determine the "unique" input order of the function before the test. This technique would be more efficient than current techniques for many functions, particularly those with a large number of inputs and a small number of outputs for which signature-based techniques may prove intractable.

To the best of our knowledge, no techniques for "canonicalizing" the variables in a BDD has been proposed. Perhaps a metric by which a BDD could be recursively ordered can be determined. If so, this would be a significant contribution to the field.

7.2 Representation of available information

Design recovery for digital systems is challenging as some information about the circuit may be unavailable. Therefore, it is necessary to be able to recognize the functionality of any set of circuit components from available system information. Since this information may not have been fully tested, conflicts between available information must be identified so that they may be resolved externally. Although complete

deduction of functionality may be impossible in an incompletely specified implementation, available information allows the deduction of complete system specification in many cases.

The traditional BDD representation of circuit functionality represents the external functionality of the circuit. That is, circuits are generally represented as a set of functions representing primary outputs in terms of primary inputs. Representations of this sort are equivalent to behavioral-level descriptions of the device. Since BDDs can be used to represent any Boolean function, they are also capable of representing the characteristic equations of circuit structures. In Chapter 3, we defined the structure function to be a characteristic function which represents the behavior of internal circuit structures as well as the circuit's primary outputs. This representation corresponds to a structural-level representation.

In Chapter 4, we described a more "relaxed" characteristic function in which any variable assignment which leads to a 0-terminal is illegal, but in which a variable assignment leading to a 1-terminal is not guaranteed to be legal. This new characteristic function is capable of representing partially specified information. We refer to a BDD which represents this "relaxed" characteristic function as an SBDD, and it is this interpretation of our new function which is the basis for our approach.

It should be noted that SBDDs may contain decision variables representing more than one output. Standard BDD representations of a circuit require one BDD for each output. Furthermore, BDDs contain no decision variables representing internal circuit structures. An SBDD can represent an entire circuit, with all of its variable relationships, in one structure. It is exactly this ability to represent relationships

throughout an entire circuit which makes SBDDs a powerful tool for reengineering.

Recovery of unspecified functionality

SBDDs allow for the representation of partial Boolean functions involving variables represented as a decision nodes. Furthermore, we have shown how characteristic functions representing information available at various design levels can be created and included to specify new relationships within the SBDD or to identify conflict. New relationships between any of the variables represented can be introduced by limiting all 1-paths which contradict the relationship. Any Boolean relationship between variables represented in the SBDD can be included, regardless of the level of design from which the information comes. The satisfying set of an SBDD represents the subset of the Boolean space determined by net variables which encapsulates the system's behavior.

After introducing new knowledge, the circuit may then become completely specified, or it may still contain unknowns, (nodes labeled as representing blackboxes, for which either a 0 or 1 output may be legal). For a completely specified circuit, we know for which inputs the value of the blackbox has an effect upon the output and what the value of the blackbox is under these inputs. For all other inputs (for which the blackbox output has no effect upon the circuit output), the value of the blackbox does not need to be known. These values are *don't cares* for the blackbox functionality in the context of the overall circuit function.

These techniques allow the identification of the three possible results (Section 4.2.2) which occur after the inclusion of all available information. In cases which the

overall circuit functionality is completely specifiable, we have shown how to apply efficient graphical techniques to identify the specification of blackbox structures to within *don't care* conditions. In cases in which the overall circuit functionality is not completely specified, we have shown how to create a minimal list of input/output relationships which, if determined, allow complete deduction of the specification. Finally, we have shown how conflicting information can be identified by graphically identifying situations in which no legal 1-path exists for any primary input assignment.

Initial work in this area is promising. "Complete knowledge" representations of partially specified combinational implementations have been produced which encode the necessary functionality of the circuit hierarchy. Furthermore, we have been able to apply information from another level of description (test vectors) and automate a deductive task. To the best of our knowledge, this is among the first such work which makes use of test vector information to aid in the design recovery of a digital component. Parallel research has proposed an informal approach towards recovering the design of digital VLSI circuits with incomplete implementation information using methods such as exhaustive simulation (Wey and Khalil, 1998; Khalil, 1998). Our approach allows for the mathematical simplification of the problem search space as opposed to relying on partitioning heuristics of the sort utilized in this informal approach. Furthermore, our approach allows the utilization of relationships between any set of net variables to be used in recovering the design rather than being limited to relationships between the primary inputs and outputs. Both approaches have merit and need to be studied further.

In the future, we hope to apply this approach to more difficult problems. In par-

ticular, we wish to incorporate constraints representing information from other levels of design. Additionally, the feasibility of redesign for systems in which large numbers of blackboxes produce the possibility of multiple legal implementations should be explored.

There are several functional library packages for BDDs in common use. The SBDD interpretation will remain a ROBDD in function, hence all theorems which apply to ROBDDs will apply to SBDDs as well. SBDD implementations can take advantage of existing tool-sets and will be able to take advantage of ongoing research in distributed algorithms (Ranjan et al., 1996) for BDD implementations or other techniques which may increase the utility of BDDs in the future.

7.3 Reengineering methodology

Finally, we have suggested a reengineering approach which make uses of both semantic matching and SBDD-based techniques to recover the design of digital systems to the register-transfer level. Since these techniques use logical equivalence and deduction, the resulting design is provably correct.

We have presented a formal approach to recovering the design of components in a partially specified combinational design. Unlike traditional BDD representations of circuit function, our approach is capable of *representing partial specifications* of the circuit's external or internal functionality. This approach uses characteristic functions to represent relevant Boolean relationships among net variables, where the relationships can come from any level of the design process.

We have presented techniques which allow for the deduction of the functionality of unspecified circuit components. When complete deduction is not possible, our representation allows for the enumeration of unknown relationships which may allow complete recovery if a means for acquiring these relationships exists.

A final salient feature of this approach is its ability to detect conflicting information about a design. Because the SBDD represents legal assignments of variable values, information incorporated into an SBDD that results in conflicting legal assignments can be easily detected. This will allow the user of the system to examine the sources of the conflicting information and determine what course of action should be taken.

Our approach focuses primarily on the identification of block-level combinational devices. We take advantage of the fact that the identification of latches in a sequential circuit is a relatively simple problem to allow our approach to be used on simple sequential devices (Section 6.1). However, it should be noted that our approach is not currently capable of identifying block-level sequential devices (such as a shift register). The extension of our work to include the identification of such devices would greatly extend the utility of this approach as a tool for recovering the design to a level more readily understandable by humans (although this identification is not necessary for simple reimplementation).

It must be kept in mind that the focus of our proposed reengineering approach is to deal with the effective reimplementation of legacy digital systems. An important future direction of research involves determining additional techniques which will allow similar reengineering on the complex and highly-optimized devices pro-

duced using modern synthesis techniques. Conversely, such research would also prove invaluable in determining effective techniques to protect intellectual property from "hostile" reverse engineering through intentional obfuscation.

Clearly, the initial approach presented in this dissertation forms a basis for pursuing such future research. We hope that our contributions to this topic provide insights which prove valuable to the reengineering community at large. Although a comprehensive design recovery methodology has yet to be developed, we hope that our work is convincing proof of the feasibility of design recovery for digital hardware, and that it motivates additional research into this open problem.

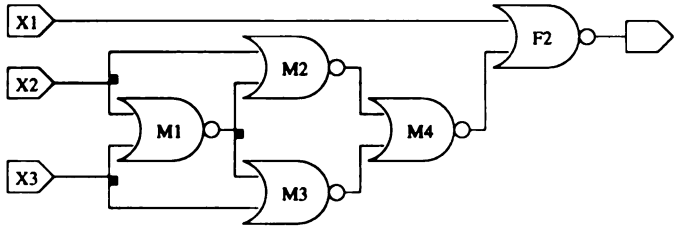
APPENDICES

Appendix A

Complete RFPS solution for `simplecircuit`

This appendix contains a detailed walk-through of the application of the SBDD-based design recovery methodology to an RFPS problem (Section 4.2.1) for the circuit **`simplecircuit`**. This circuit (Figure A.1) is referenced extensively throughout Chapters 3 and 4. Throughout the dissertation, graphs representing SBDDs were represented with multiple terminals and repetition of nodes representing decision variables for primary outputs so as to be more intuitive. In this appendix, SBDDs are presented as they are actually represented in the system (i.e., with subgraph isomorphism sharing).

Figures A.2 and A.3 present the VHDL and BLIF code (respectively) for **`simplecircuit`**. Figure A.4 presents the set of ATPG test vectors generated for this circuit the SIS synthesis package (Sentovich et al., 1992). The information presented in these figures represents a subset of the design information commonly associated



(a)

$$F_2 = \overline{X_1}(\overline{X_2}X_3 + X_2\overline{X_3})$$

X_1	X_2	X_3	F_2
1	d	d	0
0	1	1	0
0	1	0	1
0	0	1	1
0	0	0	0

(b)

Figure A.1: **Schematic and functional truth-table.**

with a digital device.

In the RFPS problem for `simplecircuit` presented in Chapter 4, the only design information is a partial schematic (Figures A.5) in BLIF format (Figure A.6) and the circuit's set of ATPG test vectors (Figure A.4). The use of the algorithm presented in Section 4.3.1 to recover the specifications of this partial design will now be shown.

Initially, the SBDD representing the system's structure function is initialized to logical 1 (the 1-terminal). For each structure in the system (as described in the provided BLIF file) a constraint SBDD is created which represents the characteristic function of the component. These constraint SBDDs are composed with the SBDD representing the system's structure function. After all such SBDDs are composed, the resultant SBDD represents the sum of the separately available information.

Three structures are identified in the BLIF file for this RFPS problem. Figure A.7 presents the SBDD representing the functional constraint imposed by the first structure appearing in the BLIF file (the gate $M1$). The second structure in the BLIF file is that of the unknown structure BB. Including the SBDD representing the

```
ENTITY simplecircuit IS
  PORT( X1,X2,X3: in bit; F2: out bit );
END simplecircuit;

ARCHITECTURE behavioral OF simplecircuit IS
BEGIN
  F2 <= (not X1) and (((not X2) and X3) or (X2 and (not X3))) after 10 ns;
END behavioral;

ARCHITECTURE structural OF simplecircuit IS
  SIGNAL M1, M2, M3, M4 : bit ;
  FOR ALL : nor2 USE ENTITY trace.nor2(behav);
  FOR ALL : probe USE ENTITY trace.probe(behav);
  BEGIN
    gate0 : nor2 PORT MAP ( O => M1, a => X2, b => X3 );
    gate1 : nor2 PORT MAP ( O => M2, a => X2, b => M1 );
    gate2 : nor2 PORT MAP ( O => M3, a => M1, b => X3 );
    gate3 : nor2 PORT MAP ( O => M4, a => M2, b => M3 );
    gate4 : nor2 PORT MAP ( O => F2, a => X1, b => M4 );
    output_F2 : probe;
  GENERIC MAP( "F2", "sim_results/F2" );
  PORT MAP( F2 );
END structural;
```

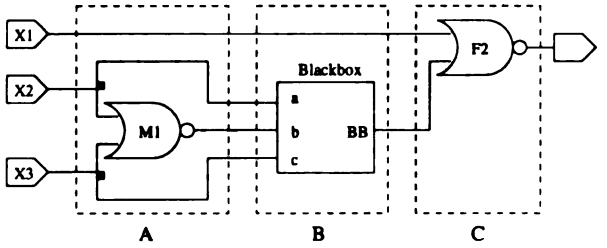
Figure A.2: **Behavioral and structural VHDL code for `simplecircuit`.** The behavioral description corresponds to the circuit specification, which is efficiently represented by a BDD. The structural description corresponds to the circuit implementation, which the SBDD efficiently represents.

```
# file name: simplecircuit.blif
.model simplecircuit
.inputs X1 X2 X3
.outputs F2
.gate nor2 a=X2 b=X3 O=M1
.gate nor2 a=X2 b=M1 O=M2
.gate nor2 a=M1 b=X3 O=M3
.gate nor2 a=M2 b=M3 O=M4
.gate nor2 a=X1 b=M4 O=F2
.end
```

Figure A.3: **BLIF code for `simplecircuit`.**

```
ATPG test sequences for simplecircuit
inputs:
X1 X2 X3
outputs:
F2
0 0 0    0
0 0 1    1
0 1 1    0
0 1 0    1
1 1 0    0
```

Figure A.4: **ATPG vectors for `simplecircuit`.**



(a)

$$F_2 = \overline{X_1} \cdot \overline{BB(X_2, M_1, X_3)}$$

X_1	X_2	X_3	$M1$	BB	F_2
1	d	d	d	d	0
0	1	1	0	u	$\overline{BB(1,0,1)}$
0	1	0	0	u	$\overline{BB(1,0,0)}$
0	0	1	0	u	$\overline{BB(0,0,1)}$
0	0	0	1	u	$\overline{BB(0,1,0)}$

(b)

Figure A.5: **A partial specification of `simplecircuit`.** (a) Circuit with unspecified subcircuit: Area A represents the known input logic, area B represents the unknown blackbox subcircuit, and area C represents the known output logic. (b) The symbolic solution space to this circuit: The blackbox representing M_4 is denoted symbolically as BB.

```
# file name: partial_simplecircuit.blif
.model partial_simplecircuit
.inputs X1 X2 X3
.outputs F2
.gate nor2 a=X2 b=X3 O=M1
.gate bb3 a=M1 b=X2 c=X3 O=BB
.gate nor2 a=BB b=X1 O=F2
.end
```

Figure A.6: **BLIF code for `partial simplecircuit`.**

unknown functionality of BB introduces no useful new relationships (Figure A.8). After the introduction of the final structure, the SBDD in Figure A.9 is produced. This SBDD represents the useful logical relationships deducible from the information available in the BLIF file. Most importantly, this SBDD allows the identification of those input conditions under which the value of the blackbox BB affects the system output $F2$ and under which input conditions its output is a *don't care* (Figure A.9(c)).

Once all information available in the partial schematic has been introduced, the algorithm attempts to introduce any additional system information available. In the RFPS problem, a set of test vectors is provided. For each test vector, a SBDD representing the relationship is constructed (as discussed in section 4.2.1) and applied to the system SBDD.

Figure A.10 shows the introduction of the relationship represented by the first ATPG test vector to the system SBDD. Note that after the introduction of this relationship, the value of BB has been determined under the blackbox input conditions $X_2 = 0, X_3 = 0, M_1 = 1$. This relationship has been deduced from the fact that the output F_2 is sensitized to the value of BB under the primary input conditions specified by the first test vector. Therefore, since the test vector specifies the necessary value of F_2 under those input conditions, a partial specification of BB is deduced. Figures A.11 through A.14 present similar deductions. Note that the final test vector does not introduce any new information about the system, and thus the SBDDs for steps 7 and 8 are identical.

After the addition of all available information, the SBDD is examined and the specification of BB is extracted. In this example, the effective functionality of

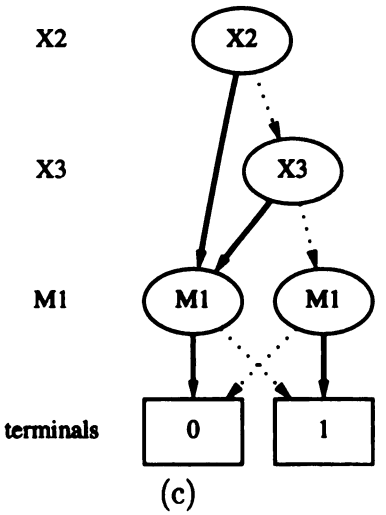
BB is specified (to within *don't care* conditions) and therefore, the functionality of **simplecircuit** is fully specified.

```
# file name: partial_simplecircuit.blif
.model partial_simplecircuit
.inputs X1 X2 X3
.outputs F2
.gate nor2 a=X2 b=X3 O=M1
.gate bb3 a=M1 b=X2 c=X3 O=BB
.gate nor2 a=BB b=X1 O=F2
.end
```

(a)

X_1	X_2	X_3	M_1	BB	F_2	$\mathcal{X}^S$
d	1	*d*	0	*d*	*d*	1
d	0	1	0	*d*	*d*	1
d	0	0	1	*d*	*d*	1
			otherwise			0

(b)



(c)

$$\mathcal{X}^S = (\overline{X_2 \vee X_3} \leftrightarrow M_1)$$

(d)

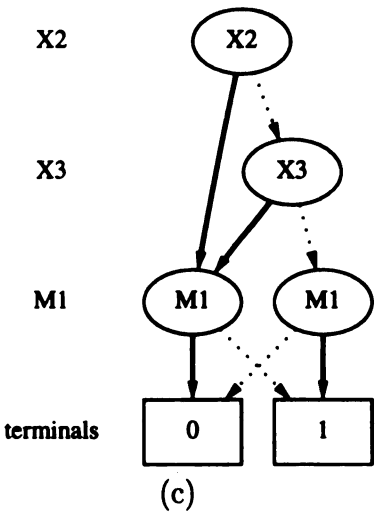
Figure A.7: **Solving the RFPS solution for `simplecircuit`: #1.** (a) The BLIF commands for the **`simplecircuit`** RFPS Problem (Structures whose relationships have been represented in the SBDD are shown in italics. The structure whose relationship is newly represented in the SBDD is shown in bold italics.); (b) A truth-table for the structure function represented by (c); (c) The BDD representing the structure function of **`simplecircuit`** after including the relationships specified in (a); (d) The symbolic logic expression for the structure function.

```
# file name: partial_simplecircuit.blif
.model partial_simplecircuit
.inputs X1 X2 X3
.outputs F2
.gate nor2 a=X2 b=X3 O=M1
.gate bb3 a=M1 b=X2 c=X3 O=BB
.gate nor2 a=BB b=X1 O=F2
.end
```

(a)

X_1	X_2	X_3	M_1	BB	F_2	$\mathcal{X}^S$
d	1	d	0	d	d	1
d	0	1	0	d	d	1
d	0	0	1	d	d	1
			otherwise			0

(b)



(c)

$$\mathcal{X}^S = (\overline{X_2 \vee X_3} \leftrightarrow M_1)$$

(d)

Figure A.8: **Solving the RFPS solution for `simplecircuit`: #2.** (a) The BLIF commands for the **`simplecircuit`** RFPS Problem (Structures whose relationships have been represented in the SBDD are shown in italics. The structure whose relationship is newly represented in the SBDD is shown in bold italics.); (b) A truth-table for the structure function represented by (c); (c) The BDD representing the structure function of **`simplecircuit`** after including the relationships specified in (a); (d) The symbolic logic expression for the structure function.

```
# file name: partial_simplecircuit.blif
.model partial_simplecircuit
.inputs X1 X2 X3
.outputs F2
.gate nor2 a=X2 b=X3 O=M1
.gate bb3 a=M1 b=X2 c=X3 O=BB
.gate nor2 a=BB b=X1 O=F2
.end
```

(a)

X_1	X_2	X_3	M_1	BB	F_2	χ^S
1	1	*d*	0	*d*	0	1
1	0	1	0	*d*	0	1
1	0	0	1	*d*	0	1
0	1	*d*	0	1	0	1
0	1	*d*	0	0	1	1
0	0	1	0	1	0	1
0	0	1	0	0	1	1
0	0	0	1	1	0	1
0	0	0	1	0	1	1
otherwise						0

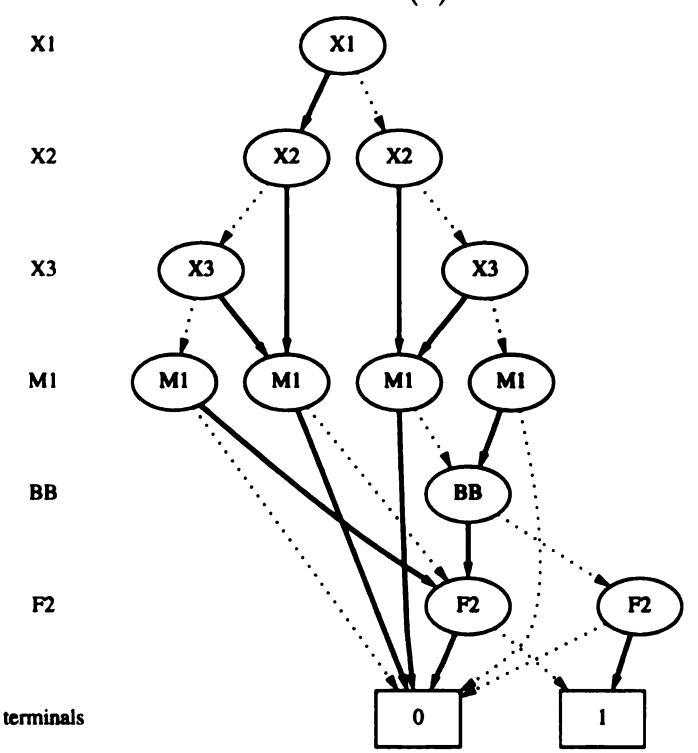
(b)

X_2	X_3	M_1	BB
1	*d*	0	*u*
0	1	0	*u*
0	0	1	*u*
otherwise			*d*

(c)

$$\chi^S = (\overline{X_2 \vee X_3} \leftrightarrow M_1) \wedge (\overline{BB \vee M_1} \leftrightarrow F_2)$$

(e)



(d)

Figure A.9: **Solving the RFPS solution for `simplecircuit`: #3.** (a) The BLIF commands for the `simplecircuit` RFPS Problem (Structures whose relationships have been represented in the SBDD are shown in italics. The structure whose relationship is newly represented in the SBDD is shown in bold italics.); (b) A truth-table for the structure function represented by (d); (c) The specification of BB encoded in the SBDD; (d) The BDD representing the structure function of `simplecircuit` after including the relationships specified in (a); (e) The symbolic logic expression for the structure function.

X_1	X_2	X_3	F_2
0	**0**	**0**	**0**
0	0	1	1
0	1	1	0
0	1	0	1
1	1	0	0

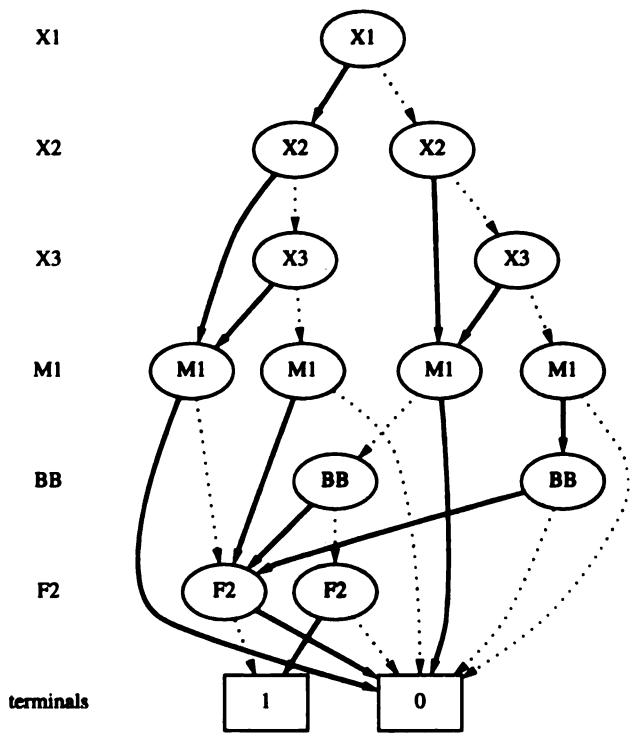
(a)

X_1	X_2	X_3	M_1	BB	F_2	$\mathcal{X}^S$
1	1	*d*	0	*d*	0	1
1	0	1	0	*d*	0	1
1	0	0	1	*d*	0	1
0	1	*d*	0	1	0	1
0	1	*d*	0	0	1	1
0	0	1	0	1	0	1
0	0	1	0	0	1	1
0	0	0	1	1	0	1
otherwise						0

(b)

X_2	X_3	M_1	BB
1	*d*	0	*u*
0	1	0	*u*
0	0	1	1
otherwise			*d*

(c)



(d)

$$\mathcal{X}^S = (\overline{X_2 \vee X_3} \leftrightarrow M_1) \wedge (\overline{BB \vee M_1} \leftrightarrow F_2) \wedge (\overline{X_1} \wedge \overline{X_2} \wedge \overline{X_3} \rightarrow \overline{F_2})$$

(e)

Figure A.10: **Solving the RFPS solution for `simplecircuit`: #4.** (a) The ATPG test vectors for the `simplecircuit` RFPS Problem (Vectors whose relationships have been represented in the SBDD are shown in italics. The vector whose relationship is newly represented in the SBDD is shown in bold italics.); (b) A truth-table for the structure function represented by (d); (c) The specification of BB encoded in the SBDD; (d) The BDD representing the structure function of `simplecircuit` after including the relationships specified in (a); (e) The symbolic logic expression for the structure function.

X_1	X_2	X_3	F_2
0	*0*	*0*	*0*
0	**0**	**1**	**1**
0	1	1	0
0	1	0	1
1	1	0	0

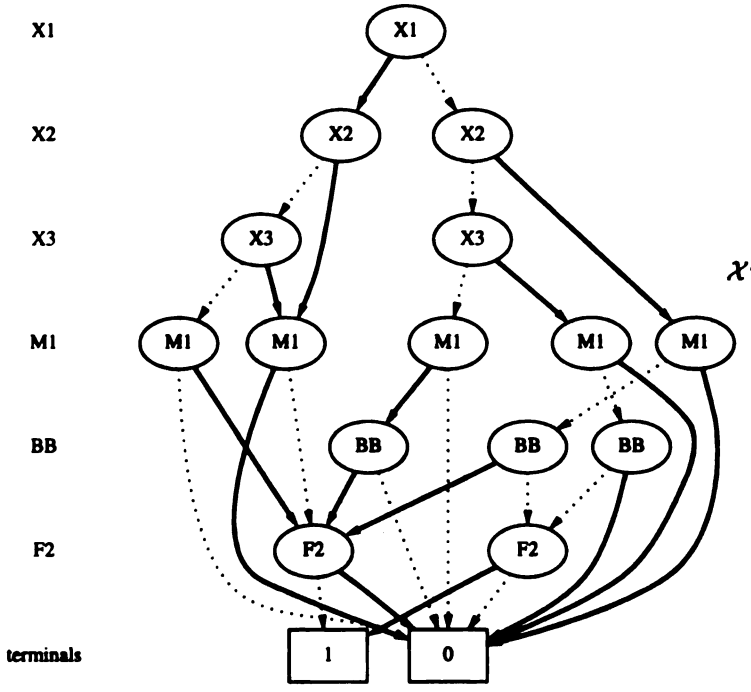
(a)

X_1	X_2	X_3	M_1	BB	F_2	$\mathcal{X}^S$
1	1	*d*	0	*d*	0	1
1	0	1	0	*d*	0	1
1	0	0	1	*d*	0	1
0	1	*d*	0	1	0	1
0	1	*d*	0	0	1	1
0	0	1	0	0	1	1
0	0	0	1	1	0	1
otherwise						0

(b)

X_2	X_3	M_1	BB
1	*d*	0	*u*
0	1	0	0
0	0	1	1
otherwise			*d*

(c)



(d)

$$\mathcal{X}^S = (\overline{X_2 \vee X_3} \leftrightarrow M_1) \wedge (\overline{BB \vee M_1} \leftrightarrow F_2) \wedge (\overline{X_1} \wedge \overline{X_2} \wedge \overline{X_3} \rightarrow \overline{F_2}) \wedge (\overline{X_1} \wedge \overline{X_2} \wedge X_3 \rightarrow F_2)$$

(e)

Figure A.11: **Solving the RFPS solution for `simplecircuit`: #5.** (a) The ATPG test vectors for the **`simplecircuit`** RFPS Problem (Vectors whose relationships have been represented in the SBDD are shown in italics. The vector whose relationship is newly represented in the SBDD is shown in bold italics.); (b) A truth-table for the structure function represented by (d); (c) The specification of BB encoded in the SBDD; (d) The BDD representing the structure function of **`simplecircuit`** after including the relationships specified in (a); (e) The symbolic logic expression for the structure function.

X_1	X_2	X_3	F_2
0	*0*	*0*	*0*
0	*0*	*1*	*1*
0	**1**	**1**	**0**
0	1	0	1
1	1	0	0

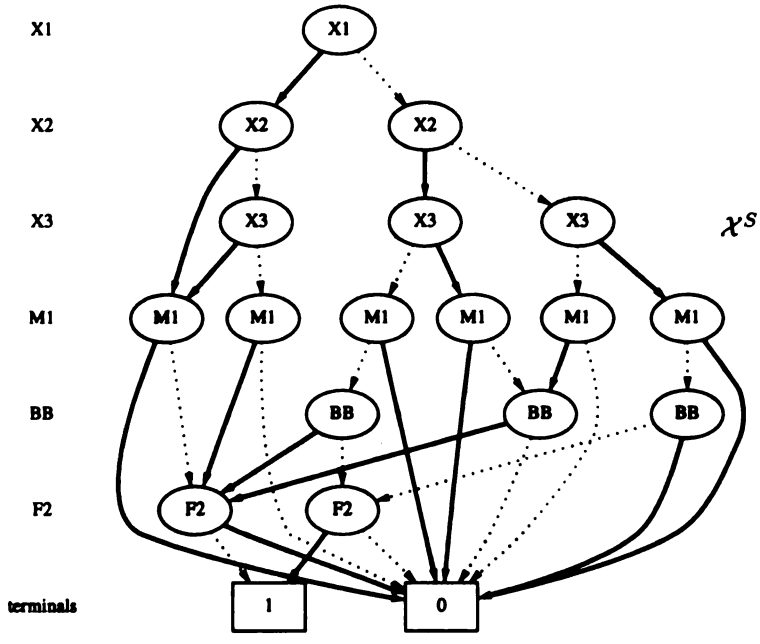
(a)

X_1	X_2	X_3	M_1	BB	F_2	$\mathcal{X}^S$
1	1	*d*	0	*d*	0	1
1	0	1	0	*d*	0	1
1	0	0	1	*d*	0	1
0	1	1	0	1	0	1
0	1	0	0	1	0	1
0	1	0	0	0	1	1
0	0	1	0	0	1	1
0	0	0	1	1	0	1
			otherwise			0

(b)

X_2	X_3	M_1	BB
1	1	0	1
1	0	0	*u*
0	1	0	0
0	0	1	1
otherwise			*d*

(c)



(d)

$$\mathcal{X}^S = (\overline{X_2 \vee X_3} \leftrightarrow M_1) \wedge (\overline{BB \vee M_1} \leftrightarrow F_2) \wedge (\overline{X_1} \wedge \overline{X_2} \wedge \overline{X_3} \rightarrow \overline{F_2}) \wedge (\overline{X_1} \wedge \overline{X_2} \wedge X_3 \rightarrow F_2) \wedge (\overline{X_1} \wedge X_2 \wedge X_3 \rightarrow \overline{F_2})$$

(e)

Figure A.12: **Solving the RFPS solution for `simplecircuit`: #6.** (a) The ATPG test vectors for the `simplecircuit` RFPS Problem (Vectors whose relationships have been represented in the SBDD are shown in italics. The vector whose relationship is newly represented in the SBDD is shown in bold italics.); (b) A truth-table for the structure function represented by (d); (c) The specification of BB encoded in the SBDD; (d) The BDD representing the structure function of `simplecircuit` after including the relationships specified in (a); (e) The symbolic logic expression for the structure function.

X_1	X_2	X_3	F_2
0	*0*	*0*	*0*
0	*0*	*1*	*1*
0	*1*	*1*	*0*
0	***1***	***0***	***1***
1	1	0	0

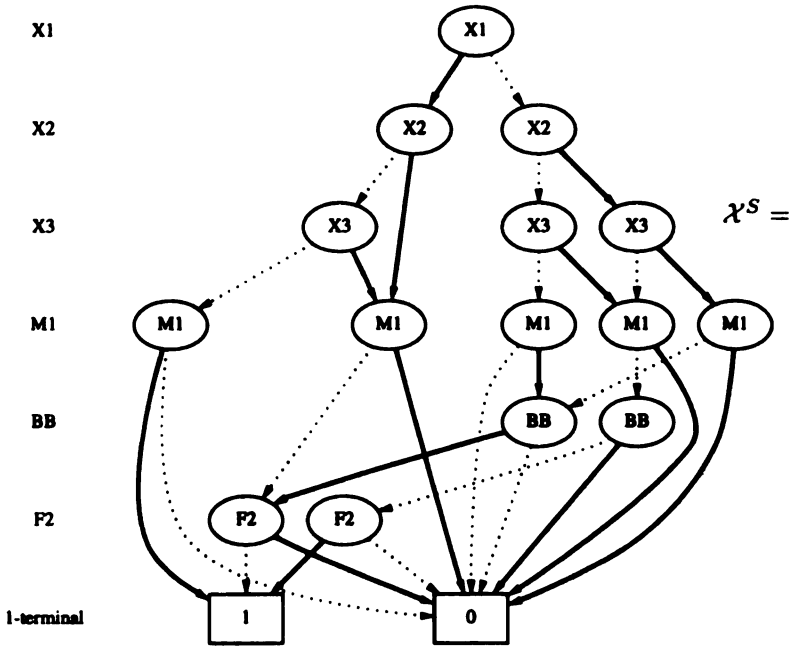
(a)

X_1	X_2	X_3	M_1	BB	F_2	$\mathcal{X}^S$
1	1	*d*	0	*d*	0	1
1	0	1	0	*d*	0	1
1	0	0	1	*d*	0	1
0	1	1	0	1	0	1
0	1	0	0	0	1	1
0	0	1	0	0	1	1
otherwise						0

(b)

X_2	X_3	M_1	BB
1	1	0	1
1	0	0	0
0	1	0	0
0	0	1	1
otherwise			*d*

(c)



(d)

$$\mathcal{X}^S = (\overline{X_2 \vee X_3} \leftrightarrow M_1) \wedge (\overline{BB \vee M_1} \leftrightarrow F_2) \wedge (\overline{X_1} \wedge \overline{X_2} \wedge \overline{X_3} \rightarrow \overline{F_2}) \wedge (\overline{X_1} \wedge \overline{X_2} \wedge X_3 \rightarrow F_2) \wedge (\overline{X_1} \wedge X_2 \wedge X_3 \rightarrow \overline{F_2}) \wedge (\overline{X_1} \wedge X_2 \wedge \overline{X_3} \rightarrow F_2)$$

(e)

Figure A.13: **Solving the RFPS solution for `simplecircuit`: #7.** (a) The ATPG test vectors for the `simplecircuit` RFPS Problem (Vectors whose relationships have been represented in the SBDD are shown in italics. The vector whose relationship is newly represented in the SBDD is shown in bold italics.); (b) A truth-table for the structure function represented by (d); (c) The specification of BB encoded in the SBDD; (d) The BDD representing the structure function of `simplecircuit` after including the relationships specified in (a); (e) The symbolic logic expression for the structure function.

X_1	X_2	X_3	F_2
0	*0*	*0*	*0*
0	*0*	*1*	*1*
0	*1*	*1*	*0*
0	*1*	*0*	*1*
1	**1**	**0**	**0**

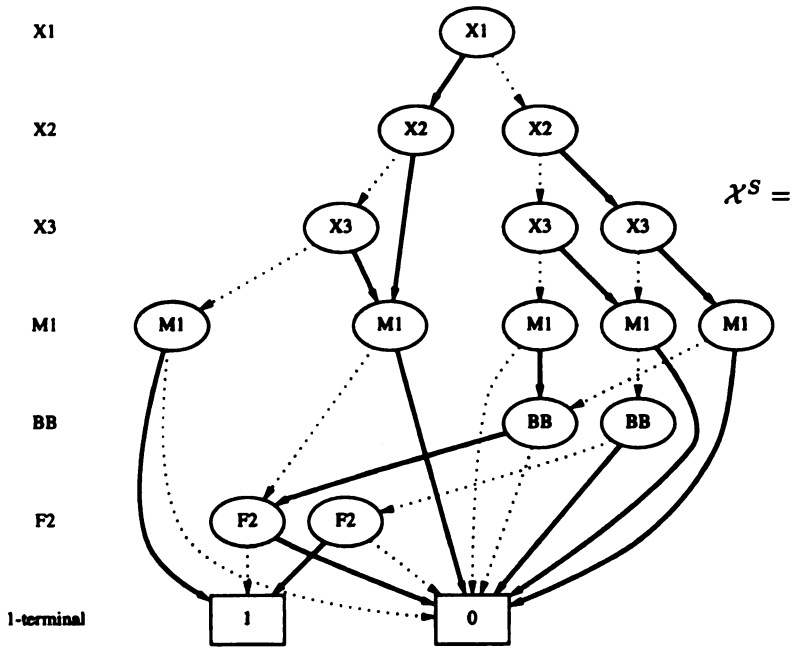
(a)

X_1	X_2	X_3	M_1	BB	F_2	χ^S
1	1	*d*	0	*d*	0	1
1	0	1	0	*d*	0	1
1	0	0	1	*d*	0	1
0	1	1	0	1	0	1
0	1	0	0	0	1	1
0	0	1	0	0	1	1
otherwise						0

(b)

X_2	X_3	M_1	BB
1	1	0	1
1	0	0	0
0	1	0	0
0	0	1	1
otherwise			*d*

(c)



(d)

$$\chi^S = (\overline{X_2 \vee X_3} \leftrightarrow M_1) \wedge$$
$$(\overline{BB \vee M_1} \leftrightarrow F_2) \wedge$$
$$(\overline{X_1} \wedge \overline{X_2} \wedge \overline{X_3} \rightarrow \overline{F_2}) \wedge$$
$$(\overline{X_1} \wedge \overline{X_2} \wedge X_3 \rightarrow F_2) \wedge$$
$$(\overline{X_1} \wedge X_2 \wedge X_3 \rightarrow \overline{F_2}) \wedge$$
$$(\overline{X_1} \wedge X_2 \wedge \overline{X_3} \rightarrow F_2) \wedge$$
$$(X_1 \wedge X_2 \wedge \overline{X_3} \rightarrow \overline{F_2})$$

(e)

Figure A.14: **Solving the RFPS solution for `simplecircuit`: #8.** (a) The ATPG test vectors for the **`simplecircuit`** RFPS Problem (Vectors whose relationships have been represented in the SBDD are shown in italics. The vector whose relationship is newly represented in the SBDD is shown in bold italics.); (b) A truth-table for the structure function represented by (d); (c) The specification of BB encoded in the SBDD; (d) The BDD representing the structure function of **`simplecircuit`** after including the relationships specified in (a); (e) The symbolic logic expression for the structure function.

Appendix B

Computation of vector signatures for the four-bit ALU

An implementation of the TI SN54181 four-bit ALU (Figure B.1) is presented in Figure B.2. Table B.1 presents the function table for the circuit.

The output values of the ALU function for each input's positive and negative unit vector are presented in Table B.2. For each such vector, the one-sum of the output values is calculated. These values are used to determine each input's signature class. For example, the table shows that the sum of the function outputs for the input vector (S2 = 1, all other inputs = 0) is 2. This value is S2's positive vector input signature. Likewise, S2's negative vector input signature is shown to be 7. Thus S2 is a member of the signature class $< 2, 7 >$.

This information is used to partition the inputs into suspect sets (Section 5.3.2) and presented in Table B.3 for a particular implementation of the '181 ALU in which the selection inputs are labeled sel0-sel3, the carry input is labeled Cn', the mode

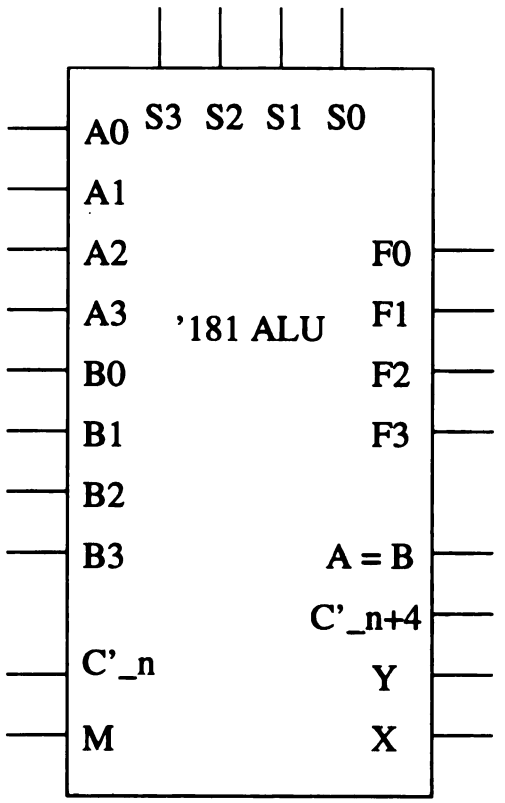


Figure B.1: **TI SN54181 fout-bit ALU.**

```
# file name: 181.blif
.model 181_4bit_ALU
.inputs s0 s1 s2 s3
.outputs f0 f1 f2 f3 aEQb x Cn4_ y
#
.inputs b3 a3
.gate inv a=b3 O=b3_
.gate and3 a=b3 b=s3 c=a3 O=k15
.gate and3 a=a3 b=s2 c=b3_ O=k16
.gate nor2 a=k15 b=k16 O=n32
.gate and2 a=b3_ b=s1 O=k7
.gate and2 a=s0 b=b3 O=k8
.gate buf a=a3 O=a3b
.gate nor3 a=k7 b=k8 c=a3b O=n33
#
.inputs b2 a2
.gate inv a=b2 O=b2_
.gate and3 a=b2 b=s3 c=a2 O=k9
.gate and3 a=a2 b=s2 c=b2_ O=k10
.gate nor2 a=k9 b=k10 O=n22
.gate and2 a=b2_ b=s1 O=k1
.gate and2 a=s0 b=b2 O=k2
.gate buf a=a2 O=a2b
.gate nor3 a=k1 b=k2 c=a2b O=n23
#
.inputs a1 b1
.gate inv a=b1 O=b1_
.gate and3 a=b1 b=s3 c=a1 O=k11
.gate and3 a=a1 b=s2 c=b1_ O=k12
.gate nor2 a=k11 b=k12 O=n12
.gate and2 a=b1_ b=s1 O=k3
.gate and2 a=s0 b=b1 O=k4
.gate buf a=a1 O=a1b
.gate nor3 a=k3 b=k4 c=a1b O=n13
#
.inputs a0 b0
.gate inv a=b0 O=b0_
.gate and3 a=b0 b=s3 c=a0 O=k13
.gate and3 a=a0 b=s2 c=b0_ O=k14
.gate nor2 a=k13 b=k14 O=n02
.gate and2 a=b0_ b=s1 O=k5
.gate and2 a=s0 b=b0 O=k6
.gate buf a=a0 O=a0b
.gate nor3 a=k5 b=k6 c=a0b O=n03
#
.inputs m Cn_
.gate inv a=m O=m_
#
.gate buf a=n33 O=m1
.gate and2 a=n32 b=n23 O=m2
.gate and3 a=n32 b=n22 c=n13 O=m3
.gate and4 a=n32 b=n22 c=n12 d=n03 O=m4
.gate nor4 a=m1 b=m2 c=m3 d=m4 O=y
.gate nand5 a=n22 b=n12 c=n02 d=Cn_ e=n32 O=m5
.gate nand4 a=n32 b=n22 c=n12 d=n02 O=x
.gate inv a=y O=y_
.gate inv a=m5 O=m5_
.gate or2 a=y_ b=m5_ O=Cn4_
.gate xor2 a=n32 b=n33 O=m6
#
.gate and5 a=n02 b=n12 c=n22 d=m_ e=Cn_ O=m7
.gate and4 a=n12 b=n22 c=n03 d=m_ O=m8
.gate and3 a=n22 b=n13 c=m_ O=m9
.gate and2 a=n23 b=m_ O=m10
.gate nor4 a=m7 b=m8 c=m9 d=m10 O=m19
.gate xor2 a=m6 b=m19 O=f3
.gate xor2 a=n22 b=n23 O=m11
#
.gate and4 a=Cn_ b=n02 c=n12 d=m_ O=m12
.gate and3 a=n12 b=n03 c=m_ O=m13
.gate and2 a=n13 b=m_ O=m14
.gate nor3 a=m12 b=m13 c=m14 O=m20
.gate xor2 a=m11 b=m20 O=f2
.gate xor2 a=n12 b=n13 O=m15
#
.gate and3 a=Cn_ b=n02 c=m_ O=m16
.gate and2 a=n03 b=m_ O=m17
.gate nor2 a=m16 b=m17 O=m21
.gate xor2 a=m15 b=m21 O=f1
.gate xor2 a=n02 b=n03 O=m22
#
.gate nand2 a=Cn_ b=m_ O=m18
.gate xor2 a=m22 b=m18 O=f0
.gate and4 a=f3 b=f2 c=f1 d=f0 O=aEQb
.end
```

Figure B.2: **BLIF code for the TI SN54181 four-bit ALU.**

SELECTION				LOGIC $M = 1$	ARITHMETIC $M = 0$	
$S3$	$S2$	$S1$	$S0$		$C_n = 1$	$C_n = 0$
0	0	0	0	$F = \overline{A}$	$F = A$	$F = A$ PLUS 1
0	0	0	1	$F = \overline{A + B}$	$F = A + B$	$F = (A + B)$ PLUS 1
0	0	1	0	$F = \overline{A}B$	$F = A + \overline{B}$	$F = (A + \overline{B})$ PLUS 1
0	0	1	1	$F = 0$	$F =$ MINUS 1 (2's COMPL)	$F =$ ZERO
0	1	0	0	$F = \overline{AB}$	$F = A$ PLUS $A\overline{B}$	$F = A$ PLUS $A\overline{B}$ PLUS 1
0	1	0	1	$F = \overline{B}$	$F = (A + B)$ PLUS $A\overline{B}$	$F = (A + B)$ PLUS $A\overline{B}$ PLUS 1
0	1	1	0	$F = A \oplus B$	$F = A$ MINUS B MINUS 1	$F = A$ MINUS B
0	1	1	1	$F = A\overline{B}$	$F = A\overline{B}$ MINUS 1	$F = A\overline{B}$
1	0	0	0	$F = \overline{A} + B$	$F = A$ PLUS AB	$F = A$ PLUS AB PLUS 1
1	0	0	1	$F = \overline{A \oplus B}$	$F = A$ PLUS B	$F = A$ PLUS B PLUS 1
1	0	1	0	$F = B$	$F = (A + \overline{B})$ PLUS AB	$F = (A + \overline{B})$ PLUS AB PLUS 1
1	0	1	1	$F = AB$	$F = AB$ MINUS 1	$F = AB$
1	1	0	0	$F = 1$	$F = A$ PLUS SHIFTL(A)	$F = A$ PLUS A PLUS 1
1	1	0	1	$F = A + \overline{B}$	$F = (A + B)$ PLUS A	$F = (A + B)$ PLUS A PLUS 1
1	1	1	0	$F = A + B$	$F = (A + \overline{B})$ PLUS A	$F = (A + \overline{B})$ PLUS A PLUS 1
1	1	1	1	$F = A$	$F = A$ MINUS 1	$F = A$

Table B.1: **Functionality of the TI SN54181 four-bit ALU.**

input is labeled m, and the data inputs are labeled a0-a3 and b0-b3. Observe, for example, that the input sel2 (a cluster function variable) corresponding to the input S2 (a pattern function variable) are members of the signature class $< 2, 7 >$.

Inputs	F0	F1	F2	F3	A=B	X	C'_n+4	Y	Sum of Outputs
S3=1, all other inputs=0	1	0	0	0	0	0	1	0	2
S3=0, all other inputs=1	0	0	0	0	0	0	1	1	2
S2=1, all other inputs=0	1	0	0	0	0	0	1	0	2
S2=0, all other inputs=1	1	1	1	1	1	1	0	1	7
S1=1, all other inputs=0	0	0	0	0	0	0	0	1	1
S1=0, all other inputs=1	1	1	1	1	1	1	0	1	7
S0=1, all other inputs=0	1	0	0	0	0	0	1	0	2
S0=0, all other inputs=1	1	1	1	1	1	1	0	1	7
B3=1, all other inputs=0	1	0	0	0	0	0	1	0	2
B3=0, all other inputs=1	1	1	1	1	1	1	0	1	7
B2=1, all other inputs=0	1	0	0	0	0	0	1	0	2
B2=0, all other inputs=1	1	1	1	1	1	1	0	1	7
B1=1, all other inputs=0	1	0	0	0	0	0	1	0	2
B1=0, all other inputs=1	1	1	1	1	1	1	0	1	7
B0=1, all other inputs=0	1	0	0	0	0	0	1	0	2
B0=0, all other inputs=1	1	1	1	1	1	1	0	1	7
A3=1, all other inputs=0	1	0	0	1	0	0	1	0	3
A3=0, all other inputs=1	1	1	1	0	0	1	0	1	5
A2=1, all other inputs=0	1	0	1	0	0	0	1	0	3
A2=0, all other inputs=1	1	1	0	1	0	1	0	1	5
A1=1, all other inputs=0	1	1	0	0	0	0	1	0	3
A1=0, all other inputs=1	1	0	1	1	0	1	0	1	5
A0=1, all other inputs=0	0	1	0	0	0	0	1	0	2
A0=0, all other inputs=1	0	1	1	1	0	1	0	1	5
C'_n=1, all other inputs=0	0	0	0	0	0	0	1	0	1
C'_n=0, all other inputs=1	1	1	1	1	1	1	0	1	7
M=1, all other inputs=0	1	1	1	1	1	0	1	0	6
M=0, all other inputs=1	0	1	1	1	0	1	0	1	5

Table B.2: **Calculation of input vector signatures.** For each input i, this table presents i's positive vector input (i=1, all other inputs = 0) and i's negative vector input (i = 0, all other inputs = 1). The values of the function outputs are shown for each vector input and the one-sum of the outputs is calculated.

Input Name	Vector Input Signature	
	Positive	Negative
sel1, Cn'	1	7
sel3	2	2
a0	2	5
sel0, sel2, b0, b1, b2, b3	2	7
a1, a2, a3	3	5
m	6	5

Table B.3: **Vector input signature for the TI 54181 4-bit ALU.** The positive and negative unit vector input signatures are shown for a 4-bit ALU with selection inputs sel0-3, mode input m, carry input Cn', and data inputs a0-3 and b0-3. The vector signature partitions the function inputs into six signature classes.

Bibliography

Bibliography

Akers, S. (1978). Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516.

Augustus, E. (1990). VLSI circuit layer determination by reflectance for use in reverse engineering. Master's thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, Ohio.

Bahar, R. I., Frohm, E. A., Gaona, C. M., Hachtel, G. D., Macii, E., Pardo, A., and Somenzi, F. (1997). Algebraic decision diagrams and their applications. *Journal of Formal Methods in Systems Design*, 10(2/3):171–206.

Baldwin, R. A. (1994). *A Discipline Independent Framework for Engineering Design.* PhD thesis, Michigan State University.

Benini, L. and De Micheli, G. (1997). A survey of Boolean matching techniques for library binding. *ACM Transactions on Design Automation of Electronic Systems*, 2(3):193–226.

Bochner, M. (1988). LOGEX – an automatic logic extractor from transistor to gate level for CMOS technology. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 517–522.

Brace, K., Bryant, R. E., and Rudell, R. (1990). Efficient implementation of a BDD package. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 40–45.

Bryant, R. E. (1985). Symbolic manipulation of Boolean functions using a graphical representation. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 688–694.

Bryant, R. E. (1986). Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691.

Bryant, R. E. (1993). Symbolic analysis methods for masks, circuits, and systems. In *Proceedings of the IEEE International Conference on Computer Design*, pages 6–8.

Bryant, R. E. (1995). Binary decision diagrams and beyond: Enabling technologies for formal verification. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 236–244.

Burch, J. R., Clarke, E. M., and McMillan, K. L. (1990). Sequential circuit verification using symbolic model checking. In *Proceedings of the ACM/IEEE Design Automation Conference.*

Byrne, E. J. (1992). A conceptual foundation for software re-engineering. In *Proceedings for the Conference on Software Maintenance*, pages 226–235.

Chikofsky, E. J. and Cross, J. H. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17.

De Micheli, G. (1994). *Synthesis and Optimization of Digital Circuits.* McGraw-Hill.

Doom, T. E., White, J. L., Chisholm, G., and Wojcik, A. S. (1998). Identification of functional components in combinational circuits. Technical Report ANL/DIS/TM-47, Argonne National Laboratory.

Dukes, M., Brown, F., and DeGroat, J. (1994). A generalized extraction system for VHDL. In *Proceedings of the IEEE International ASIC Conference and Exhibit*, pages 165–171.

Dukes, M. A. (1994). Generating VHDL models from inadequately-documented integrated circuits. In *Proceedings of the Conference on Advances in Modeling and Simulation*, pages 165–171.

Eckmann, S. and Chisholm, G. (1997). Assigning functional meaning to digital circuits. Technical Report ANL/DIS/TM-43, Argonne National Laboratory.

Fretheim, E. (1988). Reverse engineering VLSI using pattern recognition techniques. Master's thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, Ohio.

Fujita, M., Fujisawa, H., and Kawato, N. (1988). Evaluation and improvements of Boolean comparison method based on binary decision diagrams. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 2–5.

Hayden, R. (1989). Analysis system for reverse engineering VLSI circuits. Master's thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, Ohio.

Jain, J., Mukherjee, R., and Fujita, M. (1995). Advanced verification techniques based on learning. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 420–426.

Jain, J., Narayan, A., Fujita, M., and Sangiovanni-Vincentelli, A. (1997). A survey of techniques for formal verification of combinational circuits. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 445–454.

Keutzer, K. (1996). The need for formal methods for integrated circuit design. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, volume 1166, pages 1–18. Springer Lecture Notes in Computer Science.

Khalil, M. A. (1998). Redesign process of digital VLSI circuit with incomplete implementation information. Master's thesis, Department of Electrical Engineering, Michigan State University, East Lansing, Michigan.

Kunz, W. and Pradhan, D. (1994). Recursive learning: A new implication technique for efficient solutions to CAD problems - test, verification, and optimization. *IEEE Transactions on Computer-aided Design of Integrated Circuits*, 13(9):1143–1158.

Lai, Y., Sastry, S., and Pedram, M. (1992). Boolean matching using binary decision diagrams with applications to logic synthesis and verification. In *Proceedings of the IEEE International Conference on Computer Design*, pages 452–458.

Lee, C. (1959). Representation of switching circuits by binary decision programs. *Bell System Technical Journal*, 38:509–516.

Long, D. (v1996). [URL: ftp://emc.cs.cmu.edu/pub/bdd/bdlib.tar.Z].

Luellau, F., Iloepken, T., and Barke, E. (1984). A technology independent block extraction algorithm. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 610–615.

Madre, J.-C. and Billon, J.-P. (1988). Proving circuit correctness using formal comparison between expected and extracted behavior. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 205–210.

Mailhot, F. (1991). *Technology Mapping for VLSI Circuits exploiting Boolean Properties and Operations.* PhD thesis, Stanford University.

Mailhot, F. and De Micheli, G. (1993). Algorithms for technology mapping based on binary decision diagrams and on Boolean operations. *IEEE Transactions on CAD/ICAS*, 12(5):599–620.

Malik, S., Wang, A., Brayton, R., and Sangiovanni-Vincentelli, A. (1988). Logic verification using binary decision diagrams in a logic synthesis environment. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 6–9.

Martin, J. C. (1997). *Introduction to Languages and the Theory of Computation - 2nd ed.* McGraw-Hill, New York.

Matsunaga, Y. and Fugita, M. (1989). Multi-level logic optimization using binary decision diagrams. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 556–559.

McElvain, K. (1993). LGSynth93 benchmark set: Version 4.0. [URL: ftp://ftp.mcnc-.org/pub/benchmark/Benchmark_dirs/LGSynth93].

Minato, S. (1996). *Binary Decision Diagrams and Applications for VLSI CAD.* Klewer, Hingham, MA.

Minato, S., Ishiura, N., and Yajima, S. (1990). Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 52–57.

Mueller, M. (1989). Investigation of Gabor filters for use in reverse engineering VLSI. Master's thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, Ohio.

Ohlrich, M., Ebeling, C., Ginting, E., and Sather, L. (1993). Subgemini: Identifying subcircuits using a fast subgraph isomorphism algorithm. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 31–37.

Querns, J. (1989). Segmentation of regions of contiguous common composition on VLSI circuits. Master's thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base, Ohio.

Ranjan, R., Sanghavi, J., Brayton, R., and Sangiovanni-Vincentelli, A. (1996). Binary decision diagrams on network of workstations. In *Proceedings of the Internation-all Conference on Computer Design: VLSI in Computers and Processors*, pages 358–364, Austin, Texas. IEEE.

Rekoff, M. G. (1985). On reverse engineering. *IEEE Trans. on Systems, Man, and Cybernertics*, SMC-15(2):244–252.

REW'98 (1998). *Reverse Engineering Workshop*, Del Mar, California. Sponsored by Argonne National Laboratory.

Rudell, R. (1993). Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of IEEE International Conference on Computer-Aided Design*, pages 42–47.

Sentovich et al. (1992). *SIS: A System for Sequential Circuit Synthesis.* Department of Electrical Engineering and Computer Science, University of California, Berkeley, California.

Sentovich, E. M. (1996). A brief study of BDD package performance. In *Proceedings of the First International Conference, Formal Methods in Computer-Aided Design*, Volume 1166 of Lecture Notes in Computer Science, pages 389–403, Palo Alto, California. Springer-Verlag.

Shannon, C. E. (1938). A symbolic analysis of relay and switching circuits. *Trans. AIEE*, 57:713–723.

Shannon, C. E. (1949). The synthesis of two-terminal switching circuits. *Bell System Technical Journal*, 28:59–98.

Somenzi, F. (1997). CUDD: Colorado University Decision Diagram package. [URL: http://www.bessie.colorado.edu/~fabio/CUDD]. Release 2.1.2.

Su, W., Michael, G., and Dukes, M. (1994). Automated generation of VHDL models by recognizing paper schematics of electronic systems. In *Proceedings of the Government Microcircuits Applications Conference*, pages 39–42.

Wey, C.-L. and Khalil, M. A. (1998). Redesignability analysis of digital VLSI circuits with incomplete implementation information. In *Proceedings of IEEE International Symposium on Circuits and Systems.*

White, J., Doom, T., Wojcik, A., Chung, M., and Chisholm, G. (1997). Candidate subcircuit generation to facilitate identification of high-level components in logic circuits. Technical Report MSUCPS:TR97-48, Department of Computer Science, Michigan State University. http://web.cps.msu.edu/TR/MSUCPS:TR97-48.

Wojcik, A., Wey, C., Doom, T., and Samarziya, J. (1997). An approach to the redesign of digital circuits from partial information. Technical Report MSUCPS:TR97-47, Department of Computer Science, Michigan State University. http://web.cps-.msu.edu/TR/MSUCPS:TR97-47.

Wu, Q., Chen, C., and Acken, J. (1994). Efficient boolean matching algorithm for cell libraries. In *Proceedings of the IEEE International Conference on Computer Design*, pages 36–39.