SYNTHESIS OF SEQUENTIAL CIRCUITS FOR VLSI DESIGN

by

Pranav N. Ashar

Memorandum No. UCB/ERL M92/62

5 June 1992

SYNTHESIS OF SEQUENTIAL CIRCUITS FOR VLSI DESIGN

by

Pranav N. Ashar

Memorandum No. UCB/ERL M92/62

5 May 1992

TITLE PAGE

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Synthesis of Sequential Circuits for VLSI Design

Pranav N. Ashar

University of California
Berkeley, California

Department of Electrical Engineering
and Computer Science

Abstract

Synthesis of VLSI circuits involves transforming a specification of circuit behavior into a mask-level layout which can be fabricated using VLSI manufacturing processes. Optimization strategies are vital in VLSI synthesis in order to meet desired specifications. The first-order optimization criteria in this process are typically all or a desired subset of area optimality, speed and testability. There are several steps in the synthesis of mask-level layout descriptions from specifications of circuit behavior. The steps involved in the transformation from the RT-level to a gate-level circuit are collectively known as *logic synthesis*. VLSI circuits are *sequential circuits*, *i.e.* they contain memory or storage elements, namely flip-flops or latches, as well as combinational (or switching) circuitry. Considerable progress has been made in the understanding of combinational logic optimization in the recent past. Consequently a large number of university and industrial CAD programs are now available for the optimal logic synthesis of combinational circuits. The understanding of sequential circuit optimization, on the other hand, is considerably less mature. The presence of internal state adds considerably to the complexity of the optimization problem. Synthesis tools are required to automatically encode the symbolic internal states. This encoding determines the complexity and the structure of the sequential circuit realizing the state machine, and therefore has a profound effect on its area, testability and performance. Techniques for the optimal synthesis of sequential circuits are presented in this thesis. Algorithms are presented for finite-state-machine decomposition using information available at the symbolic state graph level, for symbolic state graph extraction from logic-level descriptions, for the synthesis of sequential circuits for enhanced testability under the single and multiple stuck-at fault models, and for sequential logic verification. These algorithms have been implemented in a sequential synthesis program called FLAMES.

Prof. Arthur Richard Newton
Thesis Committee Chairman

Acknowledgements

To Richard Newton I owe gratitude for having the confidence in me to take me on as his graduate student, and for standing by me through the good times and the bad. At various times, and on occasion all together, he has been my professional, philosophical and emotional guide.

Much of my work was inspired by ideas that originated with Srinivas Devadas. He has been both, a close research associate and an even closer personal friend. I have learnt a great deal from him. But, most of all, it has been his example of single minded dedication to work that I have tried to emulate. It has been a pleasure working with him.

The example set by Professors Robert Brayton, Donald Pederson and Alberto Sangiovanni-Vincentelli in the CAD group has been inspiring. I have benefited enormously by virtue of my association with them. Professors Robert Brayton, Jan Rabaey and Don Glaser served on my qualifying exam committee, and Professors Robert Brayton and Don Glaser are co-signees on my dissertation. I am grateful to them for sparing the time from their busy schedules.

Kurt Keutzer, Tony Ma, Richard Rudell and Albert Wang, now all at Synopsys, but at some time or the other associated with the CAD group at Berkeley, have always been forthcoming with useful advice, help and ideas. I also have had the opportunity to interact with Tim Cheng at AT&T, Gary Hachtel and Fabio Somenzi at the University of Colorado, Boulder, Jim Kukula visiting MIT from IBM, Wayne Wolfe at Princeton University, and Ray Wei at Cadence.

My research was supported in part by financial assistance from the Defense Advanced Research Projects Agency under contracts N00014-87-K-0825 and N00039-87-C-0182, from AT&T Bell Laboratories and from Semiconductor Research Corporation, and by equipment grants from Digital Equipment Corporation. Their support is gratefully acknowledged.

It is difficult to imagine a CAD group without Kia Cooper, Elise Mills and Flora Oviedo. Their unquestioning help ensured that the graduate students could concentrate on research. Brad Krebs, Mike Kiernan and Kurt Pires were instrumental in maintaining the computing environment for the CAD group. I would like to thank them, in particular, for taking the time to attend to my individual needs. Over the years, Brad has received considerable help from the students in the group who have taken time off their own research

for the benefit of the community. Wendell Baker has always been magnanimous with his time. The environment for software development created by David Harrison, Peter Moore, Tom Quarles, Rick Rudell and Rick Spicklemier among others, tailored to meet the specific needs of our group, was instrumental in making my programming tasks considerably simpler. Also crucial to me were the TeX and Postscript based document preparation tools that were maintained by Rick Spicklemier et al. And, a huge thanks to Richard Newton for introducing all of us to Macintoshes, and to Brian Okrafka for handling the installation and maintenance of software on them.

The four years at Berkeley would have been routine without the company of the so many interesting, smart and diverse people that I had the opportunity to form close personal and professional relationships with. I am grateful to them for giving me this opportunity, and I hope, possibly in vain, that I contributed as much to them as they did to me. Given the nature of graduate school, it is common to interact with people at various stages of completion of their degrees. Over four years therefore, one can make an extremely large number of friends. At the risk of missing out some names and making this paragraph read like a roster of names, I would like to acknowledge the friendship, support and advice I received over the years from Abhijit, Alan, Alex Gollu, Alex Saldanha, Albert, Andrea, Arvind, Bill, Brian Lee, Brian Okrafka, Cho, Chuck, Cormac, Dev, Ellen, Gary, Greg Whitcomb, Hamid, Herve', Jaijeet, Karti, Ken, Luciano, Mark, Mitch, Narendra, Rajiv Ramaswami, Rajeev Murgai, Ramin, Rick McGeer, Rick Spicklemier, Rick Rudell, Sharad, Srini, Theo, Tim Kam, Tiziano, Tom Quarles, Tom Shiple, Tony, Umakanta, Vijay, Wayne, Wendell and Yoshi. Relationships with some were very special to me. I would like to say to these people that I am indebted to them for life, and that the memories of the times spent with them at Berkeley will always be with me, cherished forever and relived again and again.

This dissertation is dedicated to my Parents, Rajni and Navin.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Computer-Aided VLSI Design

Computer-Aided Design (CAD) of microelectronic circuits is concerned with the development of computer programs for the automated design and manufacture of integrated electronic systems, with emphasis today on Very Large Scale Integrated (VLSI) circuits. Automated VLSI design is referred to as *VLSI synthesis*. Synthesis of VLSI circuits involves transforming a specification of circuit behavior into a mask-level layout which can be fabricated using VLSI manufacturing processes, usually via a number of levels of representation between abstract behavior and mask-level layout. Optimization strategies, both manual and automatic, are vital in VLSI synthesis in order to meet required specifications. However, the optimization problems encountered in VLSI synthesis are typically NP-hard. Therefore, solutions to the optimization problems incorporate heuristic strategies, the development of which requires a thorough understanding of the problem at hand. Thus, automatic optimization-based VLSI synthesis has evolved into a rich and exciting area of research.

Direct application of synthesis in industry has been a significant driving force for research in CAD. Simple marketing principles dictate that, other factors being equal, a product available sooner would capture a larger share of the market and would remain in use longer. The desire to reduce the time to design and manufacture has led to the initial investment of considerable money and effort into the development of CAD tools capable of producing designs competitive with the best manual designs. Today, constantly shrinking geometries and increasingly reliable manufacturing processes have led to complex

systems being implemented on a single chip, making the use of CAD tools commonplace and mandatory. In its turn, the rapid automation of the VLSI design phase has allowed companies to keep pace with advances in other areas of VLSI like computer architecture and manufacturing, leading to a symbiotic relationship. As a consequence of this rapid development in VLSI technology, it is currently possible to produce application-specific integrated circuits (ASICs), microprocessors and other types of circuits that contain millions of transistors.

1.2 The Synthesis Pipeline

There are several steps in the synthesis of mask-level layout descriptions from specifications of circuit behavior. *Behavioral synthesis* begins with a programming-language-like description of the functionality and converts it to a register-transfer-level (RT-level) description that implements the desired functionality. Among the issues involved at this stage are the temporal scheduling of operations and the allocation of hardware. For instance, decisions regarding the number of arithmetic units in a digital signal processor are made in this step.

The steps involved in the transformation from the RT-level to a gate-level circuit are collectively known as *logic synthesis*. Switching and automata theory form the cornerstones of logic synthesis. Even though obtaining *some* gate-level circuit from a RT-level description is straightforward, it is nontrivial to obtain a gate-level circuit that conforms to the desired specifications. The first-order optimization criteria in this process are typically all or a desired subset of area optimality, speed and testability.

Once the gate-level circuit has been obtained, mask-level layout is derived using *layout synthesis* (physical design) tools. The physical design styles of choice are typically gate array, sea-of-gates, standard-cell and programmable gate array. Programmable gate array is popular for extremely rapid prototyping of designs. Standard-cell is the design style of choice in mostly-custom designs like microprocessors, where only a portion of the chip is synthesized automatically. Gate array and sea-of-gates offer superior performance and integration density compared to programmable gate array and standard-cell design styles and are chosen when the complete chip is synthesized automatically.

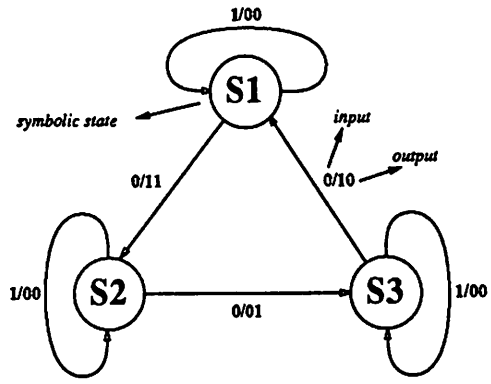


Figure 1.1: An example of a State Transition Graph (STG)

1.3 Sequential Logic Synthesis

Almost all VLSI circuits are *sequential circuits*, *i.e.* they contain memory or storage elements in the form of flip-flops or latches as well as combinational (or switching) circuitry. An RT-level description can be implemented using either synchronous or asynchronous sequential logic. While asynchronous design has certain advantages, design automation for the reliable asynchronous implementation of complex functionality is still in its infancy. The synchronous design paradigm is followed throughout this thesis.

Considerable progress has been made in the understanding of combinational logic optimization in the recent past and consequently a large number of university and industrial CAD programs are now available for the optimal synthesis of combinational circuits [15, 12, 55, 29]. These optimization programs produce results competitive with manually-designed logic circuits. For a review of current combinational optimization techniques, the reader is referred to [17, 19].

The understanding of sequential circuit optimization, on the other hand, is considerably less mature. The presence of internal state adds considerably to the complexity of the optimization problem. While the primary inputs and outputs are typically binary vectors at the RT-level, internal states are represented in symbolic form.

Sequential circuits are most often modeled using Finite-State Machines (FSMs). A FSM is a mathematical model of a system (in our case, a switching circuit) with discrete inputs, discrete outputs and and a finite number of internal configurations or states. The state of a system completely summarizes the information concerning the past inputs to the system that is needed to determine its behavior on subsequent inputs. It is convenient

to visualize a FSM as a directed graph with nodes representing the states and the edges representing the transitions between states. Such a graph is known as a State Transition Graph (STG). An edge in the STG is labeled by the input causing the transition and the output asserted on the transition. A FSM can also be equivalently represented in tabular form by a State Transition Table (STT), each row of which corresponds to an edge in the STG. To deal with complexity, VLSI circuits are invariably specified in a hierarchical fashion. Large sequential circuits are typically modeled by smaller, interacting FSMs.

Synthesis tools are required to encode the internal symbolic states of FSMs as binary strings. This encoding determines the complexity and the structure of the sequential circuit which realizes the FSM, and therefore has a profound effect on its area, testability and performance. Stated differently, synthesis tools have the freedom of encoding states in such a way that the design constraints are satisfied. The notion of structure is generally associated with the manner in which a machine can be realized from an interconnection of smaller component machines as well as with the functional interdependencies of its state and output variables. It may be desirable, for example, to construct the circuit with the minimum amount of logic, or to build it from an interconnection of smaller circuits to obtain superior performance.

A second degree of freedom available to sequential synthesis tools is based on the fact that the STG corresponding to a given functionality is not unique; transformations to the STG like state splitting, state merging or STG partitioning enable moving from one STG to another without changing the functionality. These transformations guide the encoding of states in a particular direction, in many cases into directions that would not have been taken otherwise. Such transformations are sometimes necessary to achieve the desired objectives.

Research into such transformations is exactly what has motivated the work presented in this thesis. In particular, transformations that guide the synthesis of interconnected FSMs have been investigated. Topics that have been covered under this general guideline include FSM decomposition, state transition graph extraction, the use of don't-cares in interacting sequential circuits, and the synthesis of interacting sequential circuits for single and multiple fault testability.

1.4 Early Work in Sequential Logic Synthesis

Work in sequential logic synthesis dates back to the late '40s and '50s when discrete off-the-shelf components (relays and vacuum tubes) were used. In the 1960's, small-scale integrated circuits(SSI) became popular and much of the work in that period was motivated by the need to reduce the number of latches in the circuit since that meant a reduction in the number of relatively expensive chips on the circuit board. Also, since combinational logic synthesis was still in its infancy, techniques for state encoding and FSM decomposition were unable to target the combinational logic complexity of the sequential circuit effectively.

The minimization of the number of states in completely specified FSMs was first investigated by Moore [88], Huffman [63, 64] and Mealy [84]. This work was later extended to the reduction of states in incompletely specified machines by Ginsburg [53] and Unger [89]. An interesting technique for deriving maximal compatibles in state reduction using Boolean algebra was reported in a short communication by Marcus [82].

The relationship between state encoding and the structure of the resulting sequential circuit was first investigated in terms of the algebraic theory of partitions by Hartmanis [58, 59] and later by Hartmanis and Stearns [102, 60]. Contributions to machine-structure theory were also made by Karp [65], Kohavi [68], Krohn and Rhodes [70], Yoeli [109, 110] and Zeiger [111]. The concept of state splitting to augment the possibilities of finding desirable decompositions and state assignments was developed, among others, by Hartmanis and Stearns [60], Zeiger [111] and Yoeli [109]. The book by Hennie [61] provides a lucid and intuitive description of the above contributions.

State encoding was treated from a different point of view by Armstrong [2] and Dolotta and McCluskey [43]. The procedure of Armstrong formulated the state encoding problem as one of assigning codes so that prederived adjacency relationships between states are satisfied in the Boolean domain. To a certain extent, the procedure of [2] inspired some state assignment algorithms developed in recent work (*e.g.* Devadas *et al* [36]) for targeting multilevel implementations.

1.5 Recent Developments

1.5.1 State Encoding

A number of significant new results have been obtained in the area of sequential logic synthesis in the last five years (*e.g.* [36, 86, 87, 96]). An important development in state encoding was the step from predictive to exact approaches for state encoding targeting two-level implementations. A fundamental result which made this possible was the establishment of a link between the size of a minimized symbolic tabular representation of a FSM and the maximum number of product terms required in a Programmable Logic Array (PLA) implementing the same FSM after encoding the states in the work by De Micheli *et al* [86]. The approach followed in [86] involved a two-step process. In the first step, the STT of the FSM is symbolically minimized using a two-level multiple-valued minimization program like ESPRESSO-MV [94]. This minimization step generates constraints that the state codes must satisfy if the PLA resulting from the encoding is to have as small or an equal number of product terms as the minimized STT. Obtaining a state encoding that satisfies the constraints forms the second step of the procedure. Since symbolic minimization by itself cannot account for the interactions in the next-state plane of the encoded FSM, the approach of [86] effectively approximated the state encoding problem as one of input encoding.

States appear both in the input and output planes of the PLA and therefore state encoding is actually an input-output encoding problem. The interactions between product terms in the next-state plane can occur either due to dominance or disjunctive relationships between state codes. If the code for state s_1 dominates the code for state s_2, the input parts of the cubes asserting the next-state s_1 can be used as don't-cares in order to optimize the cubes asserting the next-state s_2. Similarly, if the code for state s_1 is the disjunction of the codes for states s_2 and s_3, the input part of the cubes that assert the next-state s_1 can be optimized using the input parts of the cubes asserting the next-states s_2 and s_3.

Work by De Micheli [85] was the first to take advantage of interactions between product terms in the next-state plane. In particular, it attempted to maximize the cardinality reductions due to dominance relationships between state codes by reducing the problem to one of heuristically finding the order in which states should be encoded. Further understanding of interactions in the output plane led to the work of Devadas and Newton [41] in which a procedure was presented for encoding states to achieve the minimum product-term count. By means of this procedure, relationships between codes arising due to the

interactions in the input plane and due to both dominance and disjunctive relationships in the output plane can be handled simultaneously. The search for all possible relationships and their effects is carried out by modifying the classical Quine-McCluskey [90, 83] prime implicant generation and covering. The notion of Generalized Prime Implicants (GPIs) was introduced in [41] for that purpose. GPIs correspond to a weaker form of primality than conventional prime implicants in that for the same symbolic cover, the set of GPIs contains the set of conventional prime implicants and is much larger than it. The advantage of GPIs is that they allow interactions in the output plane to be handled formally. While the use of GPIs leads to an exact procedure, their use is only viable for small FSM examples.

State encoding targeting multilevel implementations is an even more difficult problem. The main reason for this is that combinational optimization of multilevel circuits is itself not an exact science. Even so, certain optimization strategies like common-factor extraction are fundamental to multilevel optimization and a number of predictive state encoding procedures have been proposed that maximize the gains due to these basic strategies. Devadas *et al* [36] proposed an algorithm for state encoding in which the likelihood of finding common subexpressions and common cubes in the logic prior to optimization is enhanced by minimizing the distance in the Boolean space between chosen states. A variation on this approach was followed in subsequent work by other researchers [74, 107]. However, in [74], the emphasis was on the general encoding problem, including both inputs, outputs and state variables, while the emphasis in [107] was on encoding for optimization by kernel-based combinational optimization tools. Recently, attempts have been made to extend the encoding paradigm followed for two-level circuits to multilevel circuits. In the work of Malik *et al* [71], techniques were proposed for optimizing multilevel circuits with multiple-valued input variables. As in the two-level case, these optimizations lead to constraints on possible binary encodings for the multiple-valued variables that must be satisfied if the effects of the optimizations are to be preserved after encoding.

One of the important steps in the encoding of symbolic states in a FSM is the satisfaction of encoding constraints. In the most general case, these constraints consist of both input and output constraints. In the case of input constraints, one is given a constraint matrix, C, each column of which corresponds to a state and each row to a constraint. The goal is to find an encoding matrix E where each row in E corresponds to the binary code chosen for a state, such that the constraints implied by C are satisfied and the number of columns in E is minimum. Satisfying the input constraints entails obtaining state codes

such that for each row in C the states present in the row form a face in the Boolean n-space and the states absent from the row are excluded from that face. This problem is called the face-embedding problem. Output constraints, on the other hand, force bitwise dominance and disjunctive relationships between the state codes.

Finding a minimum-length encoding satisfying the constraints is an *NP-hard problem* [95]. De Micheli *et al* [86] provided a number of results for reducing C without violating the original set of constraints and proposed a row-based encoding algorithm. According to this algorithm, E is constructed row by row with the invariant that the constraints corresponding to the rows of C are not violated by the portion of E constructed thus far. Columns are added to E when necessary. This algorithm was found to be effective only for small examples. Column-based algorithms were proposed by De Micheli [85] and Devadas *et al* [42]. In a column-based algorithm, E is constructed one column (one bit) at a time. None of these algorithms guarantee a minimum-length encoding. The work by Villa *et al* [106] includes an algorithm for obtaining the minimum length encoding satisfying all input constraints. This work represents a refinement of the methods developed previously [85, 86].

An alternate approach to constraint satisfaction uses the notion of dichotomies. A dichotomy is defined as a disjoint two-block partition on a set, in our case the set of states. The notion of dichotomies was first introduced for hazard-free state encoding of asynchronous circuits by Tracey [105]. Dichotomies were revisited for constrained state encoding by Ciesielski *et al* [108]. The main result of that work was to show that the minimum number of bits required to encode a set of constraints is equal to the minimum number of prime dichotomies required to cover all the seed dichotomies. An implementation based on graph coloring was suggested for this approach. An alternate implementation of the dichotomies approach based on prime generation and classical unate covering [93] was done by Saldanha *et al* [95]. It was also shown in that work how output constraints could be handled using dichotomies. In this case, the generation of primes was carried out using the same approach as used by Marcus [82] for generating maximal compatibles in state minimization.

In other work on constraint satisfaction [41], it was shown that if the encoding length were known *a priori*, all constraints can be represented by Boolean equations. While this idea is attractive in theory, a naive representation of all constraints involved as Boolean equations can lead to an intractable satisfiability problem. Simulated annealing has also

been used successfully for constraint satisfaction [36, 74, 106].

1.5.2 Finite State Machine Decomposition

FSM decomposition can be used to obtain partitioned sequential circuits with the desired interconnection topology. Sequential circuit partitioning can lead to improved performance, testability and ease of floor-planning. Decomposition at the STG-level allows a larger solution space to be searched for partitioning sequential circuits than techniques that operate at the logic level. However, the drawbacks of the above approaches which operate at the STG level is that it is difficult to accurately predict the effect of an operation at the symbolic level on the cost of the resulting logic.

In recent work on FSM decomposition, Devadas and Newton [40] recognized that many FSMs possess isomorphic subgraphs in their STGs. The implementation of multiple instances of such isomorphic subgraphs (called *factors* in [40]) as a single separate submachine distinct from the parent machine can lead to reduced area and improved performance. The authors also demonstrated that this decomposition approach leads to an encoding strategy that takes advantage of some interactions in the next-state plane.

1.5.3 Sequential Resynthesis at the Logic-Level

The observation that significant gains could be accrued by optimizing certain sequential circuits at the logic-level was made by Leiserson *et al* [72] for systolic arrays. Systolic arrays are sequential circuits which are capable of operating at very high clock frequencies because they are designed as highly pipelined structures with very little logic between pipeline latches on any path in the network. This design methodology implies that the number of latches is usually very large. Given an initial design, the problem of retiming or relocating these latches so that the number of latches is minimized, with the circuit still satisfying the clock frequency requirement and with the functionality of the circuit remaining unchanged, was formulated as an integer programming problem in that work. The work of Malik *et al* [79] extended the ability to retime latches to a larger class of circuits. In that work, the general problem of state encoding was reduced to one of latch retiming. While this notion is certainly attractive in theory, it was found that most sequential circuits implementing control-type functions are not amenable to global optimization by latch retiming because of the tight feedback paths they possess. The reason that latch

retiming was successful for systolic arrays was that most of the latches are used as pipeline latches and very few of them are in unique feedback loops.

1.5.4 FSM Verification

In order to validate the complex transformations involved in logic synthesis, it is important to be able to verify the equivalence of the input/output behavior of two implementations of a FSM. Clearly, exhaustive enumeration of all sequences of states and edges in the two implementations is not a viable proposition. In the work of Devadas *et al* [35], implicit cube-enumeration was used to avoid exhaustive search. In this approach, the verification is carried out as a two step process. A path is first enumerated on one of the FSMs, with the primary input combinations on this path being cubes in general. This path is then simulated on the second FSM to check if the same output sequence is produced as in the first FSM. Podem-based [54] justification techniques are used for implicit cube-enumeration. Ghosh *et al* [52] proposed a variation on [35] in which the next-state space as well as the primary-input space is enumerated implicitly. While this approach still basically involves a depth-first search, some states can be visited together.

The recent work of Coudert *et al* [28] pioneered the use of characteristic-function-based representations of sequential machines using binary decision diagrams, and the associated techniques for range computation for application in FSM verification. The fundamental contribution of this research was to illustrate that an implicit breadth-first traversal of the state space is sufficient for FSM verification. This technology was improved considerably in the work by Touati *et al* [104] leading to its application to, among other things, the computation of equivalent states in sequential circuits [75].

1.5.5 Sequential Synthesis for Testability

Ensuring the correctness of chips leaving the fabrication line is universally recognized as a key area. The typical testing methodology involves applying a series of test-vectors as inputs to the fabricated die and comparing the output produced against the expected output. The die is deemed defective and discarded if even one of the outputs so produced is different from the expected output. One of the reasons why defective die are not detected by this process is that the defect only modifies redundant portions of the die. Such a defect causes the functionality of the chip to remain unchanged leaving the fault

undetectable or *redundant.* The inability to detect die with redundant faults can lead to erroneous conclusions about the quality of the fabrication process and the quality of the masks. A circuit with no undetectable faults under a given fault model is said to be fully testable under that fault model.

The intimate relationship of logic synthesis to the testability of combinational circuits under the stuck-at-fault model has been known for some time [67, 14, 10, 56] and a number of synthesis procedures have been proposed for realizing fully testable implementations of combinational circuits [10, 56, 99]. An important contribution in that regard was that of Bartlett *et al* [10] in which a link was established between combinational testability and don't-care-based combinational logic optimization. In the work by Hachtel *et al* [57], a procedure was proposed for the synthesis of fully multiple-fault testable multilevel circuits based on the results of Kohavi [67] for multiple faults in two-level circuits.

However, since VLSI circuits in general are sequential circuits, they contain memory or storage elements in addition to combinational circuitry. Many latches occupy feedback paths that feed an output of the combinational portion of the circuit back into one of its inputs. Such latches are usually neither directly observable nor controllable. In the context of testing, this implies that for a sequential circuit to be fully testable it is not sufficient for merely the combinational portion to be fully testable; rather, for a fault to be detected, the fault effect must be propagated to those outputs that are observable, and the fault must be excited from those inputs that are controllable. As a result, the testing problem for sequential circuits is more complicated than in the combinational case.

A design-for-testability (DFT) engineering solution, called Scan Design [44], can be used to convert a sequential circuit testing problem to a combinational circuit testing problem by making the latches directly accessible. In the IBM approach [44], the latches are linked to form a serial shift register for scanning a set of values into and out of the latches. In other approaches, parallel load/unload techniques are used (*e.g.* [91]). Given the large number of latches in a typical chip, allowing parallel access to all the latches is not feasible, so the usual approach followed today is to allow serial access to all the latches. This approach involves tailoring the whole design around the testing methodology and the special latches required. In addition, a performance penalty is involved because of the complex nature of the latches. Another drawback associated with this approach is in the large testing time required since each test vector must be scanned serially in and out, one bit at a time.

In contrast to scan design, non-scan testing methodology involves testing sequential circuits without necessarily making all the latches directly accessible. The non-scan testing effort relies on the philosophy that logic present in a circuit plays a useful role only if it has an effect on what is seen at the observable outputs, based on what is applied at the accessible inputs. Therefore, a fault in such logic should be testable without making the latches accessible. In fact, if a fault cannot be tested in this manner, the associated logic is redundant and should not have been used in the first place. In non-scan testing, a *sequence* of test vectors has to be applied in general to detect each fault. An important development in non-scan sequential testing was the definition [39] of sequential redundancies with associated don't-cares which, if used optimally, would result in a fully testable sequential circuit, without direct access to any internal storage. Other techniques employing circuit partitioning accompanied by encoding constraints have been proposed for non-scan sequential testability [38]. Techniques for partial scan in which a subset of latches is made scannable have also been proposed (*e.g.* [1, 78]).

1.6 Overview of Dissertation

The goal of this work is to develop new techniques for the synthesis of sequential logic circuits. The basic terminology used in the dissertation is defined in Chapter 2. Chapter 2 also contains a detailed review of the basic concepts related to state encoding. This review has been included to facilitate the understanding of Chapter 3.

In Chapter 3, a new approach to FSM decomposition is presented. By virtue of this procedure, it is possible to target logic-level optimality of the partitioned sequential circuit. For the cost function chosen, the algorithm presented in this chapter can be used to obtain a decomposition with minimum cost. In many ways, state encoding and FSM decomposition are two sides of the same coin. FSM decomposition can be viewed as a structural transformation of the FSM that guides the subsequent steps of state encoding in the desired direction. Based on this premise, one state encoding strategy involves decomposing the FSM prior to performing the actual encoding. Variations on the decomposition strategy proposed in Chapter 3 can be used for that purpose.

In a common design scenario, one is required to redesign a sequential circuit for which a logic-level description is already available. In such a situation, it is necessary to extract the STG efficiently from the logic-level description. In Chapter 4, a procedure

for extracting symbolic information from logic-level descriptions of sequential circuits is presented. The novelty of this procedure lies in the fact that detection of some equivalent states and detection of edges that can be combined in a two-level representation of the STG is performed during the extraction process itself. As a result, this procedure avoids having to extract an intermediate representation in which the fanout of equivalent states and combinable edges are enumerated separately.

The work presented in Chapter 5 is concerned with the synthesis for testability of sequential circuits represented as interacting FSMs. In particular, procedures are presented for synthesizing interacting sequential circuits that are sequentially non-scan single-fault testable without any associated area penalty. In Chapter 6, a procedure is presented for synthesizing multiple-fault-testable sequential circuits.

Verifying the equivalence of two logic circuits is crucial since the validity of complex logic transformations must be checked at all times. The work presented in Chapter 7 is concerned with techniques for using general Binary Decision Diagram (BDD) representations for verifying combinational and sequential logic circuits. Unlike reduced, ordered BDDs, general BDDs are not canonical but are much more compact.

Chapter 8 concludes the dissertation. Most of the ideas presented in this dissertation have been implemented in the sequential synthesis system FLAMES. Implementation details of FLAMES are included as Appendix A.

Chapter 2

Basic Definitions and Concepts

Most of the terminology used in this dissertation is standard and in common use in the synthesis and testing communities [69, 17, 19]. This chapter is devoted to the definition of the nontrivial terminology and an elucidation of some of the basic concepts.

2.1 Definition of Terminology

2.1.1 Two-Valued Logic

A **binary variable** is a symbol representing a single coordinate of the Boolean space (*e.g.* a). A **literal** is a variable or its negation (*e.g.* a or $\overline{a}$). A **cube** is a set C of literals such that $x \in C$ implies $\overline{x} \notin C$ (*e.g.*, $\{a, b, \overline{c}\}$ is a cube, and $\{a, \overline{a}\}$ is not a cube). A cube (sometimes called a **product term**) represents the conjunction, *i.e.* the Boolean product of its literals. The trivial cubes, written 0 and 1, represent the Boolean functions 0 and 1 respectively. An **expression** (also called a **sum-of-products**) is the disjunction, *i.e.* a Boolean sum, f, of cubes. For example, $\{\{a\}, \{b, \overline{c}\}\}$ is an expression consisting of the two cubes $\{a\}$ and $\{b, \overline{c}\}$.

A cube may also be written as a bit vector on a set of variables with each bit position representing a distinct variable. The values taken by each bit can be 1, 0 or 2 (don't-care), signifying the true form, negated form and non-existence respectively of the variable corresponding to that position. A **minterm** is a cube with only 0 and 1 entries. Cubes can be classified based on the number of 2 entries in the cube. A cube with k entries or bits which take the value 2 is called a **k-cube**. A minterm thus is a **0-cube**. A cube

1–00	(inp1, inp3)	out3	100
111–	(inp1, inp3)	out3	011
101–	(inp1, inp2)	out2	110
1–01	(inp1, inp3)	out1	101
0–––	(inp1)	out1	111

(a)

1–00	101	001	100
111–	101	001	011
101–	110	010	110
1–01	101	100	101
0–––	100	100	111

(b)

Figure 2.1: An example of a symbolic cover and its one-hot coded representation

c_1 is said to **cover (contain)** another cube c_2, if c_1 evaluates to 1 for every minterm for which c_2 evaluates to 1. A **super-cube** of a set of cubes, c_i, is defined as the smallest cube containing all the minterms contained in c_i. The **on-set** of a function f is the set of minterms for which the function evaluates to 1, the **off-set** of f is the set of minterms for which f evaluates to 0, and the **don't-care set** or the **DC-set** is the set of minterms for which the value of the function is unspecified. An **implicant** of f is a cube that does not contain any minterm in the off-set of f. A **prime-implicant** of f is an implicant which is not contained by any other implicant of f.

2.1.2 Multi-Valued Logic

In general, a logic function may have **symbolic** (also known as **multiple-valued**) input or output variables in addition to binary variables. Like binary variables, a symbolic variable also represents a single coordinate, with the difference that a symbolic variable can take a subset of values from a set, say P_i, that has a cardinality greater than two. Let X_i be a symbolic input variable for the function f, and let S_i be a subset of P_i. Then ${X_i}^{S_i}$ represents the Boolean function

$${X_i}^{S_i} = \begin{cases} 0 & \text{if } X_i \notin S_i \\ 1 & \text{if } X_i \in S_i \end{cases}$$

${X_i}^{S_i}$ is called a **literal** of the variable X_i. The definition of a **cube (product-term)** remains unchanged; it is the Boolean product of literals. A minterm or 0-cube is now defined as the cube in which all variables take only a single value. A cube **covers (contains)** a minterm if it evaluates to 1 for that minterm. A cube c_1 covers (contains) another cube c_2 if c_1 evaluates to 1 for all the minterms for which c_2 evaluates to 1. When an output variable, say Y_j, is symbolic it implies that Y_j can take a value from a set P_j, of values when the input is some minterm. A function in which some variables are symbolic is known as a **symbolic**

cover. An example of a symbolic cover with one symbolic input and one symbolic output is shown in Figure 2.1(a). In the example, the symbolic input variable takes values from the set $\{inp1, inp2, inp3\}$ while the symbolic output variable takes a value from the set $\{out1, out2, out3\}$. A convenient method for representing a symbolic variable that can take values from a set of cardinality n is to use an n-bit vector to depict a literal of that variable such that each position in the vector corresponds to a specific element of the set. A 1 in a position in the vector signifies the presence of an element in the literal while a 0 signifies the absence. This method of representation is commonly known as **one-hot.** An example of a one-hot representation for the symbolic cover of Figure 2.1(a) is shown in Figure 2.1(b).

2.1.3 Finite Automata

A Finite-State Machine is represented by its **State Transition Graph** (STG) or equivalently, by its **State Transition Table** (STT). A STG is denoted by *G(V, E, W(E))*, where V is the set of vertices corresponding to the set of states S, where $||S||$ is the cardinality of the set of states of the FSM, E is the set of edges such that an edge (v_i, v_j) joins v_i to v_j if there is a primary input minterm that causes the FSM to evolve from state v_i to state v_j, and *W(E)* is a set of labels attached to each edge, each label carrying the information of the value of the input that caused the transition and the values of the primary outputs corresponding to that transition. The input combination and present-state corresponding to an edge, e, or a set of edges is (i, s), where i and s are cubes. The **fanin** of a state, q is a set of edges and is denoted $fanin(q)$. The **fanout** of a state q is denoted $fanout(q)$. The output and the fanout state of an edge $(i, s) \in E$ are $o((i, s))$ and $n((i, s)) \in S$ respectively. Given N_i inputs to a machine, 2^{N_i} edges with minterm input labels fan out from each state. A STG where the next-state and output labels for every possible transition from every state are defined corresponds to a **completely specified machine.** An **incompletely specified machine** is one where at least one transition edge from some state is not specified.

A STT is a tabular representation of the FSM. Each row of the table corresponds to a single edge in the STG. Conventionally, the left most columns in the table correspond to the primary inputs and the right most columns to the primary outputs. The column following the primary inputs is the present-state column and the column following that is the next-state column.

In all of the work reported here, a **starting** or **initial state** (also called the **reset state**) is assumed to exist for a machine. Given a logic-level description of a sequential machine with N_b flip-flops, 2^{N_b} possible states exist in the machine. A state which can be reached from the reset state via some input vector sequence is called a **valid state** (or a **reachable state**). The corresponding input vector sequence is called the **justification sequence** for that state. A state for which no justification sequence exists is called an **invalid state** (or an **unreachable state**). Two states a and b are equivalent if the output sequence produced starting from a is the same as the output sequence produced starting from b for any input sequence.

A **differentiating sequence** for a pair of states q_1, $q_2 \in Q$ of a machine M is a sequence (or string) of input vectors such that the last vector produces different outputs when the sequence is applied to M, when M is initially in q_1 or when M is initially in q_2. Two states q_1, q_2 in a completely specified machine M are **equivalent** (written as $q_1 \equiv q_2$), if they do not possess a differentiating sequence. If two states q_1, q_2 in an incompletely specified machine M do not possess a differentiating sequence, they are said to be **compatible** (written as $q_1 \cong q_2$).

A State Transition Graph G_1 is said to be **isomorphic** to another State Transition Graph G_2 if and only if they are identical except for a renaming of states.

2.1.4 Testing

Testing is concerned with the detection of defects (faults) on a chip. **Test generation** is used to obtain a compact set of inputs that can be applied to the chip to detect the presence of defects. Unless otherwise mentioned, the fault model assumed is the **stuck-at** model. In the stuck-at fault-model, a defect forces one or more inputs/outputs of a gate in the circuit to constant zero or constant one. If a gate output or primary input has multiple fanout, each fanout can be stuck-at a constant, independent of the other fanouts. In the **single-fault** model, only one wire is affected at a time. In the **multiple-fault** model, multiple wires are affected at the same time.

There are two kinds of untestable faults in a circuit. A fault is **combinationally testable** if there exists an input that can propagate the effect of the fault to the outputs of the combinational logic. Otherwise, the fault is said to be **combinationally redundant** or **untestable**. A finite-state machine is assumed to be implemented by combinational logic

and feedback registers. Tests are generated for faults in the combinational logic part. A fault is said to be **sequentially redundant** or **untestable** if the fault is combinationally testable and the effect of the fault cannot be propagated to a primary output by the application of a primary input sequence.

An edge in the STG of a machine is said to be **corrupted** by a fault if either the fanout state or output label of this edge is changed because of the existence of the fault. A path in the STG is said to be corrupted if at least one edge in the path has been corrupted.

If a fault is propagated to the next-state lines of a machine, it results in a faulty next-state, rather than a fault-free (or true) next-state. The fault is deemed to have produced a **faulty/fault-free state pair.**

To detect a fault in a sequential machine, the machine has to be placed in a state which can excite the fault and then the fault effect must be propagated to a primary output. These tasks are called **state justification** and **fault excitation-and-propagation**, respectively.

A primitive gate in a logic network is called **prime** if none of its inputs can be removed without causing the resulting circuit to be functionally different. The gate itself is **irredundant** if its removal causes the resulting circuit to be functionally different. A gate-level circuit is said to be **prime** if all the gates are prime and **irredundant** if all the gates are irredundant. It can be shown that a gate-level circuit is prime and irredundant if and only if it is 100% testable for all single stuck-at faults [10].

2.2 A Review of the Basic Concepts in State Encoding

A review of previous work in state encoding was presented in Chapter 1. To aid the understanding of the material in Chapter 3, a more detailed review of recent work in that area is presented here.

A finite-state machine is a logic function with symbolic inputs and outputs, with the additional caveat that the same symbols appear in the input and output planes. Therefore, state encoding considered in its entirety is an input-output encoding problem and it cannot be approximated by either a purely input encoding problem or a purely output encoding problem.

The main difficulty in state encoding lies in the need to predict the effects of the complicated logic optimization steps which follow the encoding process. A major con-

```
0 s1  s2 1
1 s1  s4 0
0 s2  s2 1
1 s2  s1 1
0 s3  s3 0
1 s3  s4 0
0 s4  s2 1
1 s4  s1 1
```

(a)

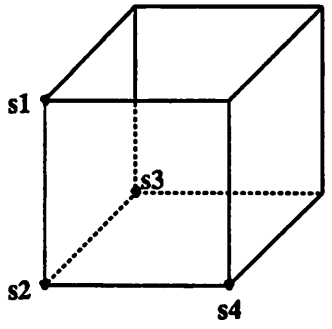
```
0 (s1, s2, s4)  s2 1
1 (s2, s4)      s1 1
1 (s1, s3)      s4 0
0 (s3)          s3 0
```

(b)

Figure 2.2: A symbolic tabular representation of a finite-state machine

```
s1 = 001
s2 = 000
s3 = 011
s4 = 100
```

(a)



(b)

Figure 2.3: Codes satisfying input constraints

tribution in that regard was that of De Micheli *et al* [86] in which a new paradigm was proposed for state encoding. It was suggested that state encoding be viewed as a two-step process. In the first step, a tabular representation of the FSM is optimized at the symbolic level. This optimization step generates constraints on the relationships between codes for different states. In the second step, states are encoded in such a way the constraints are satisfied. Satisfaction of the constraints guarantees that the optimizations at the symbolic-level will be preserved in the Boolean domain. The accuracy with which optimization at the symbolic-level can mimic optimizations in the Boolean domain is then a major issue.

2.2.1 Input Encoding Targeting Two-Level Logic

The initial effort in state encoding using the two-step paradigm was targeted toward obtaining optimal two-level implementations of encoded FSMs. A technique (implemented in the program ESPRESSO-MV [26] by Rudell *et al*) for minimizing two-level covers with symbolic variables is known as **symbolic** or **multiple-valued minimization** [94].

```
0 -1- 011 0
1 --1 100 0
1 --0 001 1
0 -0- 000 1
```

Figure 2.4: Two-level implementation of encoded finite-state machine

An example tabular representation of a FSM is shown in Figure 2.2(a). A symbolic minimization of this FSM leads to the cover shown in Figure 2.2(b). It can be seen that the minimized cover is output-disjoint and all the reduction in the cardinality of the symbolic cover is due to relationships in the input part, *i.e.* due to the fact that some states fan out to the same next state for certain primary inputs.

The goal in deriving constraints from the minimized symbolic cover is to encode the states in such a way that the cardinality of the resulting two-level Boolean implementation is no greater than the cardinality of the minimized symbolic cover. A sufficient condition that ensures the preservation of the cardinality of the symbolic cover [1] during the transition from the symbolic to the Boolean domain is to ensure that each multiple-valued input literal in the minimized symbolic cover translates into a single cube in the Boolean domain. In other words, given a multiple-valued literal, the states present in it should form a face in the Boolean n-space in such a way that the face does not include the states absent from the same multiple-valued literal. Such constraints are called **face-embedding** or **input** constraints, and the process of obtaining these constraints and satisfying them is known as **input encoding.**

Codes satisfying the face-embedding constraints implied by the minimized symbolic cover of Figure 2.2(b) are shown in Figure 2.3(a). As can be seen in the figure, three binary variables are required to satisfy the face-embedding constraints. Figure 2.3(b) shows the location of these codes in the Boolean n-space and shows that they satisfy the face-embedding constraints. The cover obtained after substitution of the state codes in the symbolic cover and a two-level Boolean minimization is shown in Figure 2.4. While two-level symbolic minimization by definition [94] can be used to explore all optimization possibilities in the input plane, it is intrinsically incapable of optimizations involving a sharing of logic among different next-state symbols. Since the work by De Micheli *et al* [86] obtained the encoding constraints using two-level symbolic minimization, it effectively approximated the

[1] minimized using ESPRESSO-MV

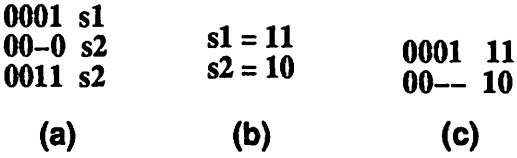


Figure 2.5: Example of the effects of dominance relationships

state encoding problem as one of input encoding.

2.2.2 Output Encoding Targeting Two-Level Logic

There are two types of interactions in the next-state plane that lead to cardinality reductions in the minimized two-level Boolean cover. The first type involves the bitwise dominance of the code of one state by the code of another. A binary vector v_1 is said to dominate another binary vector v_2 if in every bit that v_2 has a 1, $v1$ also has a 1. An example is shown in Figure 2.5. A cover with a symbolic output part is shown in Figure 2.5(a) and the codes for the symbols s_1 and s_2 are shown in Figure 2.5(b). Since the code for the symbol s_1 dominates the code for s_2, the input cubes asserting s_1 can be used as don't-cares in order to reduce the cardinality of the cover for s_2, as shown in Figure 2.5(c). The use of dominance constraints in state encoding was explored by De Micheli [85]. In that work, the problem of exploiting the dominance constraints was reduced to one of heuristically ordering the states. The ordering of states dictated the dominance relationships between states. Satisfying these dominance relationships (which should not conflict) results in some reduction of the overall cover cardinality. The basic intuition used in the ordering was that a next-state with a large cover be optimized under as large a don't-care set as possible.

A limitation of this approach is that minimum cardinality cannot be guaranteed because all possible dominance relations are not explored, nor is an optimum set selected. A more basic shortcoming is that dominance relations are not the only kind of relationships between symbolic values that can be exploited. After a symbolic cover has been encoded, it represents a multiple-output logic function and minimizing a multiple-output function entails exploiting other sharing relationships rather than just the dominance relationship.

The second type of interaction in the next-state field has to do with disjunctive relationships between state codes. A disjunction of two Boolean vectors is the bitwise-or of the two vectors. If the code of a state, say s_1 is equivalent to or is dominated by the disjunction of the state codes of two or more state variables, then the cardinality of the cover

	s1 = 00 s2 = 01 s3 = 11	s1 = 11 s2 = 01 s3 = 10
	Codes	Codes
	1-1 00 1	
101 s1 1	100 01 1	10- 01 1
100 s2 1	111 11 1	1-1 10 1
111 s3 1	Minimized Cover	Minimized Cover
(a)	(b)	(c)

Figure 2.6: Example of the effects of disjunctive relationships

for the next-state s_1 can be reduced by using the on-sets of the covers for the codes of the next-states involved in the disjunction. As an example, consider the cover in Figure 2.6(a) with symbolic values in the output part. In Figure 2.6(b), the code for the symbol s_1 is the disjunction of the codes for symbols s_2 and s_3. The cardinality of the cover of the on-set for the symbol s_1 is effectively zero. When the disjunctive relationship is absent as in the codes shown in Figure 2.6(c), the cardinality of the cover of the on-set of the symbol s_1 is one. Note that the reduction in the cardinality in Figure 2.6(b) is for the dominating state and not the dominated state, and is therefore not due to the dominance relationships described previously.

A method for exploring all possible dominance and disjunctive relationships implicitly but exhaustively was proposed by Devadas and Newton [41]. The basic approach involved a modification of the prime-implicant generation and covering procedures fundamental to two-level Boolean minimization [90, 83]. The definition of the notion of Generalized Prime Implicants (GPIs) was the key contribution in that regard.

Generalized Prime Implicants

GPIs correspond to a weaker form of primality than primes in the Boolean domain. The basic idea in the use of GPIs is to mimic prime-implicant generation in the Boolean case. Consider the two minterms in Figure 2.7(a). These minterms have three two-valued input variables and three two valued output variables. Since these two minterms (or, 0-cubes) are distance-1 from each other in the input part, they can be merged together to form a 1-cube, with the output part of the 1-cube being the bitwise conjunction of the output parts of the individual 0-cubes. Now, if the output part had been symbolic for the two 0-cubes as shown in Figure 2.7(b) (as would be the case for an output-encoding problem), the two

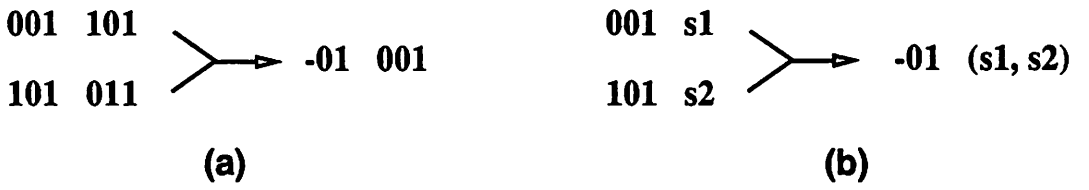


Figure 2.7: Merging cubes to form larger cubes

0-cubes would still be allowed to merge, with the output part of the 1-cube being the union of the output parts of the 0-cubes. It is implicitly understood that once the symbols are encoded, the symbolic-output part that contains more than one symbol becomes the bitwise conjunction of the codes of the symbols present in it. For example, in the 1-cube shown in Figure 2.7(b), if the symbols s_1 and s_2 were given the codes 101 and 011, respectively, the output part of the 1-cube with the symbols encoded would be 001.

Generation of all prime-implicants in the Boolean domain can be viewed conceptually as the merging of k-cubes to form $(k+1)$-cubes as shown in Figure 2.7(a), until no new cubes are generated. The generation of $(k+1)$-cubes is accompanied by the removal from the list of candidate primes of those k-cubes that are covered by a $(k+1)$-cube. When two k-cubes are merged, the output part of the $(k+1)$-cube cannot dominate the output parts of the k-cubes it was derived from since it is the conjunction of the output parts of the k-cubes. Therefore, a $(k+1)$-cube can cause the removal of a k-cube only if the input part of the $(k+1)$-cube covers the input part of the k-cube and the output part of the k-cube is the same as the output part of the $(k+1)$-cube.

The generation of GPIs follows exactly along the same lines. $(k+1)$-cubes with symbolic outputs are generated by merging k-cubes as in Figure 2.7(b) until no new primes can be generated. A $(k+1)$-cube can remove a k-cube only if the input part of the $(k+1)$-cube covers the input part of the k-cube and the symbolic output parts of the two cubes are the same. The rule for removal of cubes has to be modified for the input-output encoding problem in state encoding (*cf* Section 2.2.3). Given the procedure for generating GPIs, the reason why GPIs correspond to a weaker form of primality than prime-implicants in the Boolean domain is easy to see. In the Boolean domain, if the conjunction of the output parts of a set of k-cubes is all zeroes, then no $(k+1)$-cube is generated from these k-cubes. In the presence of symbolic outputs, however, since the codes for the symbols are not known *a priori*, all possible $(k+1)$-cubes have to be generated including those that would have

all zero output parts after the encoding of the symbols. Of course, if a cube contains all possible symbols in the output part, it can be discarded.

Any selection of GPIs implies output constraints that have to be satisfied if the encoded cover is to have the same cardinality as the selected set of GPIs. Unlike the case of input encoding, an arbitrary selection of GPIs does not necessarily correspond to satisfiable constraints. If the constraints are satisfiable, the selection of GPIs is said to be encodable. However, a selection may not be encodable because of mutual conflicts between the constraints corresponding to different GPIs in the selection.

Given a selection of GPIs, the basic constraint implied for each symbol (in this case for s_i) is of the following type:

$$e(s_i) = \bigcup_k \bigcap_j e(s_{j,k}) \tag{2.1}$$

where $e(s)$ is the encoding for symbol s, k ranges over the GPIs that contain the symbol s_i in the output part, j ranges over the symbols in the output part of each such GPI, $\bigcap$ corresponds to bitwise conjunction and $\bigcup$ to bitwise disjunction. For example, if two cubes, one with the output part (s_1, s_2, s_3) and another with the output part (s_1, s_4) were the only cubes in the selected set containing s_1 in the output part, the constraint for the symbol s_1 would be $e(s_1) = e(s_1) \cap e(s_2) \cap e(s_3) \bigcup e(s_1) \cap e(s_4)$. This basic constraint encapsulates the effects of both dominance and disjunctive relations. Given an arbitrary set of constraints, dominance and disjunctive relations between the encodings must be derived so that satisfying them satisfies the constraints. Dominance and disjunctive relationships may conflict across a set of constraints. Consider the constraint $e(s_1) = e(s_1) \cap e(s_2) \bigcup e(s_1) \cap e(s_3)$ for $e(s_1)$. There are three alternatives for satisfying this constraint:

1. $e(s_1) \subset e(s_2)$
2. $e(s_1) \subset e(s_3)$
3. $e(s_1) \subseteq e(s_2) \cup e(s_3)$

If one chooses to use the relationship $e(s_1) \subset e(s_2)$ and if another constraint requires that $e(s_2) \subset e(s_1)$ be used for satisfying it, then the selection of GPIs is not encodable.

If the length of the encoding is known *a priori*, the problem of checking for satisfiability of the constraints reduces to a Boolean satisfiability problem. For the case when the encoding length is not known *a priori*, a constructive procedure for encodability checking

was proposed [41]. In that procedure, one begins with a graph with no edges and whose nodes correspond to the symbols to be encoded. Further, if a disjunctive relation is required in which each element in the disjunction is the conjunction of multiple symbols (*e.g.* $e(s_1) \subseteq e(s_2) \cap e(s_3) \bigcup e(s_4)$), nodes (called conjunctive nodes) are also added for each conjunctive element in the disjunction. Edges are added to the graph whenever it is decided to apply a certain relationship to satisfy a constraint. For example, a directed edge is added from node s_1 to node s_2 if the satisfaction of some constraint requires that the code for s_1 dominate the code for s_2. Similarly, if the code for the symbol s_1 is required to be equal to the disjunction of the codes of the states $\{q_i\}$, a directed edge is drawn from s_1 to each state in $\{q_i\}$ and this set of edges is given a unique label. If the code for the symbol s_1 is required to be dominated by the disjunction of the codes of the states $\{q_i\}$, an undirected edge is drawn from s_1 to each state in $\{q_i\}$ and this set of edges is given a unique label. The states $\{q_i\}$ in the last two cases are called siblings and the state s_1 is called the parent. Edges are similarly added in the graph for conjunctive nodes. The graph being constructed is required to satisfy certain properties at every stage of the construction. For example:

- No edge can be added that creates a directed cycle.
- The siblings in a disjunctive relationship cannot have directed edges between them.
- The same set of siblings cannot have two different parent nodes.
- The parent node in a disjunctive dominance relationship cannot dominate all its siblings.
- The parent node in a disjunctive equality relationship cannot dominate a node that dominates all the siblings.

If such a graph can be built with the edges corresponding each constraint added to it, the selection of GPIs is encodable.

The problem of obtaining the minimum two-level representation of a function can be reduced to one of finding the minimum number of prime-implicants covering all the minterms. Since the number of primes is much lower than the total number of cubes, this approach accompanied by efficient procedures for generating the primes leads to an effective pruning of the search space in Boolean minimization. GPIs have a similar effect on the search space in output encoding. It can be easily shown that the minimum-cardinality

encodable symbolic cover can be made up of only GPIs. Thus, if one selects a minimum set of GPIs that cover all the minterms and all the constraints corresponding to which are satisfiable, one is guaranteed a minimum solution of the output encoding problem.

2.2.3 Input-Output Encoding Targeting Two-Level Logic

The state encoding problem is not merely a simple sum of the input and output encoding problems. Since the same symbols appear in the present-state and the next-state fields, state encoding requires the solution of input and output encoding problems that are tightly coupled with each other.

An approach for combining the input and output encoding problems was suggested by Devadas and Newton [41]. The approach is basically an extension of the approach for exact output-encoding (*cf* Section 2.2.2) suggested in the same work.

In the input encoding problem, any set of input constraints is always satisfiable by some encoding. The basic reason why the combined input-output encoding problem is more difficult is that one can now no longer assume that any set of input constraints is always satisfiable. The input constraints now have to be satisfied by the same encoding that satisfies the output constraints.

The combination of the input and output encoding problems involves two basic alterations to the output-encoding procedure. Firstly, the definition of GPIs is relaxed in that a $(k+1)$-cube is allowed to remove a k-cube from the list of candidate primes only if it covers the k-cube, the symbolic-output parts of the two cubes are identical *and* the symbolic-input parts of the two cubes are either the same or the symbolic-input part of the $(k+1)$-cube contains all symbols. This allows all possible combinations of symbolic-input literals and symbolic-output tags to be considered during the selection of the minimum cover of GPIs. On the other hand, the relaxation in the definition of GPIs also increases the total number of GPIs considerably.

The second alteration to the output encoding procedure is to add constraints that check for cases that correspond to a conflict between the satisfaction of an input constraint and an output constraint. A simple example that illustrates a conflict between input and output constraints is the following. Consider three states s_1, s_2 and s_3. Assume that the output constraints *require* that $e(s_1)$ dominates $e(s_2)$ and that $e(s_2)$ dominates $e(s_3)$. Now assume that the symbolic-input literal in one of the selected GPIs has states s_1 and s_3 in it

and does not have state s_2. In order to satisfy this input constraint, it is required to encode states s_1 and s_3 in such a way that the super-cube of $e(s_1)$ and $e(s_3)$ does not contain $e(s_2)$. But that is impossible because of the two dominance relationships involving s_1, s_2 and s_3. The dominance relationships imply that if $e(s_3)$ is different from $e(s_2)$ in a certain bit, it is also different from $e(s_1)$ in the same bit. Therefore, that bit becomes a don't-care in the super-cube. Similarly, if $e(s_1)$ is different from $e(s_2)$ in a certain bit, it is also different from $e(s_3)$ in the same bit. Again, that bit becomes a don't-care in the super-cube. Therefore, the input constraint can never be satisfied in conjunction with the two dominance constraints. Similarly, the input constraint corresponding to the presence of all the siblings of a disjunctive relationship in a symbolic-input literal and the absence of the parent can never be satisfied. The last constraint must be satisfied not only for the disjunctive relationships implied directly, but also for the disjunctive relationships implied by a propagation of other relationships. For example, the two relationships $e(s_1) = e(s_2) \cup e(s_3)$ and $e(s_3) \subset e(s_5)$ imply the disjunctive relationship $e(s_1) \subseteq e(s_2) \cup e(s_5)$. Similar constraints can also be derived for disjunctive relationships that involve a disjunction of conjunctions (*cf* Section 2.2.2) of state codes.

If the desired encoding length were known *a priori*, then as in the output encoding case, checking for encodability in the input-output encoding case can also be reduced to a Boolean satisfiability problem. In that case, input constraints would be described by equations that require the code for each state absent from a symbolic-input literal to be different from the codes for every state present in the literal in the bit. If the encoding length is not known *a priori*, the same constructive procedure used in output encoding can be used.

In the case of input-output encoding, as for exact output encoding, the minimum cardinality symbolic cover need consist of only GPIs. Therefore, the input-output encoding problem (hence the state encoding problem) targeting two-level logic can be solved exactly by choosing the minimum cardinality encodable selection of GPIs.

Chapter 3

Algorithms for FSM Decomposition

3.1 Introduction

FSM decomposition is concerned with the implementation of a FSM as a set of smaller interacting submachines. Such an implementation is desirable for a number of reasons. A partitioned sequential circuit usually leads to improved performance as a result of a reduction in the longest path between latch inputs and outputs. This fact is particularly true when the individual submachines are implemented as PLAs. It appears that the primary interest in using decomposition tools in industry stems from a need to improve the performance of FSM controllers, which often dictates the required duration of the system clock. FSM decomposition can be applied directly when Programmable Gate Arrays (PGAs) or Programmable Logic Devices (PLDs) are the target technology. Such technologies are characterized by I/O or gate-limited blocks of logic and latches into which the circuit must be mapped. In many cases, it is desirable for reasons of clock-skew minimization or simplifying the layout to distribute the control logic for a data path in such a manner that the portions of the data path and control that interact closely are placed next to each other. FSM decomposition can also be used for this purpose. Partitioning of the logic implementing the FSM could result in simplified layout constraints resulting in smaller chip area. In PLA-based FSMs, decomposition has the effect of partitioning the PLA that implements the original FSM into smaller interacting PLAs that implement the individual

submachines. In such situations, an area reduction can be attributed to PLA partitioning. Finally, it is not computationally feasible for current multilevel logic minimizers (*e.g.* MIS-II [15]) to search all possible area minimal solutions. In such cases, an initially-decomposed FSM could correspond to a superior starting point for multilevel logic minimization.

It should be noted that performing the decomposition at the State Transition Graph (STG) level where states are still symbolic, as against partitioning the logic *after* an encoding of states in the original machine where a subsequent logic minimization has already been performed, makes it possible to search a larger solution space for good decompositions. The work presented in this chapter addresses the problem of the decomposition of sequential machines into smaller interacting submachines, so as to optimize for area *and* achieve the other desirable features like improved performance of the resulting implementation.

Previous approaches (*e.g.* [40, 60]) to FSM decomposition used the number of states and edges in the resulting submachines as their cost function. Given that the logic implementation of a FSM is derived from its STG specification after state assignment and intensive logic optimization, this cost function does not reflect the true complexity of the eventual logic-level implementation and is, on occasion, far from accurate. In addition, previous approaches are mainly heuristic in nature and offer limited guarantees as to the quality of the final solution.

The use of partition theory in state encoding and FSM decomposition was introduced by Hartmanis and Stearns [60]. The essential point presented in this chapter is to show that state partitioning can be easily tied in with recent results in state encoding [86, 41] in order to target decompositions with logic-level optimality. The contributions of the work presented here include:

1. A formulation of the optimum two-way decomposition problem targeting two-level logic as one of *symbolic-output partitioning.* The cost function associated with this formulation is the total number of product terms in the minimized symbolic representations of the submachines. This cost function is much closer to the final logic-level complexity than the number of states/edges in the decomposition.

2. The development of an exact solution, under the formulation chosen above, to the decomposition problem via a method of prime implicant generation and constrained covering. The fact that the problem of two-way FSM decomposition is easier than that of exact state assignment [41] is exploited here.

3. Previous work in decomposition targeted specific decomposition topologies. A consequence of the exact decomposition procedure is that two-way or multi-way, parallel, cascade, general or arbitrary decomposition topologies can be targeted, simply by changing the constraints in the covering step.

4. The development of a heuristic optimization strategy applicable to large sized problems. The heuristic procedure consists of an iterative optimization involving *symbolic implicant expansion and reduction*, modified from two-level Boolean minimizers. Reduction and expansion are performed on functions with symbolic, rather than binary-valued outputs. Many different expansion/reduction heuristics have been implemented and evaluated under this global strategy.

Basic definitions relevant to this chapter are presented in Section 3.2. In Section 3.3, the decomposition problem is formulated as one of symbolic-output partitioning and an exact procedure is given to solve it. Subsequently, the correctness of the procedure is formally demonstrated. The extension of the exact decomposition procedure to arbitrary topologies is presented in Section 3.4. A heuristic expand-reduce procedure, viable for large size problems, is presented in Section 3.5. Section 3.6 briefly explains the relationship between FSM decomposition and state assignment. Experimental results on area and performance optimization are presented in Section 3.7.

3.2 Basic Definitions

A **partition** π on a set, S, is a collection of *disjoint* subsets whose union is S. The disjoint subsets are called the **groups** or **blocks** of π. Consider for example a set $S = \{s_1, s_2, s_3, s_4, s_5\}$. $\pi = \{(s_1, s_3), (s_2), (s_4, s_5)\}$ is said to be a partition on S. (s_1, s_3), (s_2) and (s_4, s_5) are said to be the blocks of π. Let $\pi_1 = \{a_i\}$ and $\pi_2 = \{b_j\}$, where a_i and b_j are blocks, be two partitions on S. The **product** of π_1 and π_2, denoted $\pi_1 \cdot \pi_2$, is defined as the partition $\pi_1 \cdot \pi_2 = \{a_i \cap b_j\}$. The **zero-partition** on S is the partition such that the cardinality of all groups is 1.

Given a State Transition Graph description of a desired terminal behavior, the essence of the decomposition problem is to find two or more machines which, when interconnected in a prescribed way, have that terminal behavior. The individual machines that make up the overall realization are referred to as **submachines**. Each submachine

```
0 s1 s3 1
1 s1 s1 0
0 s2 s4 0
1 s2 s1 1
0 s3 s2 0
1 s3 s4 0
0 s4 s2 1
1 s4 s3 1
```

Figure 3.1: A STT representation of a finite-state machine

corresponds to a partition on the set of states, S. All the states belonging to each block of the partition in a submachine are given the same code in that submachine. Therefore, there is no way of distinguishing between two states belonging to the same block in a submachine without recourse to information from other submachines. A block of states in a partition effectively corresponds to a state in the submachine associated with that partition. The **prototype machine** corresponds to the machine that was used to define the terminal behavior to be realized. The term **lumped machine** is used sometimes to denote an undecomposed implementation of the prototype machine. The machine that results as a consequence of the decomposition is called the **decomposed machine**. The functionality of the prototype machine is maintained in the decomposed machine if the partitions associated with the decomposition are such that their product is the zero-partition.

As an example, consider the FSM shown in Figure 3.1. Let the two partitions on S be $\pi_1 = \{(s_1, s_2), (s_3, s_4)\}$ and $\pi_2 = \{(s_1, s_3), (s_2, s_4)\}$. The product of these two partitions, $\pi_1 \cdot \pi_2$, is the zero-partition $\{(s_1), (s_2), (s_3), (s_4)\}$. What that means is that given one block of states from each of π_1 and π_2, the state corresponding to the prototype machine is uniquely identified. Now assume that the output logic is implemented only in the submachine corresponding to π_2. In that case, it can be verified from Figure 3.1 that in Submachine 1 (which only implements the next-state logic), there is no need for information from Submachine 2 since the transitions between the blocks in Submachine 1 are independent of the block that Submachine 2 is in. A partition like π_1 is called a **closed (or, preserved) partition**. If a closed partition can be found, it means that the prototype machine can be decomposed into a **cascade** of FSMs with the submachine corresponding to the closed partition being the head machine. A **parallel** decomposition exists when closed partitions can be found such that the product of these closed partitions is the zero-partition.

The concept of partitions can be generalized to **covers** [61, 60]. Covers differ from

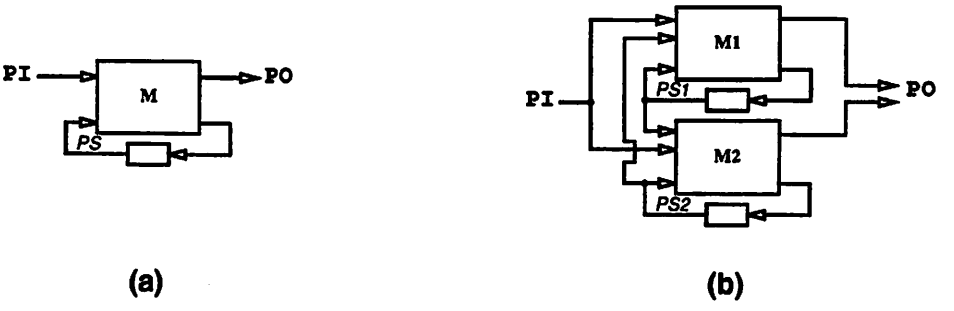


Figure 3.2: General decomposition topology

partitions in that blocks in the same cover are allowed to intersect. Allowing intersections between blocks in the same cover effectively corresponds to splitting states in the prototype machine without changing functionality and finding partitions on this new set of states. The use of state splitting allows decompositions and state encodings with closed partitions that would not exist otherwise.

3.3 Exact Procedure for Two-Way General Decomposition

General decompositions can have various topologies. The decomposition topology of Figure 3.2(b) is the one considered in this chapter. In Figure 3.2(b), the original machine M in Figure 3.2(a) has been decomposed into two submachines, M_1 and M_2, interconnected in the prescribed way. The output logic for the decomposed machine is distributed between the two submachines, unlike the formulation of [40] where a logic block external to the submachines was required to generate the primary outputs. The goal of this work is to provide exact or near-exact solutions to the problem of two-way general decomposition targeting two-level logic. An associated cost function close to the cost of the ultimate logic-level implementation is also presented. Subsequently, an algorithm for finding a decomposition that is optimum under the chosen formulation, is given.

3.3.1 The Cost Function

The cost function for a general decomposition can vary depending on the eventual targeted implementation. Here, one is concerned with two-level implementations. The cost function used allows the decomposition of the prototype machine into submachines such that the sum of the areas of the two-level implementations of each submachine, after the

states in the submachine have been assigned codes, is less than or close to the area of the two-level implementation of the prototype machine after state assignment. The area of the two-level implementation of each submachine is *always* less than the area of the two-level implementation of the prototype machine. It is also found that the cost, measured in terms of the area of the standard-cell layout of the logic, of the multilevel implementation of the decomposed machine obtained using this cost function is *almost always* less than the cost of the multilevel implementation of the prototype machine. This implies that an optimal decomposition targeting a two-level implementation is an acceptable decomposition for the multilevel case.

Consider the submachines in Figure 3.2(b). Let the number of product terms in the prototype machine, M, after one-hot coding and two-level Boolean minimization be P. Let the number of product terms in the submachines M_1 and M_2 after one-hot coding and two-level Boolean minimization be P_1 and P_2, respectively. A decomposition is deemed to be optimum (optimal) in this formulation if $P_1 + P_2$ is minimum (minimal). In the case when no good decomposition can be found, $P_1 + P_2 = P$.

The area and performance of a PLA are closely related to each other. Since the two-level area of each submachine obtained using this cost function is always less than the two-level area of the prototype machine, and because the critical path of the decomposed machine in the topology of Figure 3.2(b) is equivalent to the critical path of the larger of the two submachines, the critical path of the decomposed machine in Figure 3.2(b) will be smaller than the critical path of the prototype machine in the *two-level implementation.* To optimize the critical path of the decomposed machine, the larger of the two submachines should have as small an area as possible. One way of achieving this is to keep the sizes of the submachines similar while minimizing the overall area. Thus, a modified cost function of the form $P_1 + P_2 + \alpha\|P_1 - P_2\|$, in which an additional cost is attached to a difference in the areas of the PLAs, characterizes the optimality of the decomposition with respect to timing also. α is some empirically determined constant.

3.3.2 Decomposition, Factorization and Partitioning

The formulation of the optimum decomposition problem is described in the sequel. The first step is to obtain the initial symbolic covers representing the two submachines. Given the initial State Transition Table (STT) (or equivalently, the STG) of machine M

with N states, s_1, .. s_N, a new symbolic function L is constructed as follows: The present-state (PS) field in the STT is replaced by an N-valued variable represented by an N-bit vector in which each bit is associated with the presence or the absence of a state. The next-state (NS) field in M is *split* into *two* symbolic variables, *i.e.* s_1 is split into symbolic variables sa_1 and sb_1, s_2 is split into symbolic variables sa_2 and sb_2 and so on. The primary input (PI) and primary output (PO) fields are unchanged. An example transformation is shown below. The example has one primary input, two primary outputs and three states. In the transformed symbolic cover, say L, the first column represents the primary input as before, the second, third and fourth columns correspond to the three-valued variable representing the present state. The fifth and sixth columns represent the partitioned next-state field, and the seventh and eighth columns represent the primary outputs.

$$
\begin{array}{lcl}
0\ s_1\ \ s_2\ 10 & \longrightarrow & 0\ 100\ \ sa_2\ sb_2\ 10 \\
1\ s_1\ \ s_3\ 01 & \longrightarrow & 1\ 100\ \ sa_3\ sb_3\ 01 \\
0\ s_2\ \ s_1\ 11 & \longrightarrow & 0\ 010\ \ sa_1\ sb_1\ 11 \\
1\ s_2\ \ s_3\ 10 & \longrightarrow & 1\ 010\ \ sa_3\ sb_3\ 10 \\
0\ s_3\ \ s_2\ 00 & \longrightarrow & 0\ 001\ \ sa_2\ sb_2\ 00 \\
1\ s_3\ \ s_1\ 01 & \longrightarrow & 1\ 001\ \ sa_1\ sb_1\ 01
\end{array}
$$

The next step is to obtain the symbolic covers for the submachines. The symbolic cover, L_a, for **Submachine a** is obtained in the following manner. For each row in L, there is a corresponding row in L_a that has the same PI and PS fields as in L. The next-state field in a row in L_a consists of the symbol sa_i from the partitioned next-state field in the corresponding row in L. The symbolic cover, L_b, for **Submachine b** is obtained in a similar manner, with the next-state field in a row of L_b being the symbol sb_i from the corresponding row in L. The primary outputs in L are partitioned between L_a and L_b. By *partitioning* of the primary outputs, it is implied that some of the primary outputs are now asserted in the symbolic cover for **Submachine a** while the remaining primary outputs are asserted in the symbolic cover for **Submachine b**. In the case when the primary outputs are also symbolic and need to be encoded, they can also be symbolically partitioned and treated in the same manner as the next-state lines. In the example shown below, all the primary outputs in L are asserted in L_a (shown on the left) and no primary outputs are asserted in L_b (shown on the right).

0 100 sa_2 10	0 100 sb_2
1 100 sa_3 01	1 100 sb_3
0 010 sa_1 11	0 010 sb_1
1 010 sa_3 10	1 010 sb_3
0 001 sa_2 00	0 001 sb_2
1 001 sa_1 01	1 001 sb_1
L_a	L_b

L_a and L_b are identical to each other and to L, with the difference that some of the fields in the next-state plane and output plane of L are asserted in L_a while the remaining are asserted in L_b. L_a and L_b represent the initial symbolic covers of the two component submachines that M is to be decomposed into. The column in L_a with the sa_i's represents the next-state column for **Submachine a**, and the column in L_b with the sb_i's represents the next-state column for **Submachine b**. The operations of **Submachine a** and **Submachine b** are mutually dependent, with the communication between the submachines, as can be seen in Figure 3.2(b), via the present-state lines of each submachine. While the next-state field is partitioned in the above procedure, the present-state field, represented by the multiple-valued variable, is duplicated in L_a and L_b. As in L, the multiple-valued input variable in L_a and L_b also represents states in the *prototype machine*. Since the state of **Submachine a** and the state of **Submachine b** together uniquely determine the state of the overall machine M and since a knowledge of the state of the overall machine is sufficient to determine the states of **Submachine a** and **Submachine b**, the multiple-valued input variable in L_a and in L_b represents, implicitly, the present-state of **Submachine a** as well as of **Submachine b**.

The symbolic-outputs in L_a and L_b are now represented by means of a **one-hot code** (*cf* Section 3.2). Given the one-hot coded initial symbolic covers for the two submachines, the goal is to minimize the sum of the cardinalities of the two covers. To achieve this goal, one can use the degree of freedom that since each submachine knows its own present-state and the present-state of the other submachine, as long as two states of the prototype machine have been given different codes in *one* of the submachines, it is possible to distinguish between the two states in the decomposed machine. For example, states s_1 and s_2 can be given the same code in **Submachine a**, *i.e.* $\mathbf{e}(sa_1) = \mathbf{e}(sa_2)$, as

long as it is possible to distinguish between the two states by means of the codes given to them in the other submachine. Since the encoding is one-hot, two states, say sa_i and sa_j or sb_i and sb_j, either have the same code or the bitwise intersection of their codes is null. The symbolic output, in this case the next-state variable, is represented by sets, called **tags**, of next-state symbols that have been given the same code. Initially, when all states have different codes, the symbolic-output tag for a cube consists of only a single next-state symbol corresponding to the next-state asserted by the cube. When two states are given the same code, the symbolic-output tags containing these two states are replaced by tags that contain both next-state symbols. For example, when sa_i and sa_j are given the same code, each occurrence of the symbolic-output tags (sa_i) and (sa_j) is replaced by the tag $(sa_i\ sa_j)$. A similar replacement is carried out when three states are given the same codes, and so on.

Let the final cardinality of the minimized cover for **Submachine a** be P_a and that for **Submachine b** be P_b. Let the cardinality of the minimized cover for the prototype machine, with the states one-hot coded, be P. Consider for illustration the extreme case where all the sa_i are given the same code. To distinguish between the states of the prototype machine in this case, all the sb_i have to be assigned distinct codes. Effectively, **Submachine a** realizes the primary output logic of the prototype machine and **Submachine b** realizes the next-state logic. As a result, $P_a + P_b \geq P$. In the other extreme case, if all the sb_i are given the same code, then **Submachine b** is not required at all (*i.e.* $P_b = 0$) since **Submachine b** does not assert any primary outputs, but all the sa_i must be given different codes. Hence $P_a = P$.

Relationship to Partition Algebra

The problem, therefore, is to decide which states of the prototype machine should be given the same code and the submachine in which they should be given the same code. The states that are given the same code in a submachine belong to the same block in the *partition* (*cf* Section 3.2) corresponding to that submachine. This problem is identical to finding two or more partitions such that the sum of the cardinalities of the minimized symbolic covers of the submachines represented by these partitions is minimum and the product of the partitions is the zero-partition.

A large number of choices is available in the states that can be given the same

codes. In order to avoid enumerating all the possibilities, a formal procedure is required that searches all the possibilities exhaustively, but implicitly.

Relationship to Factorization

It was proposed in [40] that typical FSMs possess isomorphic or close to isomorphic subgraphs in their STGs. It was suggested that identifying such isomorphic subgraphs and implementing multiple instances of an isomorphic subgraph as a single separate submachine distinct from the parent machine corresponds to an effective decomposition. The effect of such a decomposition is to identify instances of similar functionality in the STG and to encapsulate them into a single submachine; in other words identify subgraphs such that it is advantageous to replace them with subroutine calls. This approach was called *factorization* in [40] because of its obvious similarity to extracting common factors in combinational logic. The ability to use subroutines has thus far been considered one of the major reasons why designers choose to use the microcode-style implementation for FSM controllers. The work in [40] shows that it is not necessary to resort to the microcode-style implementation in order to benefit from subroutine extraction.

It can be shown that factorization is equivalent to identifying specific types of partitions on the states in the prototype machine. In the ideal situation, the procedure for decomposition proposed in this chapter should automatically be able to identify such partitions if they translate into lower logic-level complexity.

3.3.3 Generalized Prime Implicants, Prime Generation and Constrained Covering

To solve an output encoding problem, one has to modify the prime implicant generation and covering strategies basic to Boolean minimization [41] (*cf* Section 2.2). The decomposition problem is slightly different and simpler than the classical output encoding problem since a one-hot coding has already been assumed; the only degree of freedom is in giving the same code to some symbolic outputs.

Prime implicants and covering are basic to two-level Boolean minimization because of the fact that a minimum/minimal cardinality two-level cover can always be composed of only prime implicants. In this manner, the two-level Boolean optimization problem is reduced to a covering problem [83] which involves the selection of an optimum set of prime

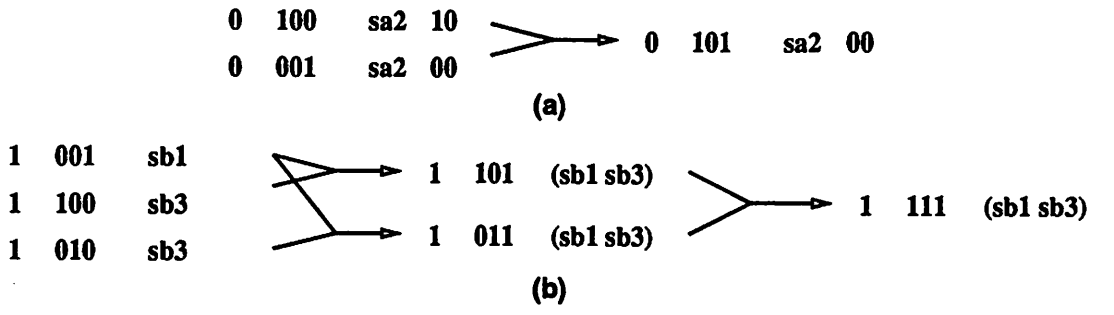


Figure 3.3: Examples of cube merging

implicants such that the functionality of the original specification is maintained. It is shown in the sequel that it is possible to define a notion of primality in terms of *generalized prime implicants* (GPIs) so that the decomposition of FSMs can also be formulated as a covering problem on GPIs. As in two-level Boolean minimization, it can be shown that a minimum cardinality solution can be made up exclusively of GPIs. By restricting the search for the optimum solution to sets of GPIs, the explicit, exhaustive enumeration of all possible combinations of cubes is avoided.

Given a symbolic cover with a multiple-valued input variable and a symbolic output variable in addition to Boolean primary inputs and outputs, a GPI is defined as a cube that (1) is not covered by a larger cube with either the same multiple-valued input literal or a multiple-valued input literal with a one in every position, and (2) has the same symbolic-output tag.

The procedure for generation of GPIs is similar to the Quine-McCluskey procedure with the additional tags corresponding to the symbolic output. Initially, all minterms have tags corresponding to the next-state they assert. Larger cubes are generated by merging smaller cubes. It is possible for two minterms with the same symbolic output tag to merge to form a larger cube. An example of two minterms (belonging to the cover L_a shown above), with the same symbolic-output tag, merging together is shown in Figure 3.3(a). Cubes with different symbolic-output tags can also merge to form larger cubes. If a cube that asserts a symbolic-output sa_i, merges with a cube asserting the symbolic output sa_j, the symbolic-output tag of the resulting cube has both symbols sa_i and sa_j. An example of this form of merging is shown in Figure 3.3(b). After larger cubes are generated in this manner, cubes that are not GPIs can be removed from the set. When no larger cubes can be generated, one has the set of all GPIs. In the example in Figure 3.3(a), the larger cube

is not allowed to remove the cubes that it was obtained from because the multiple-valued input literal of the larger cube is not the same as that of the smaller cubes. In Figure 3.3(b), the cubes **1 011** (sb_1 sb_3) and **1 101** (sb_1 sb_3) are removed because the cube **1 111** (sb_1 sb_3) covers them and has a '1' in every position of its multiple-valued input literal.

GPIs are generated in this manner for the two covers, L_a and L_b, separately. Given the GPIs for L_a and for L_b, the next step is the selection of a subset of GPIs that covers all the minterms in the initial symbolic covers for the two submachines such that the cardinality of the selected set is minimum. Subsequently, the states in the submachines are encoded according to the choices of the states (made during the selection of GPIs) to be given the same/different codes. The code for a state of the overall machine is then the concatenation of the codes of the corresponding states in the submachines. For example, the code for the state s_1 is the concatenation of the codes of the states sa_1 and sb_1. Once the states have been encoded, the multiple-valued input literal of each GPI is replaced by the super-cube containing the minterms corresponding to all the states of the prototype machine present in it. In addition, the symbolic-output tag of each GPI is replaced by the code given to the states present in it (these states are given the same code). Once a GPI has been so modified, it represents a cube in the two-level Boolean cover for the submachine to which it belongs. A selection of GPIs is said to be **encodable** if, after the states of the submachines have been encoded and the GPIs appropriately modified, the functionality represented by the resulting logic is the same as that of the prototype machine. An encodable, minimum cardinality selection of GPIs represents an upper bound on the number of product terms required in a two-level Boolean cover realizing the decomposed machine. For reasons explained in the sequel, an arbitrary selection of GPIs that covers all the minterms in the initial covers of the two submachines is not necessarily encodable. Therefore, the covering problem, to be solved for finding the optimum decomposition, is different from and more difficult than the covering problem associated with classical Boolean minimization. Since the covering procedure is constrained to those selections of GPIs that are legal (encodable), it is termed the *constrained covering problem.*

It remains to clearly define how to constrain the selection of GPIs in order to obtain encodable covers. To that end, the reasons that particular selections may not be encodable are explained.

1 001	(sa1 sa2)	01
0 011	(sa1 sa2)	00
0 010	(sa1 sa2)	11
1 010	(sa3)	10
1 100	(sa3)	01
0 110	(sa1 sa2)	10

Submachine 'a'

1 111	(sb1 sb3)
- 010	(sb1 sb3)
0 101	(sb2)

Submachine 'b'

Figure 3.4: An example of an input constraint violation

Encodability of a GPI Cover

If the selection of GPIs is such that the two states of the prototype machine are not given a different code in some submachine, it is impossible to distinguish between the two states in the decomposed machine. If the two states are not equivalent, the functionality of the decomposed machine obtained in this manner is different from that of the prototype machine. Thus, a selection of GPIs that results in the codes for some pair of non-equivalent states to be the same in both submachines is not encodable. Associated with this reason is a constraint, termed the *output constraint*, that any encodable selection must satisfy; the selection of GPIs should allow any pair of states to have different codes in some submachine.

A second reason exists. As stated earlier, the code for a state in the overall machine is a concatenation of the codes of the corresponding states in the submachines, *e.g.* the code for state s_1 is a concatenation of the codes for the states sa_1 and sb_1. After the encoding of states, the multiple-valued input literal of each GPI is replaced by a super-cube that contains the minterms corresponding to the codes assigned to all the states of the prototype machine present in that literal. A condition that the resulting cube must satisfy in order not to alter the functionality of the prototype machine, is that the super-cube must not contain the code of any state that is absent from the multiple-valued input literal. If all states had been given different codes in each submachine, then there always exists some encoding that satisfies this requirement. But, in case some states are given the same code in a submachine, it is possible that a GPI exists for which no encoding would satisfy the requirement. The constraint on the selection of GPIs associated with this requirement is termed an *input constraint*. While the input constraint encountered here is similar to that in [86], there are some differences. This is best explained by means of an example.

Consider the example, shown in Figure 3.4, which shows a selection of GPIs obtained from the initial covers L_a and L_b (*cf* Section 3.3.2). It can be verified that all the minterms in L_a and in L_b are covered by this selection. It can also be verified that no two states of the prototype machine have been given the same code in both submachines. However, this selection of GPIs is not encodable because of the presence of the GPI **0 011** (sa_1 sa_2) **00** for **Submachine a**. The multiple-valued input literal of this GPI is **011**. No encoding of the states sa_i and sb_i (and hence of the states s_i) can realize a super-cube that contains the codes of the states s_2 and s_3 but not s_1. This is because $\mathbf{e}(sa_1) = \mathbf{e}(sa_2)$ and $\mathbf{e}(sb_1) = \mathbf{e}(sb_3)$. Therefore, the code for s_1 is given by $\mathbf{e}(s_1) = (\mathbf{e}(sa_1) \mathbin{@} \mathbf{e}(sb_1)) = (\mathbf{e}(sa_2) \mathbin{@} \mathbf{e}(sb_3))$[1]. Since the super-cube corresponding to the multiple-valued input literal **011** contains the codes for both the states s_2 ($\mathbf{e}(sa_2) \mathbin{@} \mathbf{e}(sb_2)$) and s_3 ($\mathbf{e}(sa_3) \mathbin{@} \mathbf{e}(sb_3)$), it also contains the codes ($\mathbf{e}(sa_2) \mathbin{@} \mathbf{e}(sb_3)$) and ($\mathbf{e}(sa_3) \mathbin{@} \mathbf{e}(sb_2)$), and therefore the code for s_1. This implies that the minterm corresponding to the code for s_1 is in a cube where it is disallowed, resulting in the corruption of the fanout edges of s_1, hence corrupting the functionality of the decomposed machine. Note that this type of constraint is associated with the absence of a state from the multiple-valued input literal, hence the multiple-valued input literal of all states does not generate such a constraint.

3.3.4 Correctness of the Exact Algorithm

In this section, it is shown that the procedure formulated above does indeed realize the minimum cardinality solution to the decomposition problem.

Lemma 3.3.1 *The input and output constraints, stated in the previous section, are necessary and sufficient to ensure that the functionality of the decomposed machine is identical to that of the prototype machine.*

Proof. *Necessity:* If two non-equivalent states, s_i and s_j of the prototype machine are given the same codes in both submachines, the decomposed machine is unable to distinguish them. Hence, the functionality is not maintained. In addition, if the input constraint is violated, as illustrated by the example in the previous section, a fanout edge of at least one of the states is corrupted; consequently the functionality of the resulting logic is corrupted.

[1]The @ operator, as in (i @ j) denotes the concatenation of two strings, in this case i and j

Sufficiency: Assume the constraints are satisfied. Since the present-state fields of the prototype machine are not split while obtaining L_a and L_b (*cf* Section 3.3.2), there is a one-to-one correspondence between the present-state fields of the decomposed machine and the present-state fields of the prototype machine. No fanout edge of any state is corrupted since all input constraints are satisfied. Also, the selection of GPIs is such that all the minterms of L_a and L_b are covered. Therefore, the functionality of the submachines considered *separately* is maintained. Since a primary output is asserted either in L_a or in L_b, the functionality of the prototype machine with respect to the primary outputs is also retained in the decomposed machine. Also, since the selection of GPIs is such that no two nonequivalent states are assigned the same codes in both the submachines, each pair of next states asserted in the individual submachines for the same primary input and present state combination is associated with a unique next-state in the prototype machine. Because the functionality of L_a and L_b, considered individually, is maintained, this unique next-state is the same as the next-state produced by the prototype machine for the given primary input and present-state combination. Hence, the functionality of the prototype machine with respect to the next-state logic is also maintained. □

Lemma 3.3.2 *A minimum cardinality encodable solution can be composed entirely of GPIs.*

Proof. The proof is by contradiction. Assume that one has a minimum cardinality solution with a cube c_1 that is not a GPI. Then by definition of GPIs (*cf* Section 3.3.3), there exists a GPI covering c_1 such that **(1)** it has the same tag as c_1, **(2)** its multiple-valued input part is either the same as that of c_1 or has a 1 in all its positions, **(3)** its binary-input part covers the binary-input part of c_1, and **(4)** its binary-output part covers the binary-output of c_1. Replacing c_1 by the GPI does not change the functionality because all the minterms covered by c_1 are also covered by the GPI. It does not change the encodability of the solution because the symbolic-output tag of c_1 is the same as that of the GPI, and the multiple-valued input literal of the GPI is either the same as that of c_1 or is all 1's, thus requiring no new encodability constraints to be satisfied. Hence, a minimum cardinality solution made entirely of GPIs can always be obtained. □

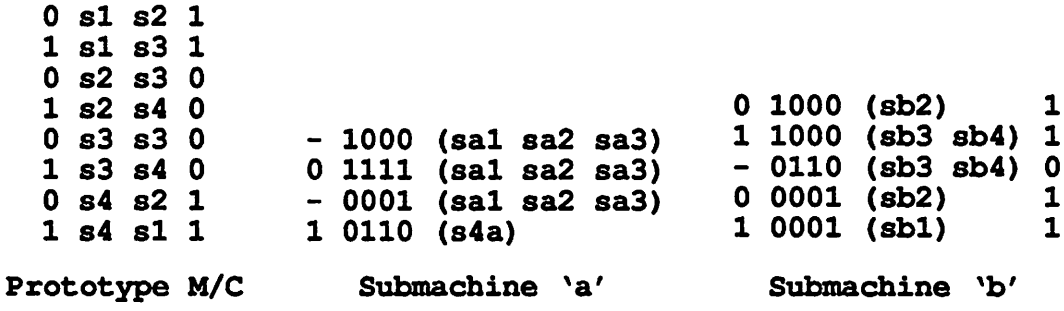


Figure 3.5: Example of a general decomposition

Theorem 3.3.1 *The selection of a minimum cardinality encodable GPI cover for L_a and L_b represents an exact solution,* i.e. *a minimum-cost solution to the decomposition problem, under the chosen formulation.*

Proof. The proof follows from Lemmas 3.1 and 3.2. □

3.3.5 Algorithm for Checking Encodability

In this section, a procedure is illustrated by means of which it can be checked rapidly whether a given selection of GPIs is encodable. The example of the four state FSM shown in Figure 3.5 is used to explain this procedure. Suppose that the selection of GPIs shown in the covers for the submachines in Figure 3.5 has been made for this FSM. It is now required to determine whether this selection of GPIs is encodable.

Checking Output Constraints

To check for output constraint violations, a labeled undirected graph, termed the encodability graph, is constructed in which each vertex is associated with a state s_i in the prototype machine and an edge occurs with label a from vertex s_i to vertex s_j in the graph if the states s_i and s_j co-exist in the tag of some GPI in **Submachine a**. Similarly, there is an edge with label b from vertex s_i to vertex s_j if the states s_i and s_j co-exist in the tag of some GPI in **Submachine b**. The encodability graph corresponding to the selection of GPIs in Figure 3.5 is shown in Figure 3.6. The states s_1, s_2 and s_3 occur in the same output tag in the cover of **Submachine a**. Hence, there exist edges between vertices s_1, s_2 and s_3 labeled a. Similarly, since states s_3 and s_4 occur in the same output tag in the cover

for **Submachine b**, there is an edge between the vertices s_3 and s_4 with label b in the encodability check graph. If any s_i, s_j pair has edges with both labels a and b, it implies that the two states have been assigned the same code in both submachines and the selection is invalid. The dotted edge with label b between states s_1 and s_2 in Figure 3.6 would have existed and caused a constraint violation if the states s_1 and s_2 had been assigned the same code in **Submachine b**.

Since the attempt is to identify partitions [60] in the prototype machine and since groups of states in a partition are disjoint, a transitivity constraint is imposed on the encodability graph whereby if vertices s_i and s_j have an edge with label a between them and vertices s_j and s_k also have an edge with label a between them, then vertices s_i and s_k must also have an edge with label a between them. In terms of codes given to states, this simply means that if states s_i and s_j are given the same code and s_j and s_k are given the same code in **Submachine a**, then s_i and s_k also should be given the same code in **Submachine a**.

A *clique* is a subgraph such that each pair of its constituent vertices is connected by edges with the *same* label. Figure 3.7 is an encodability graph for a decomposition in which states s_1 and s_2 occur in the same output tag of **Submachine a**. If a new edge with label a is added between vertices s_1 and s_3, it becomes necessary to also add another edge with label a between vertices s_2 and s_3 because of the transitivity requirement. As a result, a clique with label a consisting of vertices s_1, s_2 and s_3 is formed. This implies that the states s_1, s_2 and s_3 all must occur in the same output tag in **Submachine a**. A state that is not given the same code as any other state forms a single-vertex clique. The encodability graph is thus composed of a set of cliques satisfying the following properties if the selection of GPIs does not violate an output constraint:

- All the edges in a particular clique have only one type of label. Thus, a clique can be identified with a label. The vertices s_1, s_2 and s_3 form a clique with label a in the graph of Figure 3.6.

- Two cliques with the same label cannot have a vertex in common unless both the cliques are contained in a single larger clique. Let a clique with vertices s_i, s_j and s_k and label a be denoted by $(s_i\ s_j\ s_k)_a$. The cliques $(s_1\ s_2)_a$ and $(s_2\ s_3)_a$ in Figure 3.6 have a vertex in common and have the same label, but, are contained in the larger clique $(s_1\ s_2\ s_3)_a$. This property stems from the transitivity requirement.

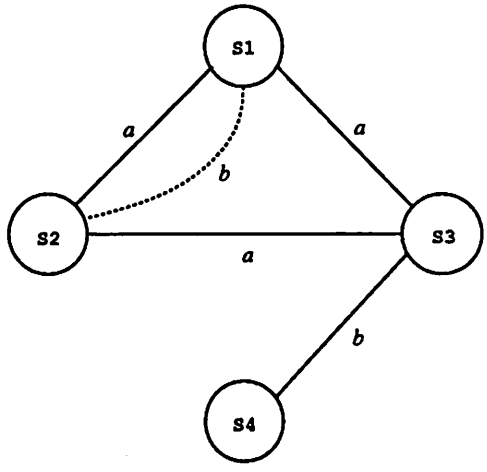


Figure 3.6: Encodability check graph for the decomposition in Figure 3.5

- Any two cliques with different labels can have, at most, one vertex in common. The cliques $(s_1\ s_2\ s_3)_a$ and $(s_1\ s_3)_b$ in Figure 3.6 have different labels and only the vertex s_3 in common. This property follows from the fact the no two states can be given the same code in all the submachines.

To check for an output constraint violation, only a single pass needs to be made through the set of *selected GPIs*. For the output tag of each GPI encountered, the corresponding clique is constructed in the encodability graph and it is checked if any of the three properties above are not satisfied. Checking for the satisfaction of the properties requires a constant number of Boolean operations, where the complexity of each Boolean operation is of the order of the number of states in the prototype machine. If the properties are satisfied, the clique is added to the encodability graph. Otherwise the selection of GPIs is not encodable. Therefore, the complexity of checking for output constraint violations in the encodability check algorithm is of the order of some constant times the product of the number of *selected GPIs* and some constant power of the number of states.

Checking Input Constraints

Once it has been verified that the selection of GPIs does not violate output constraints, violations of constraints imposed by the multiple-valued input literals of the GPIs (*cf* Section 3.3.3) are checked. Multiple-valued input literals with a 1 in all the positions represent input constraints that are trivially satisfied. Similarly, those multiple-valued in-

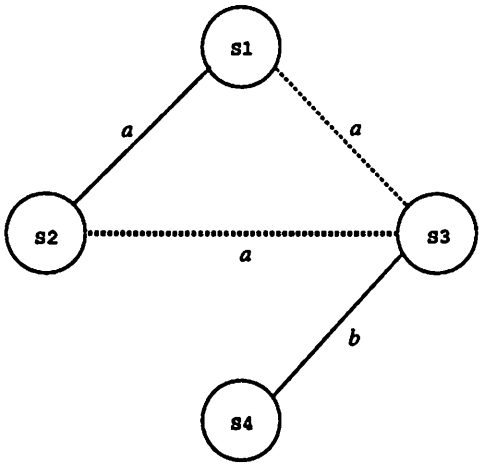


Figure 3.7: Adding a new edge

put literals with only a single 1 also do not represent constraints because its super-cube contains only a single minterm and hence will not contain the code of a state absent from the multiple-valued literal.

A selection of GPIs violates an input constraint *if and only if* there exists a multiple-valued input literal in one of the selected GPIs and a pair of cliques (one an a-clique and the other a b-clique, *cf* Section 3.3.5) created by transitive closure on the a and b cliques in the encodability graph such that the following conditions are satisfied:

- The intersection of the two cliques is non-null.
- The intersection of the two cliques, *i.e.* the vertex common to the two cliques, is a state s_i such that s_i is absent from the multiple-valued input literal.
- There is at least one state in *each* of the two cliques that is also present in the multiple-valued input literal.

It can be verified that these three requirements follow directly from the basic reason for input constraint violations (*cf* Section 3.3.3).

If the number of GPIs in the *selected set* is G, the number of cliques with label a is C_a and the number of cliques with label b is C_b, the number of checks required for input constraint violations is $G.C_a.C_b$ in the worst case since each pair of cliques may have to be checked for every GPI. Assume that there are N states in the prototype machine. If a clique is represented by an N-bit vector, like a multiple-valued input literal (*cf* Section 3.3.2), in

which a 1 in a position corresponds to the presence of a vertex in the clique, a total of βN (β is a constant) bitwise intersections are required for each of the $G.C_a.C_b$ checks. Therefore, the complexity of checking for input constraint violations is of the order of $\beta.N.G.C_a.C_b$. C_a and C_b are always some fraction of G, and given the initial FSM specification, the number of states, N, is a constant. The complexity of checking for input constraint violations is therefore of the order of some constant power of G.

The procedure for checking for input constraint violations is speeded up significantly by the use of the following pruning techniques that significantly reduce the search space:

- Pairs of cliques that have a null intersection need not be considered.
- Any GPI with a multiple-valued input literal in which only a single symbol is present need not be considered.
- Any GPI with a multiple-valued input literal in which all symbols are present need not be considered.
- Any GPI with a multiple-valued input literal that is contained within some clique need not be considered.
- A GPI-clique pair need not be considered if the clique is disjoint from the multiple-valued input literal of the GPI.
- A GPI-clique pair need not be considered if the clique is contained within the multiple-valued input literal of the GPI.

Consider the example given in Figure 3.5. The GPIs **- 1000** (sa_1 sa_2 sa_3), **- 0001** (sa_1 sa_2 sa_3), **0 1000** (sb_2) **1**, **1 1000** (sb_3 sb_4) **1**, **0 0001** (sb_2) **1** and **1 0001** (sb_1) **1** need not be considered for input constraint violation because all of them have only one state in their multiple-valued input literals. The GPI **0 1111** (sa_1 sa_2 sa_3) has a multiple-valued input literal with all the symbols present and therefore cannot cause an input constraint violation. The cliques (*cf* Section 3.3.5 for notation for cliques) in the encodability graph of Figure 3.6 are the following: $(s_1\ s_2\ s_3)_a$, $(s_4)_a$, $(s_3\ s_4)_b$, $(s_1)_b$ and $(s_2)_b$. The single vertex cliques $(s_1)_b$ and $(s_2)_b$ cannot violate constraints, and are therefore not considered. Thus, a constraint violation can only occur due to the cliques $(s_1\ s_2\ s_3)_a$ and $(s_3\ s_4)_b$. The intersection of these two cliques is the state s_3. Therefore, it is only necessary

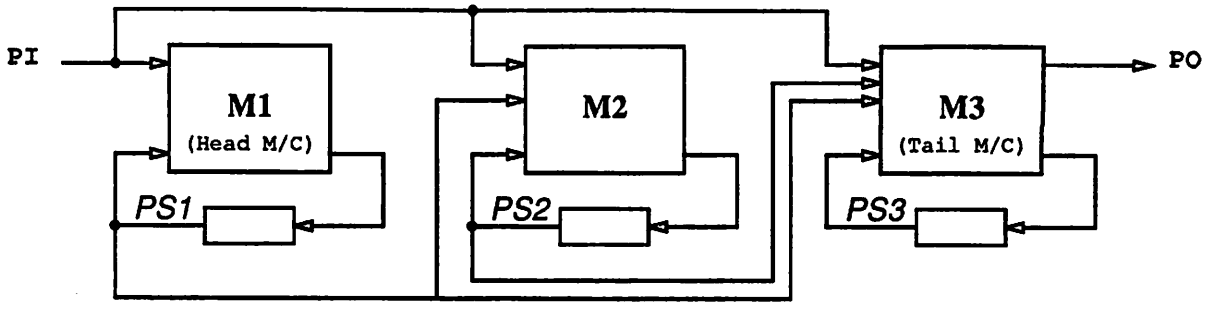


Figure 3.8: Topology for three-way cascade decomposition

to consider those GPIs for possibility of violation that have a multiple-valued input part with the state s_3 *absent*. None of the remaining GPIs satisfy that requirement. Therefore, no violation of input constraints exists in the selected set of GPIs shown in Figure 3.5.

If neither the output constraint nor the multiple-valued-input constraint are violated, a code is guaranteed to exist for the states such that the sum of the cardinalities of the encoded two-level representations of the submachines is bounded by the total number of selected GPIs. Such a selection is said to be encodable.

Relative Complexity of Encodability Checking

As has been shown in Sections 3.3.5 and 3.3.5, the complexity of checking for output and input constraint violations is polynomial in the number of *selected GPIs*. Covering on the other hand is NP-complete in the total number of GPIs [47]. The bottle-neck in the exact-decomposition procedure is therefore the covering procedure rather than encodability checking.

The exact-decomposition algorithm can be extended to decomposition into multiple component machines and in general to arbitrary topologies. The reasons that this exact algorithm may not be viable for a given problem are that the number of GPIs may be too large and/or the covering problem may not be solvable in reasonable time. Therefore, a heuristic procedure is required to solve the problem (*cf* Section 3.5).

3.4 Arbitrary Topologies

In this section, it is shown that by merely changing the encodability constraints, it is possible to target the logic-level optimality of arbitrary topologies.

3.4.1 Cascade Decompositions

The topology for a three-way cascade decomposition is shown in Figure 3.8. The characteristic property of cascade decompositions is that information flows in only one direction. For example, a submachine in Figure 3.8 receives as input only the primary input and the present state of the submachines to its left. The submachine that only receives as input the primary inputs is called the head machine, and the submachine that receives as input the present states of all the submachines is called the tail machine. Given a prototype machine, one can target such a topology by specifying the appropriate constraints on the selection of GPIs. The initial partitioning along the next-state and primary output fields, and the subsequent generation of GPIs is carried out in the same manner as in the case of two-way general decomposition (*cf* Section 3.3.2). Imposing the following constraints on the covering procedure then ensures that the topology corresponds to a cascade decomposition:

1. The code of a state from the prototype machine should be different from the code of any other state in at least one of the submachines. A selection of GPIs in which two states are in the same output tag in all the submachines is not encodable.

2. Say that **Submachine a** is a part of the cascade chain with submachines preceding it and following it. Assume that states s_1 and s_2 have not been distinguished by an appropriate selection of GPIs in a submachine preceding **Submachine a**. If s_1 and s_2 are now given the same code in **Submachine a**, all the next-state pairs for s_1 and s_2 also must be given the same code in **Submachine a**. The head machine is a special case because there are no submachines preceding it. In the head machine, therefore, this constraint has to be satisfied for any pair of states that are given the same code. Satisfying this constraint for the head machine corresponds to satisfying the *preserved partition* [60] requirement for cascade decompositions.

3. The constraint on the multiple-valued input part of the GPIs is slightly different from the case of general decomposition. Again, consider a submachine, say **Submachine a**, that is a part of the cascade chain with submachines preceding it and following it. Let MV_a be a multiple-valued input literal of a GPI for **Submachine a**. Assume that state s_1 is absent from MV_a, and states s_2 and s_3 are present in MV_a. The states s_2 and s_3 then must be distinguished from s_1 either in **Submachine a** or in the same submachine *preceding* **Submachine a**.

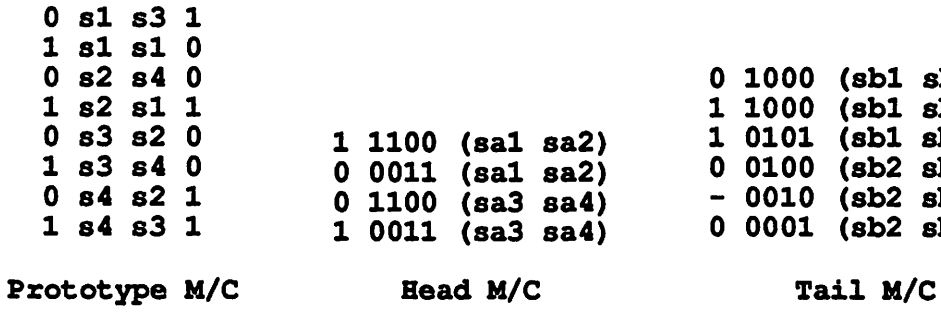


Figure 3.9: An Example of a two-way cascade decomposition

An example of a two-way cascade decomposition that does not violate any constraint is shown in Figure 3.9. A selection of GPIs that constitutes the cascade decomposition is shown in the figure. It can be seen from the figure that in the head machine, the states that have been given the same codes are contained in the same multiple-valued input literal and, consequently, have the same next-state fields. Thus, the second constraint is not violated. It can be verified that the second constraint would have been violated, for example, had states s_1 and s_2 been given the same code in the head machine but states s_3 and s_4 had been given different codes. It can also be verified that the third constraint is not violated. It would have been violated, for example, had the GPI **0 0110** (sb_2 sb_4) **0** been present in the cover for the tail machine because the state s_1, absent from the multiple-valued input literal of this GPI, does not have a code different from the codes of states s_2 and s_3 in the same submachine.

As in the case of general two-way decomposition (*cf* Section 3.3.4), it can be shown that the above constraints for decomposition into cascade chains are necessary and sufficient to ensure that the decomposition has the same functionality as the prototype machine.

Lemma 3.4.1 *The three constraints described in the above section are necessary and sufficient to ensure that a selection of GPIs satisfying them results in a cascade decomposition that has the same functionality as the prototype machine.*

Proof. The proof is similar to the proof for Lemma 3.3.1. *Necessity:* **(1)** If the decomposition is such that two non-equivalent states are given the same code in all the submachines, the decomposed machine is not able to distinguish between them. Hence, the functionality is not maintained. **(2)** If the states that are given the same code in some submachine, say **Submachine a**, were not given different codes in a submachine preceding **Submachine**

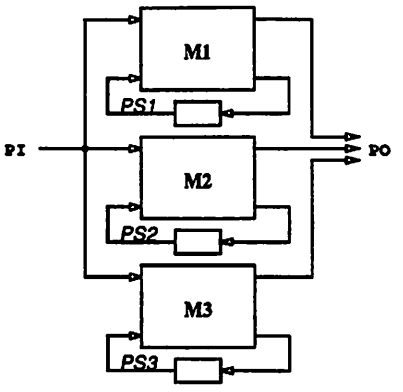


Figure 3.10: Topology for three-way parallel decomposition

a, it is not possible for **Submachine a** to distinguish between. This is so because any submachine receives present-state information only from the submachines that precede it. Therefore, the next states and the outputs must be the same in **Submachine a** for these two states for any primary input combination. **(3)** The third constraint is similar to the input constraint in general decomposition, with the difference that the super-cube for a multiple-valued input literal in a submachine consists only of its own present-state lines and those from the preceding submachines. Say a GPI exists in **Submachine a** such that a state which is absent from its multiple-valued input literal does not have different codes from the states that are present in that multiple-valued input literal in **Submachine a** or in some submachine preceding **Submachine a**. Then, any super-cube containing the codes of the states present in the multiple-valued input part will also contain the code of the state absent from it (as in the example in Section 3.3.5). Thus, the state absent from the multiple-valued input part would assert an incorrect next state or primary output. Therefore, the functionality of the machine would be changed. *Sufficiency:* If the constraints in this lemma are satisfied, the constraints for general decomposition (which are weaker, *cf* Lemma 3.3.1) also get satisfied. Also, the constraints for general decomposition were shown to be sufficient to maintain functionality. Therefore, it follows that the constraints in this lemma are also sufficient to maintain functionality. Also, if the second constraint is satisfied, each submachine only needs to know the present state of the submachines preceding it. Therefore, if the second constraint is satisfied, the decomposition corresponds to a cascade decomposition. □

```
0 s1 s2 01
1 s1 s3 10
0 s2 s1 00
1 s2 s4 11
0 s3 s4 11      0 1100 (sa1 sa2) 0     0 1010 (sb2 sb4) 1
1 s3 s1 00      1 1100 (sa3 sa4) 1     1 1010 (sb1 sb3) 0
0 s4 s3 10      0 0011 (sa3 sa4) 1     0 0101 (sb1 sb3) 0
1 s4 s2 01      1 0011 (sa1 sa2) 0     1 0101 (sb2 sb4) 1

Prototype M/C      Submachine 'a'          Submachine 'b'
```

Figure 3.11: An example of a two-way parallel decomposition

3.4.2 Parallel Decompositions

The topology for a three-way parallel decomposition is shown in Figure 3.10. The characteristic property of parallel decompositions is that all the submachines operate independently of each other. Thus, a submachine does not receive as input the present-state lines of *any* other submachine. All the submachines are, therefore, similar to the head machine of a cascade chain. Given a prototype machine, one can target such a topology by specifying the appropriate constraints. The initial partitioning and the subsequent generation of GPIs for a parallel decomposition is carried out in the same manner as in the case of general decomposition (*cf* Section 3.3.2). The following constraints are then imposed on the selection of the GPIs to ensure that a parallel decomposition is obtained:

1. The code for a state in the prototype machine should be different from the code for any other state in at least one of the submachines. A selection of GPIs in which two states are in the same output tag in all the submachines is not encodable. This constraint is identical to the corresponding constraint for general and cascade decompositions.

2. No submachine in a parallel decomposition receives as input the present state of any of the other submachines. Therefore, states s_1 and s_2 can be given the same code in a submachine only if, for each input, the primary output combination asserted by s_1 is the same as that asserted by s_2. Also, as in the case of the head machine in a cascade decomposition, if the states s_1 and s_2 are given the same code in a submachine, the next-state pairs of s_1 and s_2 should also be given the same code. For example, assume that for input i, the next state of s_1 is s_3, and the next state of s_2 is s_4. Then if s_1 and s_2 are given the same code, s_3 and s_4 should also be given the same code.

3. The third constraint is similar to the corresponding constraint in general and cascade decompositions except that the state absent from the multiple-valued input literal has to have a different code from all the states present in that multiple-valued input literal in that same submachine.

An example of an encodable selection of GPIs resulting in a two-way parallel decomposition is shown in Figure 3.11. As in the case of cascade and general decompositions, a lemma can be stated regarding the necessity and sufficiency of the encodability constraints for parallel decompositions.

Lemma 3.4.2 *The three constraints described in the above section are necessary and sufficient to ensure that a selection of GPIs satisfying them results in a parallel decomposition that has the same functionality as the prototype machine.*

Proof. The proof is similar to the proof for Lemma 3.4.1. *Necessity:* **(1)** If the decomposition is such that two non-equivalent states are given the same code in all the submachines, the decomposed machine is not able to distinguish between them. Hence, the functionality is not maintained. **(2)** If two states that are given the same code in some submachine, say **Submachine a**, it is not possible for the submachine to distinguish between them because the submachine does not know the state of any other submachine. Therefore, the next states and the outputs must be the same in **Submachine a** for these two states for any primary input combination in order to maintain functionality. **(3)** The necessity of the third constraint is shown by an argument almost identical to the argument for the third constraint for cascade decomposition. The difference is that since a submachine does not have any other submachines preceding it in parallel decomposition, the codes for two states present in the multiple-valued input literal of a GPI in a submachine have to be different from the code of a state absent from that literal in the same submachine in order to maintain functionality. *Sufficiency:* If the constraints in this lemma are satisfied, the constraints for general decomposition (which are weaker, *cf* Lemma 3.3.1) also get satisfied. Also, the constraints for general decomposition were shown to be sufficient to maintain functionality. Therefore, it follows that the constraints in this lemma are also sufficient to maintain functionality. Also, if the second constraint is satisfied, no submachine requires that any other submachine precede it. Therefore, if the second constraint is satisfied, the decomposition corresponds to a parallel decomposition. □

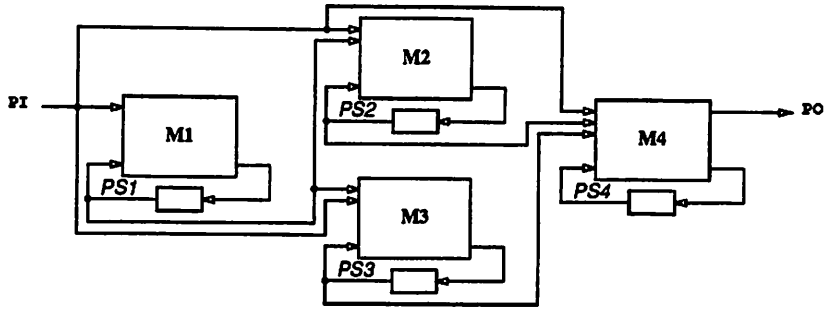


Figure 3.12: An arbitrary decomposition topology

3.4.3 Arbitrary Decompositions

An example of a topology that does not conform to any of the preceding topologies is shown in Figure 3.12. It is possible to target such topologies also by suitably modifying the covering constraints. The basic output constraint that any pair of states of the prototype machine should have distinct codes in the decomposed machine remains unchanged. The remaining constraints are dependent on the other submachines that a particular submachine receives present-state information from. The type of constraints imposed are briefly illustrated by means of the example topology in Figure 3.12. The constraints for submachine M_1 are the same as those for the head machine of a cascade chain. In the case of submachine M_2, if two states that have the same code in M_1, are given the same code in M_2 also, then all the next-state pairs of these two states should also be given the same codes in M_2. Similarly, if a state is absent from a multiple-valued input literal of M_2, it should have a different code from the states present in that multiple-valued literal in either M_1 or M_2. The justification for this constraint is similar to that for cascade and parallel decompositions. The constraints for M_3 are identical to those for M_2. The constraints for M_4 are similar. *In general, it is possible to target any desired topology by formulating the decomposition problem as a covering problem and suitably modifying the covering constraints.*

3.4.4 Exactness of the Decomposition Procedure

Lemmas 3.4.1 and 3.4.2 for the cascade and parallel decomposition cases, respectively, are similar to Lemma 3.3.1 for the case of two-way general decomposition. Similar lemmas can also be stated regarding the encodability constraints for arbitrary topologies. These lemmas in conjunction with Lemma 3.3.2 show that the decomposition obtained

in this manner is actually the minimum cardinality solution. It should be noted that an arbitrary decomposition topology with a series-parallel structure can also be obtained by successive application of 2-way decomposition, except that no minimum cardinality guarantee can be given.

3.5 Heuristic Procedure for Two-Way General Decomposition

3.5.1 Overview

The basic iterative strategy that has been used successfully for the two-level Boolean minimization problem appears promising for two-way general decomposition also. The encodability requirements for the selected GPIs are defined in the same manner as for the exact procedure (*cf* Section 3.3.3). But, instead of enumerating all the GPIs, one begins with a set of GPIs corresponding to L_a and L_b (*cf* Section 3.3.2), and an attempt is made to reduce their count, while maintaining the encodability of the GPI covers. Operations similar to the *reduce* and *expand* operations of MINI [62] and ESPRESSO [18, 94] can be performed in an effort to minimize the cover cardinalities.

The three basic steps in the iterative loop are given below in the order in which they are carried out:

- Symbolic-reduce
- Symbolic-expand
- Minimize covers and remove input constraints

The cost function that is used for the iterative procedure is the same as that used for the exact algorithm, namely, the sum of the cardinalities of the minimized one-hot coded submachines. The steps of the iterative procedure are repeated until the cost of the solution, given by this cost function, remains unchanged after a pass through the loop.

The symbolic-reduce and symbolic-expand steps attempt to modify the symbolic-output tags of the GPIs with the goal of reducing the cover cardinality. These two operations were so named because of their obvious similarity to the reduce and expand steps in iterative two-level Boolean minimization. The reduce and expand steps in Boolean minimization

attempt to modify the cubes making up the Boolean cover so that the possibility of reducing the cover cardinality after the *irredundant* step is increased. The steps of the basic iterative loop are explained below.

3.5.2 Minimization of Covers and Removal of Constraint Violations

In the minimization step, which follows symbolic-expand, a two-level minimization of the covers for both the submachines is carried out. This step incorporates *all* the cube-merging that becomes possible as a result of the symbolic-expand. The cover produced as a result of this minimization is termed the *over-minimized* cover because the minimization is carried without taking into account violations of the input constraints, and therefore may *not* be encodable. The conditions giving rise to such a violation were described in Section 3.3.5. When a violation of a multiple-valued input constraint is detected, the GPI with the constraint violation is split into two new cubes along the multiple-valued input part so that the multiple-valued input part of the first new cube is contained entirely within one of the two cliques (*cf* Section 3.3.5) and the multiple-valued input part of the second new cube intersects only the remaining clique (does not intersect the other clique). This method of splitting the cube is not unique but is effective because at least one of the new cubes is devoid of any input constraint violations and the cardinality of the cover is only increased by one. The end result of this *unraveling* of the multiple-valued input literals of the GPIs is to make the over-minimized cover encodable.

3.5.3 Symbolic-expand

The goal of symbolic-expand is to increase the size of the output tags of GPIs (by giving states in submachines the same codes) in order to maximize the possibility of cube merging (*cf* Section 3.3.3). The sizes of the output tags are expanded until any further expansion would result in the violation of an output constraint (*cf* Section 3.3.3).

The atomic operation in the expansion procedure inserts two states in the same output tag of a GPI, checking while doing so that the output constraint is not violated. This operation is carried out alternately between the two submachines. The state pair is given a label corresponding to the submachine with which it is associated. The following three non-trivial cases are possible when a state pair is selected for insertion:

- The state pair intersects no clique (*cf* Section 3.3.5) with the same label in the encod-

ability graph, or if it does intersect a clique, the clique has a different label and the cardinality of the intersection equals one. In this case, a new clique is added to the graph with the same label as the state pair.

- The state pair (a) intersects only one clique in the graph, (b) that clique has the same label, and (c) the cardinality of that intersection is one. In this case, the state pair is merged with that clique if the creation of the new clique (the label of the clique is unchanged) is such that no output constraint is violated. Otherwise the state pair cannot be inserted.

- The state pair intersects two cliques with the same label.The cardinality of each intersection has to be one in this case because no output constraint is violated up to the current point. In this case, the two cliques are merged into a single new clique with the same label if the new clique does not violate an output constraint. If an output constraint is violated upon merging, the state pair cannot be inserted.

The effectiveness of the expansion procedure depends a great deal on the order in which the state pairs are inserted into the same tag. To maximize the possibilty of cardinality reductions, state pairs can be heuristically ordered at the beginning of every symbolic-expand. In order to arrive at a heuristic, it is necessary to analyze the effect of giving a pair of states the same code in a submachine on the cardinality of the minimized covers. Deciding to give the states s_1 and s_2 the same code in **Submachine a** (by inserting them in the same tag of a GPI) can affect the cardinality of the overall cover, after the cover has been minimized, in the following two ways:

Firstly, cubes that asserted the next-state s_1 and cubes that asserted the next state s_2 in **Submachine a** may merge, making the total number of cubes that assert the next states s_1 and s_2 smaller than before this expansion step. Thus, an expansion step can lead to a reduction in the cardinality of the cover (in **Submachine a** or **b**) in which the expansion is carried out. In order to maximize the above cardinality reductions, one could assign large weights to those next-state pairs that have a large number of common present states, or that are asserted by a large number of common primary input combinations. The state pairs would then be chosen for merging in the order of decreasing weight.

Secondly, the insertion of a pair of states into the same output tag of a GPI in **Submachine a** generates *new* input constraints for *both* **Submachine a** and **b**. To

make the cover encodable, the multiple-valued input parts are unraveled (*cf* Section 3.5.2), thereby invalidating some of the merging of cubes that occurred in the minimization step. It is possible for the increase in cardinality due to unraveling to be greater than the reduction in cardinality due to minimization *if* the choice of the state pair is inappropriate. In such a situation, the expansion step would, actually, lead to an increase in the overall cardinality rather than a reduction.

The following heuristic can be used to minimize the increase in cardinality due to unraveling. Large weights are given to present-state pairs that assert the same next states and primary outputs[2]. Pairs of states are then chosen for merging in the order of decreasing weight. The basis for this heuristic is the following. Assume that states s_1 and s_2 occur in the same multiple-valued input part and state s_3 is absent from it. The input constraint arising from such a multiple-valued input part is that states s_1 and s_2 must be distinguished from s_3 in the same submachine. Now, in two-way decomposition, if s_1 and s_2 are given the same code in a submachine, then the above input constraint is satisfied automatically if it is ensured that s_1, s_2 and s_3 are given different codes in at least one submachine (*i.e.* the output constraint is satisfied for s_1, s_2 and s_3). Since the output constraints are satisfied during the symbolic-expand step itself, no unraveling of the multiple-valued input part is required and the cardinality does not increase. Since two states asserting the same primary outputs and next states are likely to be in the same multiple-valued part in the minimized cover frequently, the heuristic attempts to give a high priority to assigning such states the same code.

3.5.4 Symbolic-reduce

Symbolic-reduce is essential to the iterative process for moving out of the local minimum that it may have entered following the symbolic-expand and minimization steps. The basic operation used by symbolic-reduce is to remove a state from the symbolic output tags that it is contained in, while maintaining functionality. The following heuristic is used for symbolic-reduce. The states selected for removal are those whose insertion in the output tags of GPIs during the symbolic-expand generated new input constraints. Since the input constraints are generated whenever there is non-null intersection between cliques with different labels (*cf* Section 3.3.5), symbolic-reduce is carried out until the intersection

[2]Such pairs of present states are likely to occur together in multiple-valued input parts frequently.

between *every* pair of cliques becomes null. Such a cover is said to be *maximally reduced.* This formulation of the symbolic-reduce operation is order independent.

3.6 Relation to State Assignment

A state assignment algorithm for optimum two-level implementations was presented in [41]. In that work, the problem of optimum state assignment was formulated as a constrained covering problem similar to the formulation of decomposition presented in this chapter. The difference in the application of constrained covering to decomposition and state assignment lies in the step that checks for the encodability of a selected set of GPIs. The goal of two-way decomposition is merely to find two partitions on the states where each partition could consist of a number of blocks. Decomposition does not carry out the complete encoding of the states, it merely 'preprocesses' the states so that the subsequent state encoding applied on this preprocessed set of states will be guaranteed to realize the decomposition with the desired topology. As a result, the constraints involved in two-way decomposition are much simpler than the constraints that must be checked for optimum state encoding, and the two-way decomposition problem is simpler than optimum state encoding.

State encoding can be viewed as the problem of finding the optimal general decomposition of the prototype machine into as many submachines as there are state bits in the final state encoding. Based on this premise, one approach to making state encoding tractable for large problem sizes is to decompose the STG prior to the actual encoding. The decomposition makes the subsequent encoding steps much more tractable. The goal in such, and for that matter any, decomposition is not to compromise the optimality of the final solution.

3.7 Results

The motivation for FSM decomposition was elucidated in some detail in Section 3.1. Based on the properties desired, the efficacy of a decomposition can be judged from the following criteria:

- The sum of the areas of the two-level (multilevel) implementation of the encoded submachines compared to that of the two-level (multilevel) implementation of the

Example	States	PI	PO	One-Hot Card.	Enc. Bits	Enc. Card.[1]	Two-level Area[1]
bbara	10	4	2	34	4	24	528
bbsse	16	7	7	30	4	29	957
bbtas	5	2	2	16	3	8	120
beecount	7	3	4	12	3	10	190
dk27	7	1	2	10	3	9	117
dk512	15	1	3	21	4	20	340
ex4	14	6	9	21	4	15	465
fsl	67	8	1	583	7	119	4522
fstate	8	7	7	22	4	16	528
modulo12	12	1	1	24	4	12	180
sla	20	8	0	92	5	73	2263
scf	121	27	54	151	7	137	17947
scud	8	7	6	86	3	62	1798
styr	30	9	10	111	5	94	4042
tav	4	4	4	12	2	10	180
tbk	32	6	3	173	5	57	1710

[1]**Using NOVA [106] for state assignment.**

Table 3.1: Statistics of the encoded prototype machines

encoded prototype machine.

- The areas of the two-level (multilevel) implementation of each encoded submachine compared to that of the two-level (multilevel) implementation of the encoded prototype machine.

- The total number of inputs and outputs in each encoded submachine compared to the number in the encoded prototype machine.

The heuristic procedure for optimal two-way general decomposition has been implemented in a program called HDECOM. The input to HDECOM is a KISS-style [86] state table description of the prototype machine. As output, HDECOM can produce either a fully encoded decomposed machine or a decomposed machine in which the present-state inputs to the submachines and their next-state outputs are symbolic. The heuristic algorithm in HDECOM was tested on a number of examples obtained from the MCNC FSM benchmark suite [76] and industrial sources. The statistics of the chosen examples are shown in Table 3.1. The first step in the decomposition process is to obtain the STG representations

Example	States	Area[1] of Decomposed M/C[2]	Area[1] of Lumped Machine		
			Random[3]	KISS[86]	NOVA[106]
bbara	4/3	360/180/**540**	649	650	**528**
bbsse	10/2	950/168/**1118**	1144	1053	**957**
bbtas	3/2	91/60/**151**	215	195	**120**
beecount	4/2	126/105/**231**	293	242	**190**
dk27	4/2	55/30/**85**	143	117	**117**
dk512	8/2	238/104/**342**	418	414	**340**
ex4	4/4	128/325/**453**	627	589	**465**
fs1	16/5	3430/1254/**4684**	6764	5510	**4522**
fstate	5/2	450/39/**489**	600	726	**528**
modulo12	6/2	117/44/**161**	180	225	**180**
s1a	8/3	1246/616/**1862**	3108	2263	**2263**
scf	90/2	14832/3135/**17967**	21278	18760	**17947**
scud	4/2	1050/720/**1770**	2533	2698	**1798**
styr	16/2	3589/728/**4317**	5591	4186	**4042**
tav	2/2	75/75/**150**	198	180	**180**
tbk	16/2	1275/425/**1700**	6114	4410	**1710**

[1]Two-Level Area.
[2]Encoding for each submachine obtained using NOVA [106].
[3]Average random solution.
The first number in a column is the value for the first machine, the second for the second machine and the third, if present, for the overall decomposed machine

Table 3.2: Results of the heuristic two-way decomposition algorithm

of the submachines into which the prototype machine is decomposed. The second step involves the implementation of the individual submachines, *i.e.* an encoding of the states of the submachines and a minimization of the resulting logic. In Table 3.2, the statistics of the final two-level implementations for the decomposed machines are shown. The encoding of the states of the submachines was obtained using NOVA [106]. Also shown in Table 3.2 are the areas of the logic-level implementations of the undecomposed prototype machines. The logic-level implementations of the prototype machines were obtained using three different state-encoding strategies: **(1)** Random Encoding **(2)** KISS [86] and **(3)** NOVA [106]. NOVA is a second generation state encoding program that represents the state of the art in heuristic encoding techniques targeting two-level implementations. NOVA typically produces better results than KISS. It can be observed from Table 3.2 that the overall area of the

decomposed machine is better than the area of the prototype machine implemented using random encoding or using KISS. It can also be observed that the implementation of the decomposed machine compares favorably in area to the prototype machine encoded using NOVA. Thus, the goal of decomposing FSMs and at the same time keeping a tab on the cost of the resulting logic-level implementation has been achieved. In the decompositions shown in Table 3.2, the individual submachines are generally much smaller in area than the prototype machine, illustrating that at least in the case of two-level implementations, a decomposition of the FSM translates directly into improved performance.

The proposed decomposition approach does not directly target either the area or the performance of multilevel implementations. Therefore, it is not possible to claim that it can be used to target area or performance optimality of multilevel implementations. Even so, we find in many examples that the significantly smaller PLA areas of the submachines result in the area-optimized multilevel-logic implementation of each submachine being much smaller than that of the prototype machine. Reported in Tables 3.3 are the multilevel areas for some examples. It can be seen that the multilevel area of the decomposed machine is comparable to that of the best multilevel implementation of the prototype machine for these examples, and that the area of each submachine is much smaller than that of the prototype machine.

As an example, consider **tbk** which is a moderately sized FSM with 32 states. A KISS encoding of **tbk** requires a two-level area of 4410 units while a NOVA encoding requires an area of 1710 units. When **tbk** is decomposed prior to the two-level implementation and the submachines are then encoded using NOVA, the total two-level area required is 1700 units. The area of the individual submachines in this case is only 1275 and 425 units, representing a reduction in area compared to the NOVA encoding of the prototype machine by 25% and 75%, respectively.

The exact algorithm for decomposition was also implemented and the results are reported in Table 3.4. The same encoding strategies as used in the case of heuristic decomposition have been used to compare the implementations of the decomposed and prototype FSMs. The exact algorithm is not viable for large problem instances because the number of GPIs tends to be extremely large, resulting in excessive memory requirements and an untractable covering problem.

Because the submachines in a decomposition could have common inputs, extra routing area is required, over and above the PLA area. Typically, this extra area is small in

Example	Literals in Decomposed M/C[1]	Literals in Lumped M/C[1]	
		NOVA[106]	JEDI[74]
bbara	37/26/63	59	54
bbsse	100/18/118	103	99
bbtas	15/10/25	22	22
beecount	21/13/34	32	33
dk27	17/6/23	25	22
dk512	41/25/66	53	50
ex4	31/26/57	57	57
fstate	39/20/59	62	-
modulo12	18/9/27	22	22
s1a	176/59/235	244	130
scud	101/97/198	195	-
tav	12/13/25	25	26

[1]After optimization using the standard script in MISII.
The first number in a column is the value for the first machine, the second for the second machine and the third, if present, for the overall decomposed machine

Table 3.3: Comparison of literal counts of multilevel-logic implementations

Example	States	Area[1] of Decomposed M/C[2]	Area[1] of Lumped Machine		
			Random	KISS[86]	NOVA[106]
contrived	4/4	40/66/**106**	108	120	**108**
shiftreg	4/2	36/10/**46**	132	72	**48**
lion9	5/2	112/5/**117**	170	136	**136**

[1]Two-Level Area.
[2]Encoding for each submachine obtained using NOVA [106].
The first number in a column is the value for the first machine, the second for the second machine and the third, if present, for the overall decomposed machine

Table 3.4: Results of the exact decomposition algorithm

comparison to the PLA areas and does not offset the area gain via decomposition. In addition, a submachine may actually be independent of some of the primary input and present-state lines, thus reducing the number of inputs required to be routed to that submachine. An example of such a case would be a cascade decomposition in which the operation of the head machine is completely independent of the present state of the tail machine.

Since the logic for a primary output or next-state line is entirely contained in a single submachine, it is apparent that the decomposition does not add multiple levels of logic between latch inputs and outputs. The effect of decomposition is therefore similar to partitioning the set of primary outputs and next-state lines in the overall FSM into two (or more in case of decomposition into multiple submachines) groups and implementing the logic driving each group of primary output and next-state lines separately. Such a partitioning of logic is termed **vertical partitioning**. In general, it is not necessary that a good vertical partition should exist for an arbitrary implementation of a FSM. [3] The decomposition procedure, on the other hand, *ensures* that a good vertical partition does exist.

The results imply that a good decomposition targeting two-level area can be a good decomposition for the multilevel case.

These results are significantly better, in general, than those obtained via factorization [40] due to a different problem formulation and the targeting of an improved cost function.

3.8 Conclusions

Exact and heuristic algorithms for optimum and optimal two-way general decomposition of finite-state machines were proposed in this chapter. These algorithms are based on a cost function that is more reflective of the cost of a logic-level implementation of the decomposed machine than the cost function used by previous approaches to the decomposition problem. It was shown in the chapter the exact algorithm for optimum two-way decomposition can be generalized to target arbitrary topologies. The heuristic algorithm has been implemented in the program HDECOM.

Decomposition provides a method for simplifying the process of constraint gen-

[3] Experimental results using vertical partitioning on FSMs encoded using the state assignment programs KISS and NOVA indicate that partitions that decrease overall machine area are not usually found.

eration for state encoding. The complexity of the optimal constraint generation problem for state encoding grows exponentially with the problem size. Decomposition by symbolic-output partitioning provides a way of simplifying the constraint generation problem. Rather than impose constraints on each bit of the code for every state (as would be done in state encoding), decomposition allows constraints to be imposed on groups of bits at a time. The actual encoding for each of these groups can subsequently be obtained in the second step. The heuristics described in this chapter for generating the constraints to be imposed on groups of bits are quite simple-minded. It is possible that combining factorization [40] with symbolic-output partitioning can lead to superior results.

Chapter 4

Synthesis from Logic-Level Descriptions

4.1 Introduction

Techniques for optimizing, verifying and testing FSMs have relied on the use of State Transition Graph (STG) or State Transition Table (STT) descriptions of the machines. While the STG is an easily manipulable representation of behavior, VLSI sequential circuits, consisting of large, interacting FSMs, can require astronomical amounts of memory and CPU time to store and generate from logic-level descriptions of the machine.

The problem with conventional STG descriptions is twofold. First, the STG is a flattened sum-of-products representation. A variety of VLSI sequential circuits have combinational portions that require exponential amounts of storage in sum-of-products form. The second, and the more severe problem, especially for controller-type circuits, (which can be flattened to two-level logic quite easily) is that all states in the STG are constrained to be *minterm* states, *i.e.* all state variables are set to 0/1 values in each state. This implies that states in the sequential circuit that are equivalent have to be represented by distinct minterms (or equivalently, by distinct symbols) in the STG.

The second restriction precludes the optimization of some sequential circuits, typically those that are designed as interconnections of FSMs, because they tend to have a large number of equivalent states. While the state minimized representation is small, the conventional STG with equivalent states represented as minterms is usually large. Thus, the

conventional STG is unsuitable as a symbolic representation that can be used for sequential optimization. Obtaining a better representation is one of the problems addressed in this chapter.

Previous attempts at solving this problem have relied on distributed-style STG representations (*e.g.* [30]) and retiming-based algorithms (*e.g.* [79]). Unfortunately, to perform global optimization, the approaches of [30] require, in the limit, information corresponding to the entire STG of the interacting set of FSMs, which grows rapidly with circuit size. Sequential synthesis approaches like those in [79] that operate at the logic-level hold promise as far as efficiency and accurate cost functions are concerned, but to date are also lacking in global optimization capabilities.

It is shown in this chapter that it is possible to extract *Implicit State Transition Graphs* (ISTGs) from logic-gate and flip-flop descriptions of sequential circuits that allow equivalent states to be represented as cubes, and edges from different states merged into a single edge, thereby decreasing significantly the CPU time and memory requirements of the extraction process, and resulting in a manipulable symbolic representation.

Optimization algorithms based on ISTGs for FSMs described at the logic level are proposed in this chapter. A sum-of-products representation is used for ISTGs because most mature state assignment and decomposition strategies use such a representation. However, the algorithms and ideas presented here can quite easily be modified to use alternate representations. Certain sequential circuits, notably ALUs interconnected with registers, are not amenable to sequential logic optimization. While algorithms exist that can verify/test such circuits, improving the performance of such circuits via re-encoding or re-decomposition appears improbable. Selection strategies that focus on the control portions of a sequential circuit, where the most room for sequential optimization exists, are a must for real-life chips. The approaches presented here appear promising for global, sequential optimization of interacting FSM controllers. Experimental results using these techniques are also presented in this chapter.

4.2 Implicit STGs

4.2.1 Implicit State-Enumeration

The implicit state-enumeration procedure is given in Figure 4.1 and is illustrated with the help of a machine whose STG is shown in Figure 4.2. The state whose *fanout edges* will be enumerated is denoted by S_1. Initially, S_1 is the reset state of the machine (in this example $S_1 = 000$). In general, S_1 can be minterm or a cube state. At first, S_1 is checked to ensure that it is not in the path currently being enumerated. Following that, a PI vector I (if possible, the largest cube) that produces a 1 or a 0 when concatenated with S_1, for each PO line in the machine is determined. This is performed by heuristically selecting a

```
stg_enumerate(S1) {
  /* S1 is the current state */
  if (S1 is in the current path)
    return (TRUE);
  Add S1 to current path ;
  while (decision_tree != NULL) {
    (S'1, flag) = set_outputs (S1);
    if (flag)
      /* Some state variables were set */
      return (BACKTRACK);
    flag2 = stg_enumerate(S'1);
    if (flag2 == BACKTRACK) {
      (S'1, flag) = set_more_inputs(S1);
      if (flag)
        /* Setting more inputs does not help */
        return (BACKTRACK);
      else {
        flag2 = stg_enumerate(S'1);
        if (flag2)
          return (BACKTRACK);
      }
    }
    Reassign last variable on decision tree a different value;
  }
  return (TRUE);
}
```

Figure 4.1: Implicit state-enumeration procedure

particular input and setting it to either a 1 or 0. This new input cube is then concatenated with S_1 and intersected with the cubes in the ON and OFF-set of the PO lines. If the cube

intersects only the ON-set (OFF-set) the value of the corresponding PO line is 1 (0). If the cube intersects both the ON and the OFF-set, then some more inputs are heuristically selected and set to a value so that the new cube intersects only one of the sets. All inputs that are set are stored in a decision tree. The next-state corresponding to this input vector is determined by intersecting the same cube with the ON and OFF-set of every NS line. If the cube intersects only the ON-set (OFF-set) the value of the corresponding NS line is 1 (0), else it is a don't care. The new state S'_1 is the fanout state of S_1 for the input I. This part of the procedure is implemented in the routine **set_outputs**. Having determined S'_1, enumeration proceeds by recursively calling the STG enumeration procedure. For example, consider only the first output for the machine of Figure 4.2. Starting with $S_1 = 000$, and selecting $I = 0-$ sets the PO to 1 and S'_1 to $1-1$. In the next step, starting with $S_1 = 1-1$, setting $I = 1-$ sets the PO to 1 and S'_1 to $01-$. In this process, through two steps of the enumeration procedure, four edges in the actual STG were enumerated.

Since S_1 can be a cube, some PS lines might have to be set in order to set all the outputs to a known value. In doing so, one has to ensure that the resulting state is a valid state. Whenever some PS lines (call them the required state variables) have to be set, the procedure **set_outputs** sets a flag. This initiates backtracking, *i.e.* the procedure returns to a state from where it is possible to set some more *input* variables to set the required state variables. If the required state variables cannot be set without setting some other PS variables, the procedure backtracks one more level and repeats the same step. Ultimately, a state will be reached from which no more backtracking would be necessary. Forward enumeration would then proceed from that state. Consider the example of Figure 4.2, this time with both outputs. Starting with $S_1 = 000$, and selecting $I = 0-$ sets the POs to 10 and S'_1 to $1-1$. In the next step, with $S_1 = 1-1$, setting $I = 1-$ sets the POs to 10 and S'_1 to $01-$. Now that $S_1 = 01-$, the second output cannot be set for the input $I = 1-$ without setting the third state variable. The procedure backtracks to the previous level where $S_1 = 1-1$, and tries to set the third state variable in S'_1 by setting more inputs. But, to do that the second state variable in S_1 would have to be set. Thus, the procedure backtracks one more level where $S_1 = 000$, sets the input to $I = 00$, thereby getting the fanout state to be $S'_1 = 111$. Forward enumeration can proceed from this new S'_1.

This procedure does not require all the next-state lines to be set to a known value. Thus the fanout state of S_1 could be a cube and the state space is implicitly enumerated. All possible fanout edges from a state are enumerated by changing values of input variables

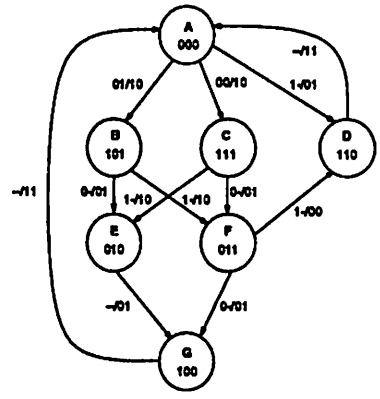


Figure 4.2: An example STG

in the decision tree, as shown in Figure 4.1.

In the final STG, all cube states contain minterm states that are equivalent. For example, consider Figure 4.2. The STG corresponding to the first output is shown in Figure 4.3. The states (101, 111), (010, 011), and (100, 110) are equivalent. The enumeration procedure detects this equivalence and produces an STG with only 4 states and 6 edges. Minterm enumeration of the state space would have resulted in an STG with 7 states and 12 edges. The resultant STG is therefore appropriately called an Implicit State Transition Graph (ISTG).

ISTGs have two important characteristics. Firstly, some equivalent states are merged into one state. Secondly, some transitions from two different states to the same next state for a given primary input combination can be coalesced. This implies that some minimization of the state space and some minimization of the symbolic cover is carried out *during* the enumeration. The number of states and edges enumerated are reduced, thereby speeding up the enumeration and producing more compact STGs.

The enumeration procedure uses cubes to represent transitions between states. For certain circuits (ones involving arithmetic functions), using Binary Decision Diagrams (BDDs) is more practical [25]. In one possible representation, a set of states is represented as one BDD and all the edges between any pair of states as another single BDD. For machines with a large number of primary inputs but a relatively smaller number of states, this BDD based enumeration approach can generate a very compact STG.

I	PS	NS	O
0–	000	1 – 1	1
1–	1 – 1	01–	1
––	01–	1 – 0	0
––	1 – 0	000	1
0–	1 – 1	01–	0
1–	000	110	0

Figure 4.3: STG for first output of Figure 5.2

4.2.2 Implicit State-Traversal

The traversal of a STG differs from its enumeration in that traversal does not require explicit knowledge of the connectivity of states. STG traversal is typically carried out to ascertain whether a certain property is satisfied by the STG. The problem of verifying the equivalence of two FSMs can be viewed as one of *traversing* the STG of the product of the two FSMs with the output of the product being the XNOR of the output of each FSM. The property to be ascertained is whether the output on each edge in the product STG is 1. A traversal algorithm for FSM verification using such an approach was presented in [28]. The intent of this section is to show that the enumeration algorithm of the previous section can be modified to traverse the STG of a FSM. This modification is illustrated with an example. Consider the machine of Figure 4.2. Starting with $S_1 = 000$, the part of the STG enumerated before a PS variable is required to be set is the following :

0–	000	1 – 1	10
0–	1 – 1	01–	10
0–	01–	100	01
––	100	000	11

The edge for the input $I = 0-$ has been enumerated from state 01–, but on setting the input $I = 1-$ (as one would do in the next step of depth-first traversal), the output cannot be set without setting the third state variable. Instead of backtracking, as in enumeration, in this situation one can proceed forward. The third state variable is set to a value, and checked to ensure that the resulting state is reachable from $1 - 1$. The reachability check is performed using the justification procedure of [50]. Note that unlike in STG enumeration, the edges from states 010 and 011 for input 0– are coalesced into one edge. Also, it was not necessary to split cube state $1 - 1$.

4.3 Optimization Strategies

The optimization strategy proposed here begins from a logic-level description of a sequential circuit. An attempt is made to extract the ISTG of the entire circuit for subsequent re-encoding and re-decomposition. Depending on the size of the ISTG, *different* options are exercised in the synthesis procedure.

Small number of states, small number of edges: If the ISTG of the entire circuit has a small number of non-equivalent states ($< 1,000$) and a small number of edges ($< 10,000$), then existing decomposition or state assignment programs [7, 36, 106] that operate on the ISTG can be used for the global optimization of the circuit. It may happen that only certain outputs in the circuit have small ISTGs. In that case, the alternate strategies given below have to be followed for the remaining outputs.

Small number of states, large number of edges: If the ISTG of a particular output has a small number of non-equivalent states ($< 1,000$), but a large number of edges ($\geq$ 10,000), then it is not efficient to use decomposition programs like those in [7] and state assignment programs like NOVA [106]. In this case, a dynamic state assignment strategy that does not require storage of the entire ISTG is used. One pass is made through the extraction process to determine the set of symbolic states in the circuit without storing the edges in the ISTG in the process. In the second pass, given the symbolic state information, the edges in the ISTG are inspected, one by one, to determine a good adjacency-based coding for the states, much like the *counting algorithms* in MUSTANG [36]. Again, one does not require the storage of the entire set of edges. Once a new encoding for the states has been constructed, an encoder and a decoder is added to the sequential circuit. For instance, a state originally may have the code $10 - 1 - 1$ and may be assigned the code 0000. The combinational logic (that now includes the encoder and decoder) of the sequential circuit is then optimized for area or performance using programs like MIS-II [15], potentially leading to an improved implementation.

Large number of states, large number of edges: In this case, the number of the states in the ISTG to be manipulated is reduced by attempting to find a good parallel, cascade, or general decomposition of the circuit, in that order, by suitably choosing subsets of latches that would form each component machine. Selecting any subset of latches corresponds to identifying a submachine in some general decomposition. It is of interest to select a subset of latches and outputs such that the submachines that are so created interact minimally.

Each of these submachines would have much fewer states than the original FSM, making it possible to apply some of the techniques in Options (1) and (2) above for re-encoding.

4.3.1 Synthesis Results Using Implicit State-Enumeration

Some of the significant synthesis results are shown in Tables 4.1 and 4.2. The relative size of the STG obtained via implicit state-enumeration to the size of the STG obtained via explicit state enumeration is compared in Table 4.1. Data on the area and performance improvements obtained as a consequence of sequential synthesis from a symbolic description of the circuit is provided in Table 4.2.

In Tables 4.1 and 4.2, **#I** is the number of primary inputs, **#O** the number of primary outputs, **#L** the total number of latches, and **#G** the number of gates. **O/P #** indicates the output(s) being optimized. The following experiment was carried out: Given a circuit, outputs were chosen as candidates for optimization. Using the algorithm described in previous sections, the ISTG was extracted for these outputs. A new implementation was then obtained for these outputs by performing state encooding and logic optimization. This new implementation was compared against the original implementation of the logic feeding just the selected outputs. For area comparisons, the new and old implementations were both optimized using area optimization techniques in MIS-II [15]. Similarly, for delay comparisons, the new and old implementations were both optimized using delay optimization techniques in MIS-II. The area was measured using the factored-form literal count in MIS-II, and the delays were measured using the "mapped" delay model in MIS-II.

The first example is a set of FSMs forming the controller of the **Viterbi processor** [103]. As it happens, the description for the controller contains three parallel FSMs asserting different sets of outputs, but which are driven by the same set of primary inputs. After optimization however, this decomposition could not be identified via a topological analysis. STGs of none of the outputs could be obtained using the program of [35]. However, the ISTG for each output was obtained using the techniques described in Section 4.2. Extracting the ISTG for any particular output, takes advantage of the inherent parallel decomposition, *i.e.* the fact that that output is functionally dependent only on a subset of the latches, and therefore, all the states with the same value in this subset of latches are equivalent as far as this output is concerned. In a sense, the ISTG extraction method attempts to extract the minimal amount of symbolic information necessary to be able to

CKT	#I	#O	#L	#G	O/P #	Size ISTG		Size STG	
						States	Edges	States	Edges
viterbi	11	34	12	227	0-3	15	48	359	8421
					4-7	8	159	136	6464
sbc	35	51	33	1011	11-12[1]	6	70	*	*
					27-32	1	1	*	*
					48[1]	6	190	*	*
tlc	3	6	21	162	0-5[1]	34	865	*	*
s344	9	11	15	160	4	258	569	*	*

'*' Conventional STG could not be obtained. [1] Incorporating latch selection heuristics.

Table 4.1: Results using implicit state enumeration

optimize that particular output. In an initial pass, ISTGs were extracted for each output separately. After this pass, the outputs that were dependent on the same set of latches (the outputs that are asserted by the same component FSM in the parallel decomposition) were clustered together and a single ISTG extracted for each such cluster.

The next two examples are large FSM controllers, **sbc** and **tlc**. Conventional STG descriptions for almost all outputs (considered individually) of either of these controllers are not obtainable. Manipulable ISTG descriptions, on the other hand, could be obtained for most of these outputs. In addition, the number of outputs for which manipulable ISTG descriptions could be extracted was further increased by only re-encoding an appropriate subset of latches. When it is required to re-encode a subset of latches, only the symbolic information corresponding to that subset need be extracted. The latches not in the chosen subset can be considered to be primary inputs.

Both the examples have inherent two-way cascade decompositions that can be identified after cover extraction, but not via a topological analysis. After exploiting the cascade decomposition in the case of **sbc**, about two-thirds of the outputs had a moderate number of states and edges and did not require further latch selection. However, the remaining outputs either had too large a number of edges or states or both. In the case of **tlc**, once the inherent cascade decomposition had been identified, all the outputs could easily be re-encoded using ISTGs. The final example, **s344**, is from the ISCAS Sequential Testing Benchmark set.

Significant area and performance improvements were obtained on **Viterbi** and

CKT	#I	#O	#L	#G	O/P #	Delay[2] Init./Fin./Impr.	Area (literals) Init./Fin./Impr.
viterbi	11	34	12	227	0-3	32/15.7/51%	321/69/78%
					4-7	30.5/23.9/22%	319/85/73%
sbc	35	51	33	1011	11-12[1]	11.8/10.5/11%	632/538/15%
					27-32	19.5/0/100%	397/0/100%
					48[1]	12.4/9.6/23%	630/550/13%
tlc	3	6	21	162	0-5[1]	18.5/18.5/0%	161/161/0%
s344	9	11	15	160	4	20.7/20.7/0%	131/131/0%

[1] Incorporating latch selection heuristics. [2] Using the mapped delay-model in MISII [15].

Table 4.2: Results obtained by synthesizing from ISTGs

sbc. As can be seen from the tables, the conventional STGs could not be obtained for any of the outputs of **sbc** while some of the ISTGs for the same outputs are extremely compact. For instance, for the reset state chosen, some of the outputs (**27-32**) became wires since all states are equivalent. Using the algorithm of [35] to extract a conventional STG for outputs **27-32** would result in a STG with millions of equivalent states. Using state minimization would eventually produce the same reduction, but such a method would require exorbitant amounts of CPU time.

The ISTGs for the examples **tlc** and **s344** are much more compact than the corresponding conventional STGs. However, for these examples re-encoding did not provide any area or performance gain. A possible explanation is that on large ISTGs, encoding programs like MUSTANG [36] and NOVA [106] are either unable to find state assignments that are comparable to the initial literal count, or do not complete in reasonable amounts of CPU time. Experiments with alternate encoding strategies are currently in progress.

4.4 Conclusions

It appears that the size of sequential circuits for which current sequential logic synthesis strategies are viable, can be increased significantly via the use of Implicit State Transition Graphs (ISTGs). The focus here has been on a sum-of-products representation for ISTGs, given that the most mature state assignment and decomposition strategies in use today target and use such a representation. However, the algorithms and ideas presented can be modified to use alternate representations of Boolean functions as a base.

Certain classes of sequential circuits, described usually at the logic-level, notably ALUs interconnected with registers, are not amenable to sequential logic optimization. While algorithms exist that can verify/test such circuits, improving the performance of such circuits via re-encoding or re-decomposition appears improbable. Selection strategies that can focus on the control portions of a logic-level sequential circuit, where the most room for sequential optimization exists, are a must for large, real-life chips.

In particular, the approaches presented appear promising for global, sequential optimization, such as re-encoding and re-decomposition, of interacting FSM controllers, which individually may have compact STG specifications, but whose overall STGs are too large. The overall STG typically has a large number of equivalent states and ISTGs allow for efficient synthesis.

Chapter 5

Irredundant Interacting Sequential Circuits

5.1 Introduction

The relation between logic synthesis and test generation is strong and is the focus of recent research [37]. One relationship hinges on the simple observation that *redundancy* in a logic circuit, corresponding to the absence of a test for an associated fault in the circuit, is typically a product of a sub-optimal logic synthesis step. In the case of stuck-at fault redundancies for instance, a redundant faulty line can be "replaced" with a constant 1 or 0, thereby reducing area without altering functionality. Thus, a "perfect" set of optimization steps can preclude the occurrence of redundancies in a synthesized logic-level implementation of a combinational or sequential circuit.

The relationship between combinational logic synthesis and combinational test generation has been investigated previously [10, 14]. The work of [10] showed an intimate relationship between don't-care conditions in a combinational network and stuck-at fault redundancies in the circuit. An optimal synthesis procedure that exploited the *complete* don't-care set, and which resulted in a fully testable combinational circuit was presented.

Sequential circuits are more complicated than combinational circuits. Sequential logic synthesis includes the steps of state minimization, state assignment, retiming and combinational logic optimization. Stuck-at faults in a non-scan sequential circuit may be testable from a combinational point of view, but may be redundant in the non-scan circuit

due to limited observability and controllability of the memory elements. These faults are termed sequentially redundant.

Constrained synthesis procedures that result in fully and easily testable finite-state machines (FSMs) were presented in [38]. These procedures involve the addition of extra logic to produce an easily testable machine. A different approach, taken in [39], defines a complete don't-care set for the synthesis of FSMs represented by a *single* State Transition Graph (STG) and provides an optimal synthesis procedure that results in a *fully testable* sequential machine.

The work in [39] was restricted to single finite-state machines. Industrial chip designs are typically composed of interacting finite-state machines. The types of possible redundancies in interacting sequential circuits and their associated don't-care sets are more complicated than in the single machine case.

The classification of sequential redundancies in interacting sequential circuits and their removal is the focus of this chapter. An intuitively obvious classification of sequential redundancies in interacting sequential circuits is based on the limited controllability and observability of the intermediate lines by means of which the component FSMs communicate. In this chapter, a relationship is shown between *sequential don't-cares* corresponding to sequences of vectors that never occur at the inputs (outputs) of the driven (driving) FSM, and certain classes of sequential redundancies. Methods of exploiting sequential don't-cares in cascaded circuits for area optimization were presented in [30]. These are extended for application to testability-driven synthesis.

Subsequently, a specific, but frequently occurring class of interacting FSMs is considered, where the individual FSMs communicate solely via their present states. For this class of circuits, efficient algorithms that detect unreachability of states and edges, and compatibility between states are devised. Further, it is shown that a uniform treatment of arbitrary topologies of interacting FSMs is possible for such circuits.

Finally, experimental results are presented which indicate that these procedures can synthesize *fully testable* medium-sized sequential circuits in reasonable CPU time.

5.1.1 Organization of the Chapter

The classification of redundancies in single finite-state machines is reviewed in Section 5.2. The classification of sequential redundancies based on the limited controllabil-

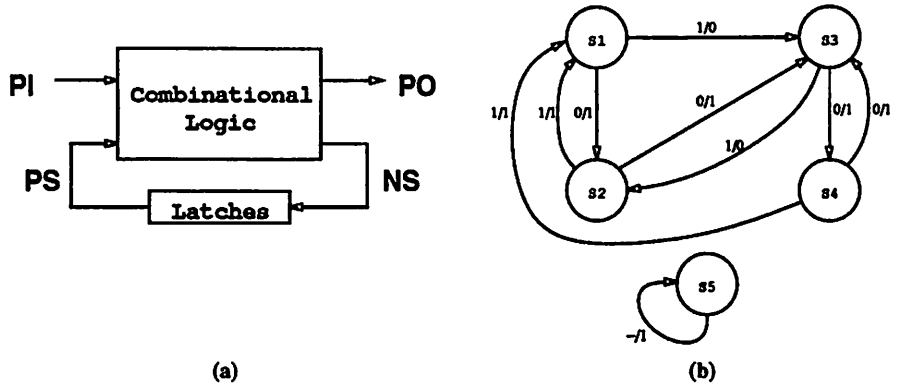


Figure 5.1: A sequential circuit

ity and observability of the intermediate lines is presented in Section 5.3, and a synthesis procedure that produces irredundant cascades is developed. The generalization of this synthesis procedure to multiple interacting FSMs is explored in Section 5.4. In Section 5.5, efficient procedures for detecting unreachable states and compatible states in interacting FSMs that communicate via their present state lines are presented. Based on these procedures, an algorithm for the synthesis of irredundant multiple interacting FSMs is presented. Experimental results are presented in Section 5.6.

5.2 A Review of Redundancies in Single Finite-State Machines

A sequential circuit M, comprised of a single FSM is shown in Figure 5.1(a). The State Transition Graph corresponding to the circuit is shown in Figure 5.1(b). Redundant faults in M may be combinationally redundant (CRFs) or sequentially redundant (SRFs). SRFs can be classified into three categories.

1. **Equivalent-SRF**: The fault causes the interchange/creation [1] of equivalent states in the STG.

2. **Invalid-SRF**: The fault does not corrupt any fanout edge of a valid state in the STG.

3. **Isomorph-SRF**: The fault results in a faulty machine that is isomorphic (*i.e.* a machine which is different only w.r.t. the encoding of states) to the original machine.

[1] Replacement is included as a form of interchange.

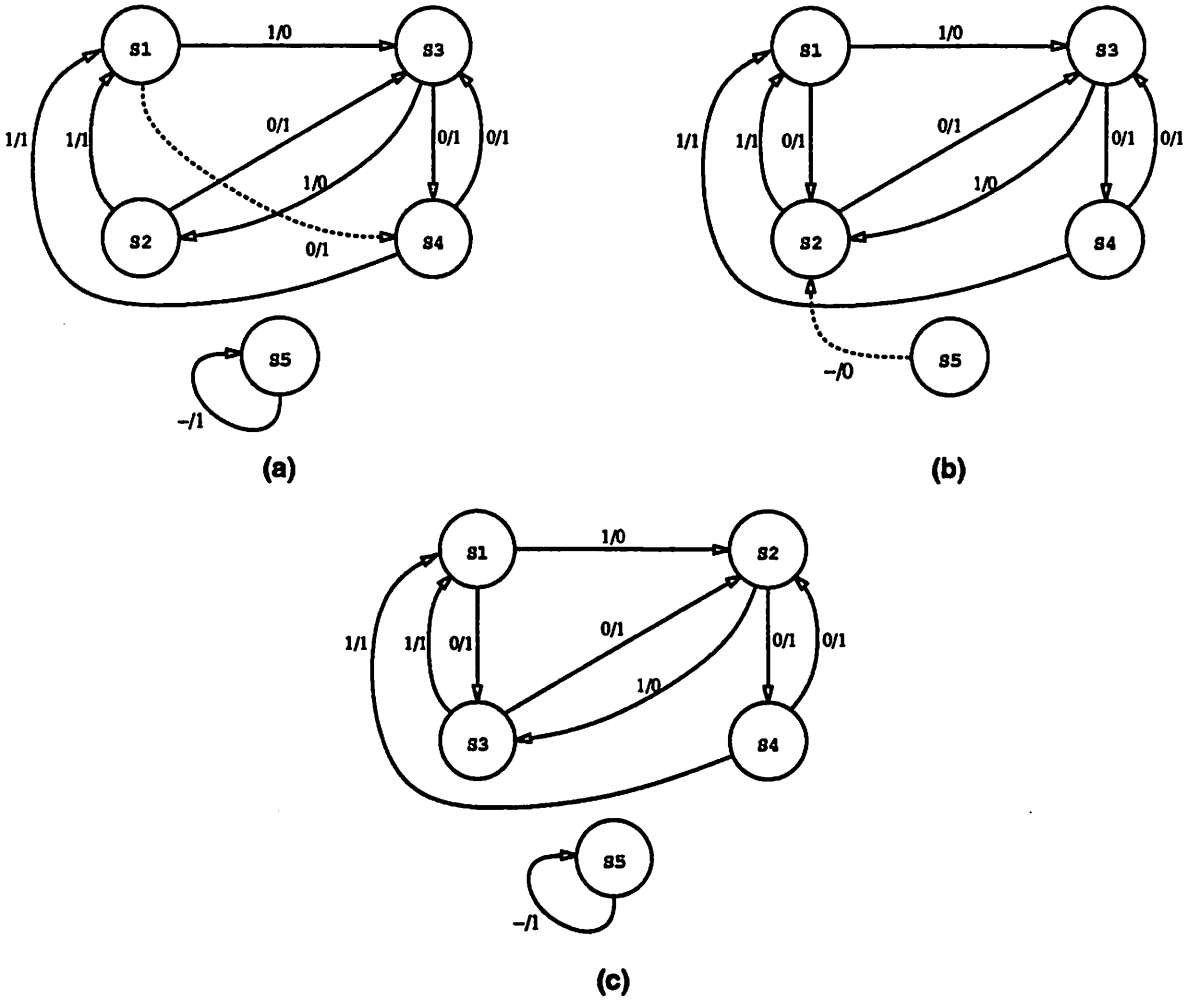


Figure 5.2: Three types of sequential redundancies

It is shown in [39] that no other kind of sequential redundancy can exist.

In Figure 5.1(b), states $s2$ and $s4$ are equivalent states. An equivalent-SRF in M may produce the faulty STG of Figure 5.2(a), where the only corrupted edge (shown in dotted lines) goes to $s4$ instead of $s2$ and does not change the terminal behavior of M. A faulty STG corresponding to an invalid-SRF is shown in Figure 5.2(b). As can be seen, only fanout edges from the invalid state $s5$ have been corrupted by the invalid-SRF. In Figure 5.2(c), an isomorphic faulty machine which is equivalent to the true machine in all respects but for the interchange of states $s2$ and $s3$, is shown.

5.2.1 Eliminating Isomorph-SRFs

It was shown in [39] that stuck-at faults in a sequential machine implemented by a two-level combinational network cannot cause isomorphism. On the other hand, for a sequential machine implemented by a multilevel network, stuck-at faults could conceivably produce an isomorphic faulty STG. There are many ways of ensuring that isomorphism does not occur in multilevel networks. Isomorphism due to a stuck-at fault is caused by a *sub-optimal state assignment.* The faulty STG has a different encoding from the true STG such that the new encoding corresponding to the isomorph/faulty STG represents a better machine, *i.e.* one with the redundant line removed and hence with fewer literals. Therefore, if a fault F in a sequential machine, M, causes isomorphism between any set of states $Q \in S_M$, then we can find an encoding that results in a machine M' identical in functionality to M, where the combinational logic of M' has at least one less literal than M.

A locally optimal state assignment across any given set of states can ensure that isomorphism does not occur in multilevel circuits, across this set of states. If the assignment is locally optimal across a set of states, it implies that the exchange of two or more state codes within this set of states cannot produce a better logic implementation with fewer lines. This in turn means that no fault can cause isomorphism within this set of states. It is worthwhile to note that optimal state assignment corresponds to the optimal usage of the don't-care that the actual code assigned to a particular state is not important as long as that code is distinct from the codes assigned to other states.

As illustrated in [39] isomorphism can also be prevented by implementing the logic for the FSM in two-level form or in algebraically factored multilevel form.

5.2.2 Eliminating Invalid-SRFs

The codes corresponding to invalid states can be used as don't-cares for all primary input combinations during logic optimization. An invalid-SRF arises due to the sub-optimal usage of these don't-cares. Making the combinational logic prime and irredundant under this don't-care set ensures that invalid-SRFs will not exist. Primality and irredundancy under a don't-care set guarantees the existence of a test vector *outside* the don't-care set for every single stuck-at fault [10]. In the context of sequential circuits, this property implies that there always exists a *valid state* that can propagate the effect of the fault to the next

state or primary output lines if the combinational logic is prime and irredundant under the invalid state don't-care set.

Theorem 5.2.1 : *An invalid state in a State Transition Graph of a machine M is never required to detect a fault if the combinational logic of M is made prime and irredundant under the invalid state don't-care set.*

Proof. See Lemma 4.3 and Theorem 4.5 of [39]. □

5.2.3 Eliminating Equivalent-SRFs

Equivalent-SRFs are related to redundant states in a sequential machine. Given a reduced machine, a fault that corrupts a single edge into going to a faulty, but valid, state cannot be redundant, since all states are differentiable. Thus, an initial state minimization will preclude the occurrence of an SRF of the form shown in Figure 5.2(a). However, there may be a case where the fault results in a faulty *invalid* next state that is equivalent to the true next state. This is illustrated in Figure 5.3. Shown in Figure 5.3(a) is the state-minimal STG of a fault-free machine which has been optimized under the invalid state don't-care set (*cf* Section 5.2.2). The code for the invalid state $s4$ is used as a don't-care and consequently, $s4$ becomes equivalent to state $s2$ after logic minimization under this don't-care condition. A fault could result in the scenario shown in Figure 5.3(b), where a single corrupted edge whose true next state is $s2$ produces a faulty invalid next state, $s4$. The fault is redundant because $s4$ is equivalent to $s2$. The reason this redundancy exists is that the don't-care condition corresponding to the next state that the edge (0, $s3$) can fan out to has not been exploited. Since states $s4$ and $s2$ are equivalent, one can specify $n(0, s3) = (s4, s2)$ and not just $s2$. The next state that the edge (0, $s3$) actually fans out to is determined during logic optimization. Assume, for instance, that the code of $s4$ is 10 and the code for $s2$ is 11. One can specify $n(0, s3) = (10, 11) = 1-$. In the sequel, such don't-cares will be referred to as extended don't-cares.

The following procedure of repeated logic minimization (modified from [39]) guarantees upon convergence that invalid-SRFs don't exist and that single edge corrupting and certain kinds of multiple edge corrupting equivalent-SRFs don't exist. While the elimination of all possible equivalent-SRFs can be guaranteed by the use of extended don't-cares at Step **A**, these don't-cares are *not* required in practice to produce irredundant machines.

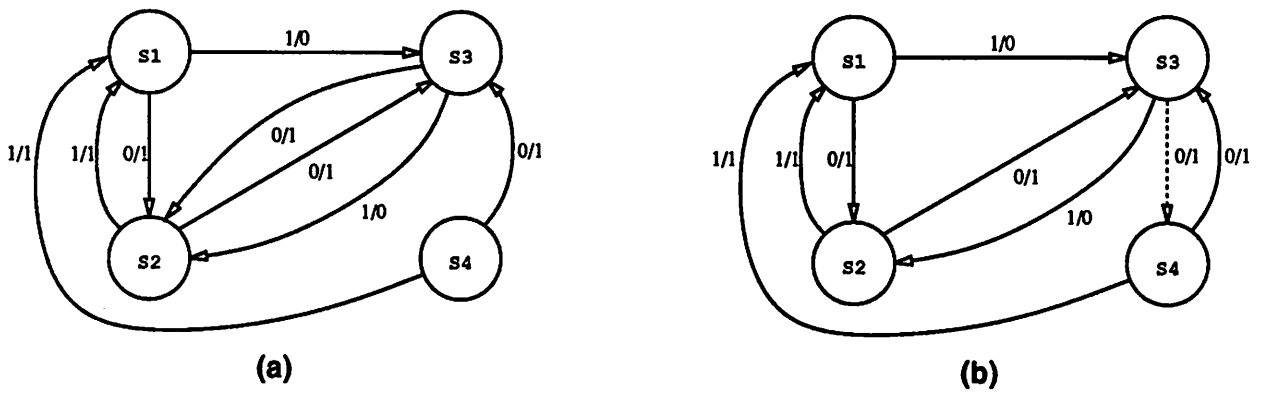


Figure 5.3: An equivalent-SRF

In the following procedure, the routine **extract-stg**() extracts the entire State Transition Graph from a logic-level implementation of a FSM. The routine **optimize**() produces a prime and irredundant multilevel realization under a given don't-care set. The routine **extract-invalid-states**() finds all the invalid states in a given logic-level implementation of a FSM. FA^{DC} is the fanin don't-care set that corresponds to equivalent states, q, v_1, .. v_x, iv_1, .. iv_y. The line corresponding to Step **A** specifies a degree of freedom (don't-care condition) in that the fanin edges to q can be moved to any of the above states without altering functionality. This degree of freedom cannot always be represented by don't-cares. In general, Boolean relations [21] might be required to exploit it. IV^{DC} is the invalid state don't-care set. The optimization under IV^{DC} is carried out separately from the optimization under FA^{DC} because the set of invalid states is known only after FA^{DC} has been used. The combinational logic of M' is made prime and irredundant under this don't-care set to produce M''.

eliminate-equivalent/invalid-SRFs(M):
{
    $iter = 1$;
    do {
        if ($iter = 1$) G = **extract-stg**(M) ;
        else G = **extract-stg**(M'') ;
        $FA^{DC} = \Phi$ /* FA^{DC} is empty initially */
        foreach (valid state $q \in G$) {
            Find all valid states $(v_1, .. v_x) \equiv q$;
            Find all invalid states $(iv_1, .. iv_y) \equiv q$;
            A: $FA^{DC} = FA^{DC} \mid fanin(q) = (q, v_1, .. v_x, iv_1, .. iv_y)$;
        }

```
        if ( iter ≠ 1 ) M = M'' ;
        M' = optimize( M, FA^DC ) ;
        IV^DC = extract-invalid-states( M' ) ;
        M'' = optimize( M', IV^DC ) ;
        iter = iter + 1 ;
    } while( M ≠ M'' ) ;
}
```

It can be proved that state minimization, a locally optimal state assignment and the procedure **eliminate-equivalent/invalid-SRFs()** produces an irredundant sequential machine (*cf* Theorem 4.7 in [39]).

5.3 Controllability and Observability Based Synthesis

The classification of redundancies presented in this section is based on two fundamental premises. The first is that cascaded pairs of FSMs can be considered as building blocks of arbitrary interacting sequential circuits. For example, the interacting sequential circuit shown in Figure 5.4 can be considered to be composed of the cascade pairs $A \longrightarrow B$, $B \longrightarrow C$ and $C \longrightarrow A$. The problem of removing redundancies from arbitrary sequential circuits can then be formulated as one of iteratively removing redundancies from the constituent cascade pairs (*cf* Section 5.4). Based on this first premise, it is only necessary to classify sequential redundancies for cascaded pairs of FSMs. The second premise is that sequential redundancies in a cascade of two FSMs can be fully classified based on the lack of controllability and observability by the two FSMs of the intermediate lines by means of which they communicate.

In the following paragraphs, a classification of sequential redundancies in a cascade of two FSMs based on controllability and observability criteria is presented. Given this classification, don't-care sets are associated with each of these forms of redundancy and give a synthesis procedure that produces an irredundant cascade. In Section 5.4, the generalization to multiple interacting circuits is described briefly.

5.3.1 Redundancies in a Cascade

Consider the cascade $A \longrightarrow B$ in Figure 5.4. A drives B via a set of latches $L1$. For the purposes of the discussion here, it is assumed that none of the flip-flops in $L1$ are

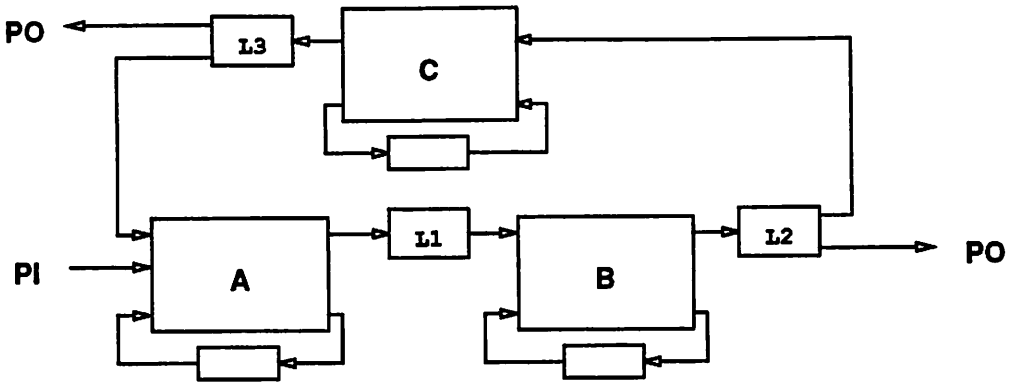


Figure 5.4: Interacting finite-state machines

directly observable or controllable. In practice, a subset of the flip-flops may be observable or controllable. B may receive primary inputs and/or A may assert primary outputs.

Redundancies in a cascade $A \longrightarrow B$ can be classified into four categories. INT denotes the intermediate lines and F denotes a single stuck-at fault.

1. $F \in A$ that cannot propagate to the intermediate lines INT.
2. $F \in A$ that propagates to INT but not PO, the primary outputs of B.
3. $F \in B$ that does not propagate to PO, but would have if INT were completely controllable.
4. $F \in B$ that does not propagate to PO and would not have even if INT were completely controllable.

Theorem 5.3.1 : *Redundancies of Type 1, 2, 3 and 4 form a complete classification of redundancies in a cascade* $A \longrightarrow B$.

Proof. Assume that $F \in A$ is redundant. Either F can be propagated to the outputs of A, namely INT, or F cannot be propagated to INT. In the latter case, F is a redundancy of Type 1. In the former case, F cannot be propagated to PO, else F is testable. Therefore, F is a redundancy of Type 2.

Similarly, assume that $F \in B$ is redundant. There are two possible cases; F becomes testable if INT were made completely controllable or F is still redundant after INT is made completely controllable. These two cases correspond to redundancies of Type 3 and 4, respectively. □

It is easy to see that redundancies of Type 1 and 4 are associated with the single machines A and B. If A and B are irredundant, these redundancies will not appear in a cascade $A \longrightarrow B$.

5.3.2 Exploiting Don't-Care Inputs for the Driven Machine

There are several don't-care conditions associated with the intermediate lines corresponding to $L1$, which are inputs to B. Let the number of intermediate/pipeline flip-flops in $L1$ be N.

1. A may or may not assert all 2^N possible output combinations. If a certain binary combination, c_1 never appears at $L1$, then B can be made incompletely specified – the transition edges corresponding to an input of c_1 need not be specified, for any state of B. (It doesn't matter what happens when B receives the input c_1)

2. A more general case of (1) is when a certain combination c_2 never appears at $L1$, when B is in some set of states $Q_B \in S_B$. It does appear when B is in states other than Q_B. In this case, the states in Q_B will have c_2 unspecified. (If an edge on c_2 exists in Q_B, it can be removed)

3. A more complicated sequential don't-care is associated with vector *sequences* that never appear at $L1$. A does not produce all possible output sequences. This type of don't-care does *not* have a straightforward interpretation. Unlike in the case of (1) or (2), this don't-care cannot be accounted for directly by leaving edges in the STG of B unspecified.

Both (1) and (2) can be easily exploited via the use of standard state minimization algorithms that handle incompletely specified machines [89]. However, exploiting don't-care input *sequences* is more complicated.

In Figure 5.5, a cascade $A \longrightarrow B$ is shown in STG form. Each of the machines is state minimal. Assuming that the starting states of the machines are $q1$ and $s1$ respectively, each transition edge in B is irredundant, *i.e.* B makes every transition with appropriate input sequences. However, A does not assert all possible output sequences. A don't-care input sequence is shown below the graph of B. Such a don't-care sequence implies that

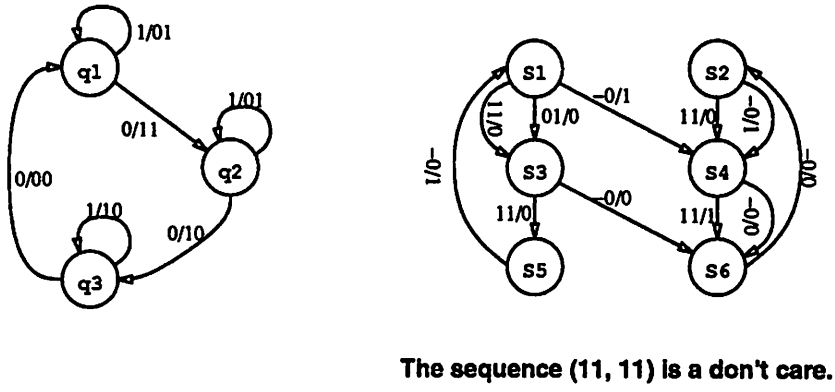


Figure 5.5: Input don't-care sequences

certain *sequences of transitions* will not be made by B. In the general case, there may exist a set of don't-care sequences.

Unconditional Compatibility

Definition 5.3.1 : *If all the differentiating sequences corresponding to a pair of states are contained within a don't-care sequence set, then these states are deemed to be unconditionally compatible under the don't care set.*

The following procedure describes a general approach for identifying unconditional compatibilities and unspecified edges:

exploit-input-dc(B**):**
{
LOOP1: **foreach(** state $s_b \in S_B$ **)** {
Find Q_A, the set of states A can be in when B is in s_b ;
foreach(edge in B, $e_b = (i_b, s_b)$ **)** {
foreach($q_a \in Q_A$ **)** {
if($\exists\, i_a \mid o(i_a, q_a) = i_b$ **)** go on to next edge e_b ;
}
}
Delete edge e_b ;
}
LOOP2: **foreach(** state pair $s_1, s_2 \in S_B$ **)** {
Find Q_{A1}, the set of states A can be in when B is in s_1 ;
Find Q_{A2}, the set of states A can be in when B is in s_2 ;
$Q_{A12} = Q_{A1} \cup Q_{A2}$;
foreach(differentiating sequence ds_{12} of s_1, s_2 **)** {
if (ds_{12} can be produced starting from some state $q_a \in Q_{A12}$ **)** {

```
                $s_1$ and $s_2$ are not unconditionally compatible ;
                Go on to next state-pair ;
            }
        }
        $s_1$ and $s_2$ are unconditionally compatible ;
    }
}
```

Theorem 5.3.2 : *The procedure* **exploit-input-dc***() finds all unspecified edges in machine B and all unconditional compatibilities between states in machine B arising from don't-care input sequences.*

Proof. Assume that Machine B is in state s_b. Machine A can be in the set of states Q_A when Machine B is in s_b. An edge, corresponding to input i_b, that fans out of state s_b in Machine B is unspecified if A never produces the output i_b from the states in Q_A. LOOP1 of **exploit-input-dc**() checks if the output i_b can be produced by Machine A from the states in Q_A. Since this check is performed for all edges that fan out of each state in Machine B, all unspecified edges in Machine B are identified.

Two states in Machine B are unconditionally compatible if none of the differentiating sequences for the two states can be produced by Machine A. For each pair of states s_1 and s_2 in Machine B, LOOP2 of **exploit-input-dc**() checks if the differentiating sequence for s_1 and s_2 can be produced by Machine A. When no differentiating sequence can be produced, s_1 and s_2 are labeled unconditionally compatible. Since this check is performed for all pairs of states in Machine B, all unconditional compatibilities are identified. □

In the example of Figure 5.5, states $s1$ and $s2$ are unconditionally compatible and can therefore be merged under the don't-care set. One approach to exploiting input don't-cares is to produce all differentiating sequences for every pair of states in B and checking for containment in the input don't-care set. Pairs satisfying the containment condition can be merged. Given a cascade, one needs to generate the set of sequences that the driving machine in a cascade $A \longrightarrow B$ never asserts, so as to optimize the driven machine B. This is done by generating don't-care sequences of increasing length, beginning from a length of 2. Starting from each valid state in A, all possible two-vector sequences are found. Single vectors that don't occur are added to this set and the set is "complemented" to find the two-vector sequences that don't occur. Next, all sequences of length 3 that A asserts are

found. The single-vector and two-vector don't-care sequences that are produced by A are added to this set and the union is complemented to find the don't-care sequences of length 3 and so on, until no more don't-care sequences are found. It may be that a sequence of vectors cannot be applied to B when B is in a particular set of states alone. In which case, this sequence is a don't-care sequence for that particular set of states in B. This information also can be extracted from the driving machine A and exploited to reduce the number of states in B. Sequences can be stored as BDDs. Assume that width of each vector in a sequence is w bits. If the longest sequence has a length of l vectors, the BDD used to store the sequences has $w.l$ inputs. The complement operations described above can then be performed on this BDD.

The procedure described in the above paragraph has obvious limitations. Primarily, a pair of states may have a large number of differentiating sequences rendering this strategy time consuming. In the worst case, every edge in the product of the driving and driven machines has to be traversed. An efficient approach for minimizing the driven machine taking advantage of unconditional compatibilities arising from input don't-care sequences was described by Kim and Newborn [66]. The following procedure is followed in that approach: In the first step, a machine, say M_A, that accepts just the output sequences that the driving machine can produce is constructed. The output on an edge in M_A is a 1 when the input sequence applied so far is a sequence that M_A accepts. The output is a 0 otherwise. In general, M_A has fewer states than the driving machine. A product machine consisting of M_A and the driven machine is constructed such that the output of the product machine is the Boolean AND of the outputs of the driven machine and M_A. Since M_A has fewer states than the driving machine, the product of M_A and the driven machine can be much smaller than the product of the driving machine and the driven machine. It can be shown that a state minimization of the product machine realizes that same machine that would have been realized by minimizing the driven machine under the complete don't-care set corresponding to the unconditional compatibilities. This formulation of the minimization problem for the driven machine does not require that the differentiating sequences in the driven machine be computed explicitly and stored. Also, determining M_A is much easier than finding all the sequences that the driving machine never produces.

Conditional Compatibility

While the procedure **exploit-input-dc()** finds all unconditional state compatibilities under an input don't-care sequence set, it does not take into account *conditional compatibility* between states in a driven machine.

Definition 5.3.2 : *A state q_1 in a driven machine B of a cascade $A \longrightarrow B$ is conditionally compatible to another state $q_2 \in S_B$ under a don't-care sequence set and a set of fanin edges $F_I \subset fanin(q_1)$ if on traversing any edge $e \in F_I$ and reaching q_1, one cannot differentiate between q_1 and q_2 in B.*

Like unconditional compatibility, conditional compatibility is also a product of the constrained controllability of a driven machine B in a cascade $A \longrightarrow B$. It arises because traversing a particular edge in B may imply that certain edges in A have been traversed and may imply a restriction on the set of states A can be in. Therefore, given that this edge has been traversed in B, A may not be able to produce certain output sequences. These output sequences that cannot be produced correspond to an input don't-care sequence set to B which can result in compatibility of states in B *under the condition* that a certain fanin edge is traversed in order to reach the states.

Consider the example in Figure 5.6 which shows a portion of the STG of a machine B that is the driven machine in a cascade. $i1$ and $i2$ are the two possible input symbols to machine B. Assume that the sequence of inputs ($i1$, $i2$) is an input don't-care sequence to machine B. Also assume that the sequence ($i2$, $i2$) can be produced by machine A. It can be seen from Figure 5.6 that the only differentiating vector for $s1$ and $s2$ is the input $i2$. Therefore, if $s1$ is reached via input $i1$, then $s1$ cannot be differentiated from $s2$, since the input $i2$ cannot be produced immediately after $i1$ has been produced. On the other hand, if $s1$ is reached via input $i2$, one can differentiate $s1$ from $s2$ since there is no restriction on $i2$ being produced immediately after $i2$.

Given a set of input don't-care sequences, conditional compatibility between any pair of states can be determined. These compatibilities can then be exploited at the logic level, though, unlike in the case of unconditional compatibility, the number of states in the machine cannot be reduced. In the procedure below @ denotes the concatenation of the vector i_b and the vector sequence ds_{12} to produce a longer vector sequence.

find-cond-compatibility(B, DC **):**

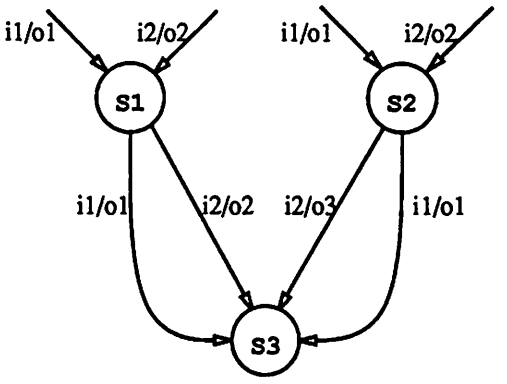


Figure 5.6: Conditional compatibility

```
{
    foreach( state pair s1, s2 ∈ S_B ) {
        foreach ( edge e_b = (i_b, q_b) such that n(i_b, q_b) = s1 or n(i_b, q_b) = s2) {
            foreach( differentiating sequence ds12 of s1, s2 ) {
                if ( DC does not contain i_b @ ds12 ) {
                    s1 and s2 are not conditionally compatible under e_b and DC ;
                    Go on to the next fanin edge e_b ;
                }
            }
        }
        s1 and s2 are conditionally compatible under e_b and DC ;
    }
}
```

Theorem 5.3.3 : *The procedure* **find-cond-compatibility**() *finds all conditional compatibilities between states under a don't-care set DC.*

Proof. Consider two states, s_1 and s_2 in the driven machine. Let the next state on edge e_b (with input i_b and present state q_b) in the driven machine be s_1. The states s_1 and s_2 are conditionally compatible under e_b if no sequence differentiating between s_1 and s_2 can be produced by the driving machine given that the edge e_b has been traversed in the driven machine. That is, s_1 and s_2 are conditionally compatible under e_b if the sequence formed by concatenating i_b and any differentiating sequence for s_1 and s_2 is never produced by the driving machine. The **find-cond-compatibility**() procedure exhaustively goes through all the pairs of states, and through every edge going to one of the two states to check if the concatenation of the edge and some differentiating sequence for s_1 and s_2 is produced by the driving machine. Therefore, the procedure finds all conditionally compatible states. □

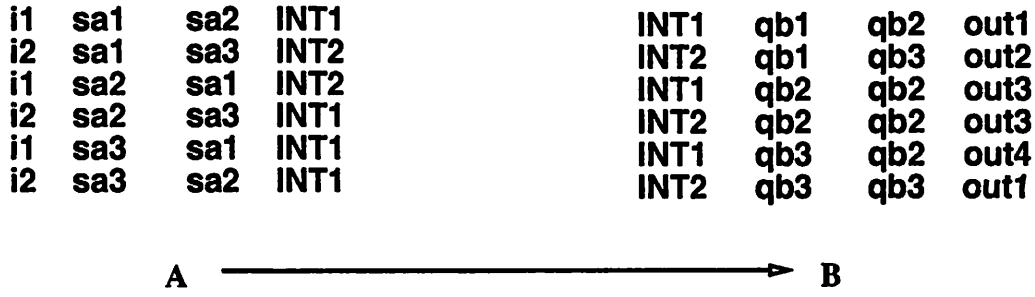


Figure 5.7: Output expansion

5.3.3 Exploiting Don't-Care Outputs for the Driving Machine

The sequential don't-cares discussed thus far are a product of the constrained controllability of the driven machine B in a cascade $A \longrightarrow B$. There is another type of don't-care that arises due to the constrained observability of the driving machine A.

Consider the individually state minimized tables of Figure 5.7. The intermediate inputs/outputs have been given symbolic codes. Given that A feeds into B, it is quite possible that for some transition edge $e_a \in A$, it does not matter if the output asserted by this particular transition edge is, say, $INTi$ or $INTj$. In fact, in Figure 5.7, the 3rd transition edge in A can be either $INT1$ or $INT2$, *without changing the terminal behavior of* $A \longrightarrow B$. This is a don't-care condition, called the *single-vector output don't care*, on A's outputs. It is quite possible that making use of these don't-cares can reduce the number of states in A. In fact, if one replaced the output of the 3rd edge in A (Figure 5.7) by $INT1$ instead of $INT2$, one less state would be obtained after state minimization. (**sa2** becomes compatible to **sa3**).

A systematic procedure to detect this type of don't-care, proposed in [30], is described below. In this procedure, the output of each transition edge of A is expanded to the set of all possible values that it can take while maintaining the terminal behavior of $A \longrightarrow B$. A transition edge e_a in A is chosen. The set of states that B can be in when A makes this transition is found. Given this set of states, the largest cube (or set of output combinations) that covers the output of the edge and produces a unique next state or a compatible set of next states, as well as a unique output when B is in *any* one of the possible

states is found. This set of output combinations is called *expout* in the procedure below. The output of e_a, namely $o(e_a)$, is then expanded to the cube (or set of output combinations). The process is repeated for all edges in A. In the procedure below, $n(c_a, q_b)$ is the next state in B for the present state q_b and all the inputs to B contained in c_a. Similarly, $o(c_a, q_b)$ is the output in B for the present state q_b and all the inputs to B contained in c_a.

```
output-expansion-1( A, B ):
{
    foreach ( edge e_a ∈ A ) {
        expout = universe ;
        foreach ( state q_b ∈ S_B ) {
            if ( B can be in q_b ∈ S_B when A traverses edge e_a ) {
                Find the largest set of output combinations c_a
                such that ( ( c_a contains o(e_a) ) ∧
                ( n(c_a, q_b) is unique or all equivalent states in B ) ∧
                ( o(c_a, q_b) is unique ) ) ;
                expout = expout ∩ c_a ;
            }
        }
        o(e_a) = expout ;
    }
}
```

Theorem 5.3.4 : *The procedure* **output-expansion-1()** *finds all single-vector output don't-cares for machine A, under the specified set of edges for machine B, in a cascade* $A \longrightarrow B$.

Proof. Assume that after using procedure **output-expansion-1()**, there is still an edge $e_a \in A$ that could be expanded to $o(e_a) \cup o_u$, without changing any of the other edges and the functionality of $A \longrightarrow B$. This expansion can only be possible if, for all states that B can be in when e_a is traversed in A, the next state in B on receiving o_u is equivalent to the next state on receiving $o(e_a)$ and o_u makes B produce the same output. But, these are exactly the two conditions that are checked in **output-expansion-1()** for every edge in A. Therefore, $o(e_a)$ will have been expanded to $o(e_a) \cup o_u$ at the end of **output-expansion-1()**. □

5.3.4 State Minimization Under Don't-Care Sets

Classical state minimization algorithms must be enhanced in order to exploit input and output don't-cares. The procedure **exploit-input-dc**() allows the detection of compatibility between states under an input sequence don't-care set. In order to exploit arbitrary output don't-cares, state minimization algorithms have to be enhanced.

The state minimization procedure proposed in [89] can be used for incompletely specified finite-state machines. In the state minimization procedure of [89], two states are compatible if the output combinations that can be asserted by each pair of corresponding fanout edges of the two states intersect. One can envision a situation where the possible output combinations of the fanout edges of q_1, $q_2 \in S_M$ intersect leading to a compatibility relation $q_1 \cong q_2$, with similar compatibility relations $q_2 \cong q_3$ and $q_1 \cong q_3$. However, the three-way intersection between the possible output combinations of the fanout edges of q_1, q_2 and q_3 may be null, implying that q_1, q_2 and q_3 cannot be merged into a single state, even though all the required pairwise compatibility relations exist. Assume that the outputs that can be asserted for some input from states q_1, q_2 and q_3 are (01, 10), (01, 11) and (10, 11), respectively. The intersection of the first and second pairs is non-null, Similarly, the intersection of the first and third, and the second and third pairs is also non-null. But, the intersection of all the three sets of outputs is null. Therefore, even though the relations $q_1 \cong q_2$, $q_1 \cong q_3$, and $q_2 \cong q_3$ could be true, the relation $q_1 \cong q_2 \cong q_3$ cannot be true.

If the possible output combinations can be represented as a single cube, then such a situation will not occur, since the three-way intersection of a set of three cubes has to be non-empty if the pairwise intersections are non-empty [2]. But, in the case of multiple cubes or Boolean expressions specifying the output combinations for fanout edges, an additional check has to be performed during state minimization while selecting the compatibility pairs to see if three or more sets of states can, in fact, be merged, preserving functionality. A similar check has to be performed during logic minimization if one is specifying compatible states as fanin don't-cares for transition edges.

5.3.5 Boolean Relations

In the synthesis of single FSMs, Boolean relations [21] may have to be exploited in order to ensure full testability [39]. For instance, in the procedure **eliminate-equivalent/invalid-**

[2] In the example above, (01, 10) cannot be represented as a cube.

SRFs() of Section 5.2.3, the set of equivalent states, namely, (q, v_1, .. v_m, iv_1, .. iv_n) at Step **A** may not be representable as a single cube, *e.g.* when q = 001 and v_1 = 010.

After using **output-expansion-1**() and performing state minimization, a set of permissible output combinations is obtained for each edge. These permissible output combinations may or may not have a cube representation – in general, they will be Boolean expressions. The combinational logic specification has to be minimized under these Boolean relations to ensure irredundancy. Similarly, the procedures **exploit-input-dc**() and **find-cond-compatibility**() determine unconditional and conditional compatibilities between states, and may result in Boolean relations that have to be exploited in the following combinational logic optimization step (*cf* Section 5.3.7).

5.3.6 Associating Redundancies and Don't-Care Sets

In order to associate redundancies with don't-care sets, it is convenient to further classify the redundancies of Type 2 and 3 that were defined in Section 5.3.1. In the general case, A and B are incompletely specified machines.

Redundancies of Type 2

Redundancies of Type 2 can be classified as:

(a) $F \in A$ produces a $int^F \neq int$ (a faulty output not equal to the true output) for some transition edge $e_a \in A$. Let Q_B be the set of states that B can be in when A traverses edge e_a. int^F is restricted to be an originally specified fanout edge for all the states in Q_B. It is required that int^F move B either to the same next state as int does, or moves B to a valid next state that is compatible to the true next state, produced by int. Therefore, F is redundant.

(b) $F \in A$ produces an unspecified or *invalid* output int^F for the states B can be in and moves B into a valid or an invalid state that is compatible to the true state, resulting in redundancy. B may be moved to an invalid state, since the fanout edge int^F from the state in B was originally unspecified.

(c) A produces a sequence of faulty outputs $int1^F$, .. $intN^F$ instead of $int1$, .. $intN$, such that the first output moves B into an *valid* next state that is *not* compatible to the

true next state, but this state effectively becomes compatible to the true state due to $int2^F$, .. $intN^F$.

(d) A produces a sequence of faulty outputs $int1^F$, .. $intN^F$ instead of $int1$, .. $intN$, such that the first output moves B into an *invalid* next state that is *not* compatible to the true next state, but this state effectively becomes compatible to the true state due to $int2^F$, .. $intN^F$.

Redundancies of Type 3

Redundancies of Type 3 can be classified as:

(a) $F \in B$ requires a transition edge in B that cannot be justified for excitation/propagation to the primary output or next state lines of B. For a fault $F \in B$ to be detected, a particular state $q_b \in S_B$ and a particular INT combination, say int_a may be required. If A can never produce the output int_a when B is in state q_b, F is redundant.

(b) A transition edge that propagates $F \in B$ to the next state lines exists and the faulty state produced is a valid state. The faulty/fault-free state pair in B possess a differentiating sequence (which constitutes part of a test sequence), but are unconditionally compatible (*cf* Definition 5.3.1) under the input don't-care sequence set for machine B.

(c) Same as (b) except that the faulty/fault-free state pair are conditionally compatible (*cf* Definition 5.3.2).

(d) A transition edge that propagates $F \in B$ to the next state lines exists and the faulty state produced is an invalid state. The faulty/fault-free state pair in B possess a differentiating sequence but are unconditionally compatible under B's input don't-care sequence set.

(e) Same as (d), except that the faulty/fault-free state pair are conditionally compatible.

Validity of the Classification

Theorem 5.3.5 : *The subclassification of redundancies of Type 2 and 3 is complete.*

Proof. First, consider the classification of redundancies of Type 2. A fault $F \in A$ has to be propagated to the outputs of A if it is a redundancy of Type 2. The fault may produce a single faulty output that is not propagated through B or a sequence (length > 1) of faulty outputs. These two cases correspond to redundancies of Type 2(a)-(b) and 2(c)-(d). If one is dealing with a single corrupted vector, then the faulty vector produced may be an originally specified fanout edge for the states B can be in or the faulty vector may be unspecified. These two cases correspond to 2(a) and 2(b), respectively. In the case of output don't-care sequences that produce the same response from B, B may initially be moved into a faulty valid state or a faulty invalid state that is *not* compatible with the true state. This corresponds to redundancies of Type 2(c) and 2(d), respectively. If the faulty state had been compatible with the true state, it corresponds to a redundancy of Type 2(a) or 2(b).

Next, consider the classification of redundancies of Type 3. A fault $F \in B$ requires an initial vector for excitation/propagation to the next state lines or primary outputs of B. If this vector cannot be justified, one has a redundancy of Type 3(a). If this vector can be justified and the effect of the fault propagated to the primary outputs of B, F is testable, hence this case is discarded. The other case is when F is propagated to the next state lines and produces a faulty/fault-free state pair. If the states in this faulty/fault-free state pair do not possess a differentiating sequence, then there is a redundancy of Type 4. If the states in this faulty/fault-free state pair possess a differentiating sequence, but cannot be differentiated to the primary outputs of B since A cannot produce that sequence, there is a redundancy of Type 3(b)-(e). There are four possible cases of the faulty state being an invalid/valid state that is unconditionally/conditionally compatible to the valid state. □

Redundancies 2(a) and 2(b) are associated with single-vector don't-care outputs of A. If, for a given edge $e_a \in A$, the output o_2 produces the same response from B as output $o_1 = o(\epsilon_a)$, this means that the output $o(e)$ can in fact be specified as $o(e) = (o_1, o_2)$. Of course, one may have multiple occurrences of faulty output vectors producing the same responses for a fault F of Type 2(a) or 2(b).

Redundancies 2(c) and 2(d) are associated with don't-care output sequences of A (*cf* Eqn. 5.1 in Section 5.3.7). Don't-care output sequences arise when a set of outputs can be *simultaneously* replaced by another set of outputs without altering the functionality of

the machine.

Redundancy 3(a) is associated with the simple form of input don't-care described in Section 5.3.2, where transition edges in B need not be specified. Redundancies 3(b)-(e) are associated with don't-care input sequences to B. If, for instance, the edge $-0/1$ fanning out to state $s1$ in the fault-free machine in Figure 5.5 was corrupted to state $s2$, there is a redundancy of Type 3(b), since states $s1$ and $s2$ are unconditionally compatible (*cf* Section 5.3.2) under the don't-care input sequence (11, 11). In Figure 5.6, if the fanin edge $i1/o1$ to state $s1$ is corrupted to move to state $s2$ instead, there is a redundancy of Type 3(d), since $s1$ and $s2$ are conditionally compatible (*cf* Section 5.3.2) under the don't-care input sequence ($i1$, $i2$).

5.3.7 A Synthesis Procedure for Irredundant Cascaded Machines

Optimizing the Cascaded Machines

The procedure presented below represents a one-pass optimization for a cascade and eliminates a large number of redundancies in a cascade.

```
optimize-cascade( A, B )
{
    output-expansion-1 ( A, B ) ;
    irredundant-1( A ) ;
    exploit-input-dc ( B, DC^A ) ;
    irredundant-1( B ) ;
}
```

The synthesis procedure given above is now described in detail. A and B are arbitrary, possibly incompletely specified State Transition Graphs. Using the procedure **output-expansion-1**(), all the single-vector don't-care outputs for edges in A in $A \longrightarrow B$ are found. The procedure **irredundant-1**() uses the techniques described in Section 5.2 to make a single machine irredundant in isolation. The initial state minimization in **irredundant-1**() is augmented by the compatibility checking techniques of Section 5.3.4, given the expanded machine A. The iterative procedure **eliminate-equivalent/invalid-SRFs**() within **irredundant-1**() follows the state minimization process. The don't-care set DC^A, that corresponds to input vector sequences that never occur at the outputs of A, is generated. The procedure **exploit-input-dc**() state minimizes B under this don't-care set, exploiting all unconditional compatibilities between states.

Theorem 5.3.6 *The procedure* **optimize-cascade**() *produces a cascade* $A \longrightarrow B$ *that is irredundant for all Type 1, Type 2(a), Type 3(a), Type 3(b) and Type 4 faults.*

Proof. Type 1 and Type 4 faults cannot exist, since A and B are irredundant in isolation.

Consider a fault $F \in A$. After the procedure **exploit-input-dc**() followed by state minimization has been performed on B with a complete don't-care input sequence set, each remaining (specified) edge in the machine B, can be justified, by some input sequence to A. After B has been made prime and irredundant, one is guaranteed that at least one of the originally specified edges is a test vector in the combinational sense for any fault $F \in B$. That is, there is a vector that excites and propagates F to the primary outputs of B or the next state lines. This vector can be reached controlling A alone by Theorem 5.3.2. Therefore, F cannot be a redundancy of Type 3(a).

Next, consider redundancies of Type 3(b). After the procedure **exploit-input-dc**() has been used on B with a complete don't-care input sequence set, each pair of valid states remaining in B possesses a differentiating sequence that is *not* in the don't-care input sequence set. This is because during state minimization in **exploit-input-dc**(), any such states will have been detected (*cf* Theorem 5.3.2). Therefore, F cannot be a redundancy of Type 3(b). (However, valid state pairs in B may be conditionally compatible, and also invalid next states that are unconditionally/conditionally compatible to the true (valid) next state may be produced due to a fault F.)

Redundancies of Type 2(a) cannot exist because output expansion has been performed on A, using **output-expansion-1**() which finds all edge expansions that elicit the same response from machine B. (*cf* Theorem 5.3.4). A fault $F \in A$ can be initially propagated to the outputs of A or the next state lines. If all test vectors for F propagate F to the output lines of A alone and produce valid/specified faulty outputs (if even one vector produces an invalid output, F cannot be a redundancy of Type 2(a)), then because all the output don't-cares for each transition edge in A have been exploited, one is guaranteed that at least one of the vectors (edges) corrupted by F will elicit a different response for some state that B can be in. (By different response it is meant that B goes to a different incompatible state or produces a different output). On the other hand, if F is propagated to the next state lines of A alone, then a corrupted vector will exist such that it produces a faulty state that can be differentiated from the true state *under the output don't-care set.* The state minimization after **output-expansion-1**() and before **irredundant-1**() will ex-

ploit the output don't-care set to determine compatibilities between states. This means there is a differentiating input vector sequence (to A) such that the final faulty output necessarily elicits a different response from B or is an invalid/unspecified fanout edge for the states B can be in. If F is propagated to both the outputs and the next state lines of A, then for some test vector either the faulty output will directly elicit a different response from B or the faulty/fault-free state pair will possess a differentiating sequence that eventually elicits a different response from B. Thus, F is testable or is not a redundancy of Type 2(a). State minimizing the machine B in **irredundant-1**() after using the procedure **exploit-input-dc**() can change the output don't-cares for A to include unspecified outputs (*cf* Section 5.3.7), but one is only concerned with specified outputs when dealing with redundancies of Type 2(a). □

Irredundant Cascades

Eliminating all forms of redundancies in a cascade requires a two-pass optimization. Since one begins from a STG specification, one initially has no knowledge as to the compatibility between valid and invalid states in A or B. After one pass through the synthesis procedure, one can determine compatibilities between invalid and valid states in machine B. This information is used to find additional output don't-cares for A, as described in the sequel. In the second pass, one expands the outputs of A to include invalid/unspecified outputs, using the above information. This, in turn, may introduce additional don't-care input sequences to B. It should be noted that the procedures **irredundant-1**() and **irredundant-2**() are *themselves* iterative procedures where optimization is carried out to convergence (*cf* **eliminate-equivalent/invalid-SRFs**() in Section 5.2). In Section 5.5 a more efficient single global optimization loop to convergence is presented, which performs a single iteration in procedure **irredundant-2**().

```
irredundant-cascade( A, B ):
{
    for( iter = 1; iter ≤ 2; iter = iter + 1 ) {
        if ( iter = 1 ) output-expansion-1 ( A, B ) ;
        else output-expansion-2 ( A, B ) ;
        irredundant-1( A ) ;
        exploit-input-dc ( B, DC^A ) ;
        find-cond-compatibility ( B, DC^A ) ;
```

```
        irredundant-2( B, DC^A ) ;
    }
}
```

The steps in **irredundant-cascade()** are now described in detail. The procedure **output-expansion-2()** is an enhanced version of **output-expansion-1()**. There are two enhancements corresponding to the don't-cares for Type 2(b) and Type 2(c)-(d) redundant faults.

1. Given an optimized B, for each valid state, all the invalid states that are compatible to this state are found. There might be a situation where, for a particular transition edge in A, an output different from the edge's output places B in an invalid state that is compatible to the true valid state. This output represents a don't-care for the transition edge and is detected in **output-expansion-2()** (but not in **output-expansion-1()**). There is also the simpler situation of A producing a faulty output that was originally unspecified for the state(s) B is in, that moves B to the same or compatible next state and produces the same output from B. The output of the transition edge can be expanded to this unspecified combination.

2. Don't-care output *sequences* are detected for A. The detection of these sequences is performed by checking if valid/invalid states in B, that are not compatible to valid states and reached by faulty outputs from A, produce the same response in B due to the corruption of other transition edges in A. The corrupted outputs represent a don't-care output sequence for edges in A. A two-vector output don't-care sequence is shown in the cascade of Figure 5.8 [73]. Assuming that the starting states of the two machines are $a1$ and $b1$ respectively, the output of edges (1, $a3$) and (1, $a5$) cannot be expanded via procedure **output-expansion-1()**. However, the functionality of the cascade is maintained if $o(1, a3)$ is replaced with output b while *simultaneously* replacing $o(1, a5)$ with output a. The choice in selecting outputs is summarized below:

$$(o(1, a3), o(1, a5)) = (a, b) \bigvee (b, a) \qquad (5.1)$$

Current state minimizers and logic minimizers are restricted in their capability to exploit don't-cares. Don't-care output sequences of the form of Eqn. 5.1 cannot be optimally exploited, other than by exhaustive search. Note that these don't-cares cannot be represented even by Boolean relations [21], discussed in Section 5.3.5.

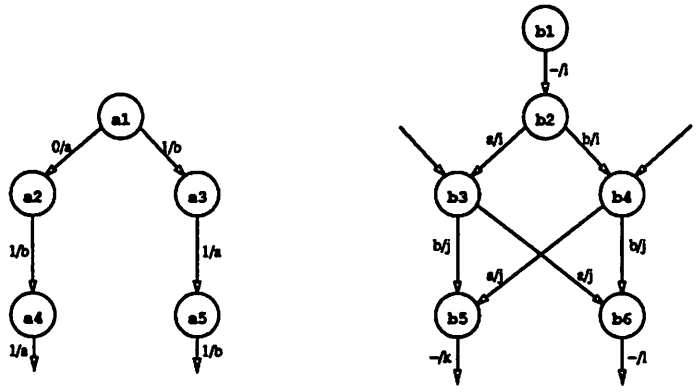


Figure 5.8: Output don't-care sequences

The procedure **irredundant-2()** is also an enhancement on procedure **irredundant-1()**.

Irredundant-2() uses **eliminate-equivalent/invalid-SRFs()** with an additional don't-care set at Step **A**. At Step **A**, one has,

$$\mathbf{A}: \quad FA^{DC} \mid e \in fanin(q) \;=\; (q,\, v_1,\, ..\, v_m,\, iv_1,\, ..\, iv_n,\, ni_1,\, ..\, ni_l)\;; \qquad (5.2)$$

where v_1, .. v_m and iv_1, .. iv_n are valid and invalid states respectively, that are compatible to q when B is viewed in isolation, *i.e.* deemed completely controllable. ni_1, .. ni_l are states *not* compatible to q when B is viewed in isolation, but conditionally or unconditionally compatible to q, given that the fanin edge $e \in fanin(q)$ is being traversed (and under the don't-care set DC^A). The detection of all possible unconditional and conditional compatibilities is achieved by procedures **exploit-input-dc()** and **find-cond-compatibility()** (*cf* Theorem 5.3.2 and Theorem 5.3.3). FA^{DC} in general is represented by Boolean relations as described in Section 5.3.5.

Theorem 5.3.7 *The procedure* **irredundant-cascade()** *results in an irredundant cascade.*

Proof. The procedure **irredundant-cascade()** is an enhanced version of the procedure **optimize-cascade()** and the arguments that Type 1, Type 2(a), Type 3(a), Type 3(b) and Type 4 faults are testable hold here as well. These redundancies are eliminated in the first pass of the procedure.

It is now shown that possible redundancies of Types 2(b), 2(c)-(d) and 3(c)-(e) are eliminated in the second pass.

The procedure **output-expansion-2**() uses the additional don't-care outputs for A corresponding to the invalid or valid states in B that are compatible to valid states and which are reached by faulty outputs other than the specified fanout edges of B[3]. Using these don't-cares ensures that Type 2(b) redundancies don't exist. The argument is similar to the argument of Theorem 5.3.1 for the Type 2(a) redundancy. A fault $F \in A$ will immediately or eventually produce an invalid/unspecified output such that the invalid output elicits a different response from B. If B is moved to a faulty invalid state one is guaranteed that the invalid state is not compatible to the true state. Thus, F is testable or is not a redundancy of Type 2(b).

Redundancies of Type 2(c)-(d) are associated with don't-care output sequences for A. That is, it does not matter if A asserts one particular sequence or another due to its constrained observability. If the don't-care sequences corresponding to Eqn. 5.1 are exploited in the output expansion procedure, one is guaranteed that the corrupted sequence does not elicit the same response as the true one from B.

Finally, redundancies of Type 3(c)-(e) are considered, namely, conditional compatibilities between valid/valid state pairs, unconditional and conditional compatibilities between invalid/valid state pairs in B. By Theorem 5.3.2, all unconditional compatibilities are detected in procedure **exploit-input-dc**(). By Theorem 5.3.3, all conditional compatibilities are detected by procedure **find-cond-compatibility**(). The don't-care set corresponding to Eqn. 5.2 in **eliminate-equivalent/invalid-SRFs**() will guarantee, after B has been made prime and irredundant, that any faulty/faulty-free state pair that is produced due to a fault F, regardless of whether the faulty state is valid or invalid, will possess a differentiating sequence not in DC^A. Also, this corrupted transition edge in B will not be such that the faulty state is conditionally compatible to the true state under DC^A (with the restriction that the transition edge has been traversed in B). This means that the pair can be differentiated from the inputs of A and F cannot be a redundancy of Type 3(c)-(e). □

[3]This information can be obtained after the first pass.

5.4 Generalization to Multiple Interacting Finite-State Machines

Multiple interacting finite-state machines are common in industrial chip designs. In Figure 5.4, an example sequential circuit composed of three interacting finite-state machines was shown. In Figure 5.4, there are not only local feedback loops for each machine, but also a global feedback loop via latches $L1$, $L2$ and $L3$. In Figure 5.9, a pair of mutually interacting machines that communicate via their present states alone was shown. In the next section, the issues involved in generalizing the optimal synthesis procedures of Section 5.3 to multiple interacting circuits are described. In Section 5.5, it is indicated how the procedures of Section 5.3 are applicable, in a uniform way, to multiple interacting circuits with arbitrary topologies that communicate via their present states.

5.4.1 Generalization of Observability and Controllability Don't-Cares

The don't-care sets associated with any arbitrary set of interacting FSMs are essentially the same as those in a cascade. At any given set of intermediate lines or flip-flops that are not observable/controllable, there are don't-care input and output sequences. An arbitrary set of interacting machines can be viewed as several occurrences of individual cascades and we conjecture that the don't-care sets required for synthesizing irredundant cascades can be used iteratively to eliminate all redundancies in the circuit.

In order to compute the input and output don't-care sequence set for an individual FSM embedded in an network, one has to propagate don't-care conditions from the primary inputs and primary outputs. In particular, to compute the don't-care input sequences for any machine, one has to go forward from the primary inputs to the machine. In order to compute the don't-care outputs for any machine, one has to go backward from the primary outputs to the machine.

The explicit propagation and computation of don't-care sequences of arbitrary lengths can be memory and CPU-intensive. The next focus of attention will be on the generalization of the invalidity and conditional equivalence detection algorithms developed previously to multiple interacting circuits.

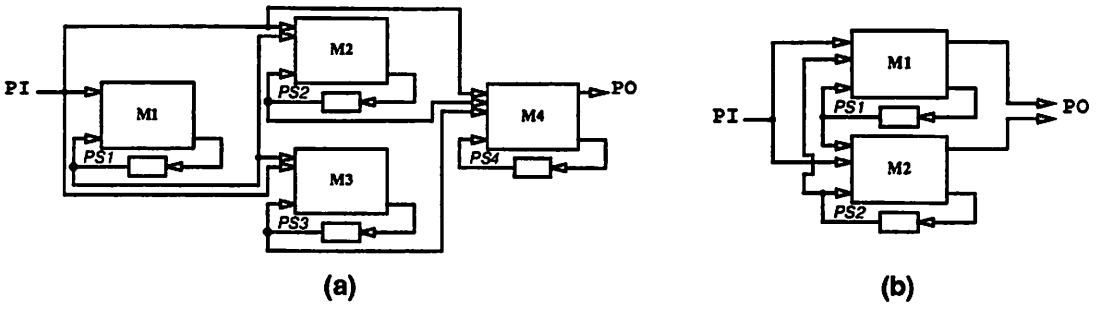


Figure 5.9: FSMs communicating via their present states

5.5 Invalidity and Conditional Compatibility Based Synthesis

While the classification of redundancies in interacting sequential circuits presented thus far has been complete, the optimization procedures that follow from the classification have tended to be computationally intensive. Even though redundancy identification in sequential circuits is by nature a difficult problem, a part of the reason for the computational complexity of the proposed procedures can be ascribed to the relatively general nature of the classification of redundancies.

In this section, we present an efficient synthesis algorithm where only a *single* global optimization loop to convergence is required for fully testable cascades *as well as* interconnections of multiple FSMs with global feedback, rather than the double loop required by the algorithms in Section 5.3.7. This improvement is brought about by the fact that the algorithm is specifically suited to a class of interacting sequential circuits commonly found in practice. Two examples of such topologies are shown in Figure 5.9. The sequential circuit in Figure 5.9(a) consists of four interacting FSMs, while the circuit in Figure 5.9(b) consists of two such FSMs. The salient feature of this class of circuits is that communication between the FSMs is solely via their present states. This is the only requirement imposed. Usually, an interacting sequential circuit that is to be made irredundant but does not satisfy this property can be made to satisfy it by an appropriate repartitioning of the combinational logic. Cascade and parallel interactions between submachines are special cases of the interaction shown in Figure 5.9.

The algorithm is illustrated by means of the example of Figure 5.9(b). Shown in Figure 5.10 are the fault-free STGs of the two interacting FSMs of Figure 5.9(b). Had

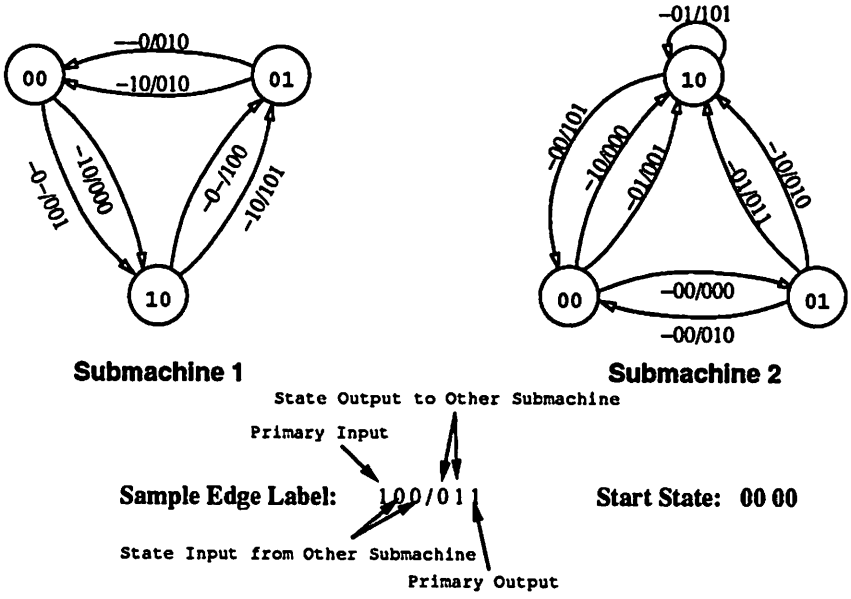


Figure 5.10: The state graphs of two fault-free interacting FSMs

the topology been a cascade, $M_1 \longrightarrow M_2$, the state input to M_1 from M_2 would be all don't-cares. The start state of the overall interacting circuit is 00 00, which corresponds to a start state of 00 for each individual FSM.

5.5.1 Exploiting Compatibility Between States

The definition of compatibility in this section is slightly different from the definitions in Section 5.3.2 in that two states are considered compatible if there is never a conflict between any pair of *primary output* sequences that can be generated starting from the two states. The outputs asserted by a FSM that are only utilized to communicate with other FSMs are not used to establish compatibility. As a result, pairs of states may be compatible under this definition for an FSM embedded in a network even if they are not compatible for the FSM considered in isolation. As in Section 5.3.2, compatibility may either be conditional or unconditional.

A fault causing only an interchange of unconditionally compatible states is redundant. An example of a pair of unconditionally equivalent states (*e.g.* states 00 and 01 of Submachine 2 in Figure 5.10) becoming a faulty/fault-free pair is shown in Figure 5.11. *EDGE 1*, shown in dotted lines, has been corrupted to *EDGE 2* by the fault. In Figure 5.12,

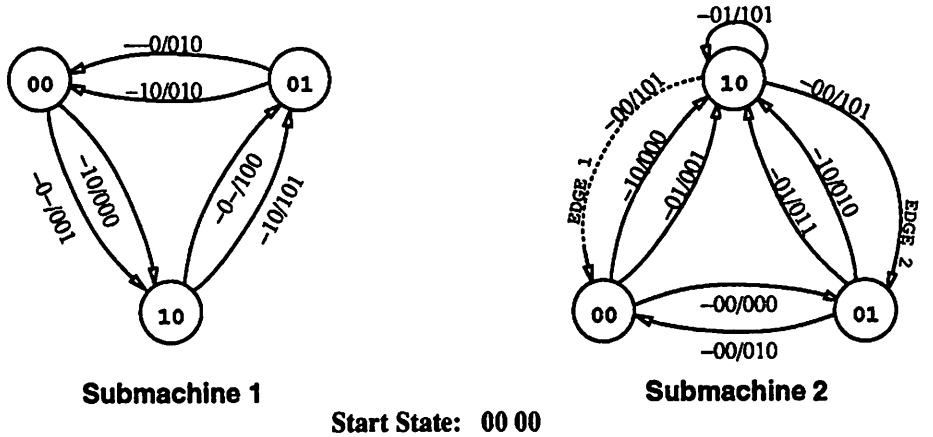


Figure 5.11: A fault causing interchange of unconditionally compatible states

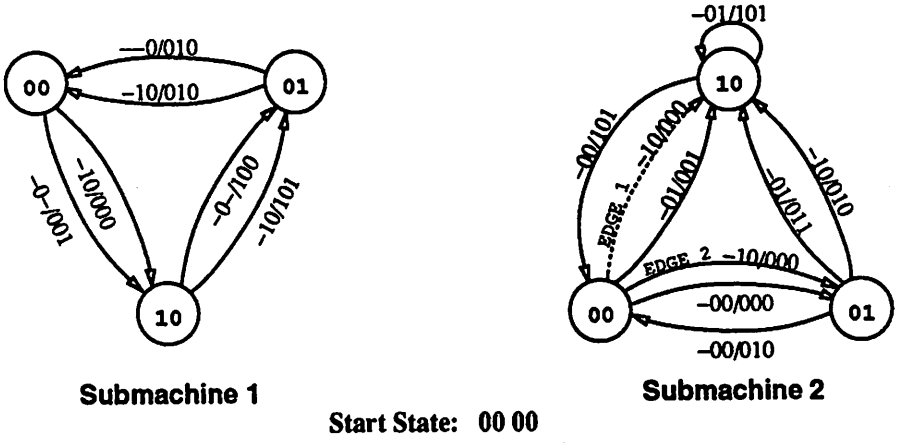


Figure 5.12: A fault causing interchange of conditionally compatible States

states 10 and 01 of Submachine 2 are conditionally compatible under the fanin edge *EDGE 1*. Therefore, when *EDGE 1* is corrupted to *EDGE 2*, one again has a redundant fault. Redundant faults due to state compatibilities are removed, as in Section 5.3.7, by optimizing the logic under a fanin don't-care set.

What gains can be accrued as a result of the fact that the FSMs communicate solely via their present states? The following lemma answers this in part:

Lemma 5.5.1 *In an FSM network, if the only means of communication between the FSMs is all their state variables and that there are no don't cares on the primary inputs, the longest don't-care input sequence that does not contain any other don't-care sequences, for*

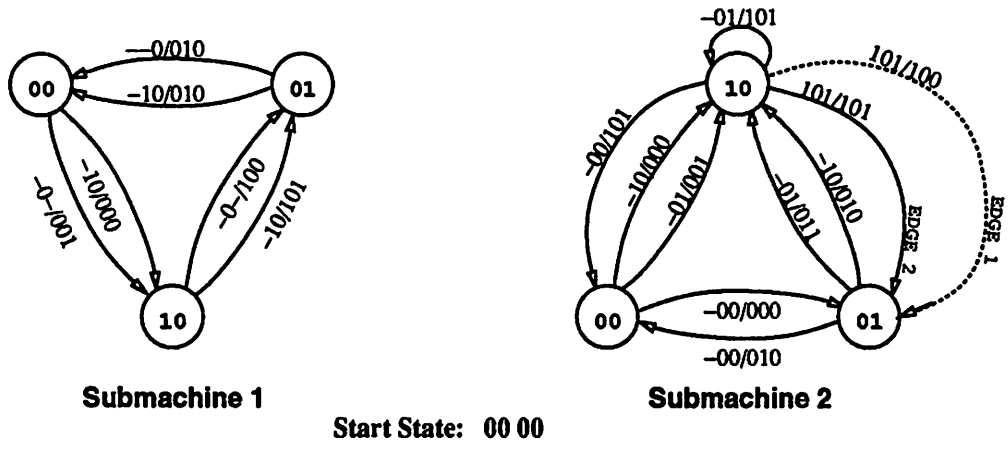


Figure 5.13: A fault causing corruption of only an unspecified edge

any machine, is of length two.

Proof. Since the intermediate lines between FSMs are all the present state lines, a don't-care sequence corresponds to sequences of states that don't occur. Invalid states correspond to don't-care sequences of length one. Thus, don't-care sequences of length two, that do not contain don't-care sequences of length one, can only consist of valid states. Say, that s_1, s_2 and s_3 are states in a FSM, and that length-two sequences $s_1 \rightarrow s_2$ and $s_2 \rightarrow s_3$ are care sequences. Then the sequence $s_1 \rightarrow s_2 \rightarrow s_3$ has to be a care sequence. Hence, there are no don't-care sequences of length three or more, that do not contain smaller don't-care sequences. □

The tangible benefit of this property is that the algorithm that identifies conditional compatibility between states (*cf* Section 5.5.4) only has to search for compatibility for every fanin edge traversed rather than for every fanin sequence traversed, as would be the case otherwise. Since the don't-care sequences considered can at most be of length two, if two states are conditionally compatible under a fanin edge, e, it implies that there is a set of length-two don't-care sequences such that e is the first vector in these don't-care sequences, and that *any* edge that is the first vector of a sequence differentiating the two states is the second vector of one of these length-two don't-care sequences.

5.5.2 Exploiting Invalidity of States and Edges

Given a starting state for the overall machine, certain states in an embedded FSM may be unreachable and certain edges may never be traversed as a result of the limited controllability of the inputs from the other FSMs to that FSM. A fault that only affects such states or edges is sequentially redundant. An example is shown in Figure 5.13 of an improperly exploited unspecified edge *(EDGE 1)* being the only edge that gets corrupted (to *EDGE 2*). *EDGE 1* is unspecified because Submachine 2 is never in state 10 when Submachine 1 is state 10, given that the starting states of Submachines 1 and 2 are 00 and 00. respectively. Such redundancies are removed by optimizing the logic with the unreachable states/untraversed edges used as input don't-cares.

It is interesting, and useful from the point of proving the completeness (*cf* Section 5.5.6) of the algorithm being described, to note the effect of exploiting the unreachability don't-cares for each individual embedded FSM on the overall sequential circuit. The following lemma is useful in this regard:

Lemma 5.5.2 *When each individual embedded FSM is prime and irredundant under its set of input don't-cares corresponding to states belonging to it that are never reached and edges belonging to it that are never traversed, and the communication between the FSMs is via their present states, there always exists at least one justifiable state in the overall machine that can be used to propagate the effect of a fault.*

Proof. Consider an example of machines that communicate via their outputs, and not just via the states. An unreachable state/unspecified edge in such a machine corresponds, in general. to unreachable states and untraversed *sequences of edges* in embedded machines. On the other hand, if the embedded machines communicate only via their present states, an unreachable state/unspecified edge in the overall machine corresponds only to unreachable states and untraversed *single edges* in the embedded machines. When an embedded machine is optimized under its set of input don't-cares corresponding to states belonging to it that are never reached and edges belonging to it that are never traversed, any fault must affect at least one edge in the embedded machine that can be traversed. Since an edge that can be traversed in the embedded machine corresponds directly to a state in the overall machine that is reachable, there exists at least one justifiable state in the overall machine that can be used to propagate the effect of the fault. □

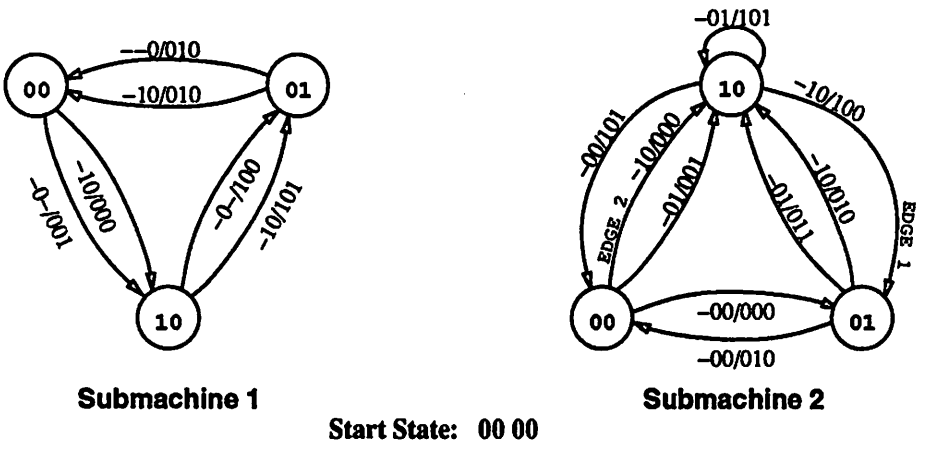


Figure 5.14: State graph of a decomposed machine in which isomorphism can occur

5.5.3 Preventing Isomorphism

A fault in the interacting sequential circuit that causes the STG of the overall faulty sequential circuit to be isomorphic to the STG of the overall fault-free sequential circuit is redundant (*cf* Section 5.2). It is shown in this section how such redundancies can be avoided. Again, the fact that the communication between FSMs is via their present states leads to useful properties. Classical isomorphism occurs when the codes for states are interchanged without changing the outputs or next states on any of the edges fanning out of the states. Isomorphism of this type can never occur in the classes of interactions considered here because a portion of the outputs asserted by all the edges fanning out of a state in an FSM is actually the code for the state itself. But isomorphism can still occur in the *overall machine* if a group of states, Θ, in the FSM within which an exchange of codes occurs satisfies a certain property. The property to be satisfied is that the response of the other FSMs in the clock cycles following the cycle in which an isomorphed edge has been traversed, does not depend upon the specific state in Θ that the FSM is in. The states 00 and 01 in Submachine 2 in Figure 5.14 are not equivalent, but it is possible for an exchange of the codes of these two states to cause isomorphism in the overall machine because the response of Submachine 1 is independent of whether Submachine 2 is in state 00 or 01.

It is also possible that an exchange of codes (that does not change the functionality of the sequential circuit as a whole) between a set of states occurs only for a subset of the fanin to these states, leading to a redundancy. Consider Figure 5.14 again. States 01 and

10 in Submachine 2 are not equivalent, but the response of Submachine 1 is independent of whether Submachine 2 is in state 01 when the fanin edge *EDGE 1* was traversed or whether it is in state 10 when the fanin edge *EDGE 2* was traversed. Thus, isomorphism can occur in the overall machine by an exchange of codes for the states 01 and 10 in Submachine 2 under the fanin *EDGE 1* and *EDGE 2*. It should be noted that only the exchange of codes between incompatible states is of significance in the case of isomorphism.

The following lemma is useful in preventing isomorphism in interacting sequential circuits:

Lemma 5.5.3 *If a fault can cause isomorphism in the STG of the overall machine, it also causes isomorphism to occur in the STG of the embedded FSM in which it occurs.*

Proof. The code for a state of the overall machine can be thought of as effectively consisting of subfields, each subfield corresponding to the code of a state in the FSM associated with the subfield. An exchange of codes between states of the overall machine due to a single fault in one machine must involve an exchange of a single subfield of the codes. Thus, it is necessary for isomorphism to occur in a single embedded FSM for it to occur in the overall machine. □

The don't-care associated with the above lemma is the choice of codes for states in embedded machines. By virtue of this lemma, one only needs to prevent a stuck-at fault from resulting in an isomorphism-causing (w.r.t. the overall machine) exchange of codes in the machine in which the fault occurs. Such an exchange in each individual FSM is easily prevented by using the techniques of Section 5.2.1.

5.5.4 Searching for Unreachability and Compatibility

In this section, the flavor of algorithms used to identify unreachable states/untraversed edges and compatible states in mutually interacting machines is presented. The actual implementation of these procedures is more efficient.

The procedures described below use the fact that given an edge in Submachine 1, one can easily find the corresponding edge in Submachine 2, and vice versa. For instance, an edge in Submachine 1 may be $(i \ @(q_{m_2}, q_{m_3}, .. q_{m_N}) \ @ \ q_{m_1})$[4], where i is the input

[4]The @ operator, as in $(i \ @ \ q)$, represents a concatenation of the strings i and q.

vector and q_{m_k} is the state of Submachine m_k. The corresponding edge in Submachine 2, for example, is then $(i\ @(q_{m_1},\ q_{m_3},\ ..\ q_{m_N})\ @\ q_{m_2})$.

traverse-edges() is a recursive procedure that traverses all the edges in the overall FSM given the starting state for each FSM. Every edge in each submachine encountered during the enumeration is marked. The set of edges that remain unmarked when **traverse-edges()** is called is the set of edges in the overall FSM that are never traversed for the specified starting state. Calling **traverse-edges()** exactly corresponds to traversing the overall machine in a depth-first manner starting from the states supplied as argument. Therefore, the efficiency of the procedure is the same as that of depth-first search. Initially, **traverse-edges()** is called *once*, with the actual starting state of the overall machine as argument, in order to find all the unreachable states and unreachable edges in each submachine. In the pseudo-code below, N is the total number of embedded FSMs. As in previous sections, n(e) denotes the next state that the edge e fans out to.

traverse-edges(q_{m_1}, .. q_{m_j}, .. q_{m_N}):
{
    foreach(m_x = m_1 to m_N) {
        foreach(unmarked fanout edge of q_{m_x}, e_{m_x} = $(i\ @(q_{m_1},\ q_{m_2},\ ..\ q_{m_{x-1}},\ q_{m_{x+1}},\ ..\ q_{m_N})\ @\ q_{m_x})$ {
            Mark e_{m_x} ;
            q'_{m_x} = n($i\ @(q_{m_1},\ q_{m_2},\ ..\ q_{m_{x-1}},\ q_{m_{x+1}},\ ..\ q_{m_N})\ @\ q_{m_x}$) ;
        }
    }
    traverse-edges(q'_{m_1}, .. q'_{m_j}, .. q'_{m_N}) ;
}

Lemma 5.5.4 *The procedure* **traverse-edges()** *is complete in that if it is called for each FSM, all unreachable states/untraversed edges are identified.*

Proof. That **traverse-edges()** traverses all the traversable edges in the specified FSM is obvious from analyzing the procedure. Therefore, if **traverse-edges()** is called once initially, all the untraversable edges are identified. An unreachable state has fanout edges that are all untraversable. Since all untraversable edges are identified, all unreachable states are identified automatically. □

$(i\ @(q_{m_2},\ q_{m_3},\ ..\ q_{m_N})\ @\ q_{m_1})$ here is a concatenation of the strings i, q_{m_2}, q_{m_3} up to q_{m_N} and the string q_{m_1}.

The procedure **traverse-edges-given-edge()** shown below, finds all the traversable edges, *given the traversal of some edge,* e_{m_j}. **traverse-edges-given-edge()** is used within a procedure that finds all conditionally compatible states.

traverse-edges-given-edge($e_{m_j} = (i\ @(q_{m_1}, q_{m_2}, .. q_{m_{j-1}}, q_{m_{j+1}}, .. q_{m_N})\ @\ q_{m_j})$):
{
    Unmark all edges ;
    Mark e_{m_j} ;
    foreach($m_x = m_1$ to m_N) {
        q'_{m_x} = n($i\ @(q_{m_1}, q_{m_2}, .. q_{m_{x-1}}, q_{m_{x+1}}, .. q_{m_N})\ @\ q_{m_x}$) ;
    }
    traverse-edges($q'_{m_1}, .. q'_{m_j}, .. q'_{m_N}$) ;
}

Conditional compatibility is found using the procedure **find-all-compatible-states()** given below. G_{m_j} denotes the STG of Submachine m_j. Say that one is required to find all the compatibilities in Submachine m_j. The basic idea is to traverse a certain fanin edge, say e_{m_j} and identify all the edges in Submachine m_j that cannot be traversed as a result. These untraversable edges are removed from G_{m_j} to obtain G'_{m_j}. By constructing G'_{m_j}, one can use standard state minimization algorithms [89] to find conditional compatibilities. Note that one merely needs to find all states compatible to the fanin state of edge e_{m_j} to find the fanin don't-care set (*cf* Eq. 5.2) for edge e_{m_j}. Unlike in state minimization, one does not require information as to the other compatibilities, nor does one have to select a maximal compatible set.

find-all-compatible-states($G_{m_j}, G_{m_1}, G_{m_2}, .. G_{m_{j-1}}, G_{m_{j+1}}, .. G_{m_N}$):
{
    foreach(valid edge $e_{m_j} = (i\ @(q_{m_1}, q_{m_2}, .. q_{m_{j-1}}, q_{m_{j+1}}, .. q_{m_N})\ @\ q_{m_j}) \in$
        G_{m_j}) {
        traverse-edges-given-edge(e_{m_j}) ;
        $G'_{m_j} \subset G_{m_j}$ with only marked edges ;
        Find all states in G'_{m_j} compatible to n(e_{m_j}) ;
    }
}

Lemma 5.5.5 *The procedure* **find-all-compatible-states()** *is complete in that it finds all the conditionally and unconditionally compatible states for the embedded FSM specified as argument.*

Proof. According to Lemma 5.5.1 that it is sufficient to search for compatibility between states only under the immediate fanin to the states. Also, it is apparent from the outline of the procedure **find-all-compatible-states**() given above that possibility of compatibility under all possible immediate fanin is searched for. States that are unconditionally compatible can be thought of as being conditionally compatible under *all* their immediate fanin. Therefore, searching for all conditional compatibilities identifies all conditional as well as unconditional compatibilities. □

5.5.5 The Optimization Procedure

The following optimization procedure ensures an irredundant set of interacting circuits by applying the results of Sections 5.5.1 to 5.5.4.

1. **State Encoding:** The encoding for the individual FSMs is made locally optimal for an arbitrary logic implementation, if possible. This involves exploring a certain number of encodings. If a locally optimal encoding is not possible, the individual FSMs are implemented in a two-level or algebraically factored multilevel network.

2. **Logic Optimization:** The logic for each FSM is optimized individually under the invalid state and fanin don't-care set. This step is carried out repeatedly for every FSM until convergence. Convergence is guaranteed because at every iteration, the logic of any FSM can only decrease. The end result is a fully irredundant interconnection of FSMs.

The pseudo-code for the repeated minimization algorithm is shown below. The algorithm takes as argument the logic-level implementation, S_i, of each submachine.

```
irredundant-interact( S_1, .. S_N ):
{
    iter = 1 ;
    do {
        foreach ( i = 1; i ≤ N; i = i + 1 ) {
            if ( iter = 1 ) G_i = extract-stg( S_i ) ;
            else G_i = extract-stg( S_i'' ) ;
        }
        foreach ( i = 1; i ≤ N; i = i + 1 ) {
            foreach ( valid state q ∈ G_i ) {
```

```
                Find all valid states (v_1, .. v_x) conditionally compatible to q ;
                Find all invalid states (iv_1, .. iv_y) conditionally compatible to q ;
                A: FA^DC | e ∈ fanin(q) = (q, v_1, .. v_x, iv_1, .. iv_y) ;
            }
            S_i' = optimize( S_i, FA^DC ) ;
            IV = extract-unspecified-edges( S_i' ) ;
            S_i'' = optimize( S_i', IV^DC ) ;
        }
        iter = iter + 1 ;
    } while( S_1 ≠ S_1' || S_2 ≠ S_2' || .. S_N ≠ S_N' ) ;
}
```

The procedures **extract-stg**() and **optimize**() are the same as those of Section 5.2.3. The procedure **extract-unspecified-edges**() finds all the unreachable states/untraversed edges in an embedded by calling **traverse-edges**().

5.5.6 Completeness of the Algorithm

The completeness of the algorithm can be proved in two ways. One way is to map the effects of the algorithm onto the redundancy classification of Section 5.3 and show completeness. An alternate method involves treating the whole interacting sequential circuit as a *single* FSM and using the single FSM theory (*cf* Section 5.2) for the purposes of the proof. This is the method employed in the proof below.

Theorem 5.5.1 *The algorithm* **irredundant-interact**() *renders an interacting sequential circuit in which the communication between the individual FSMs is solely via their present states fully irredundant.*

Proof. The complete classification of sequential redundancies in single isolated FSMs was reviewed in Section 5.2. For the purposes of this proof, if the whole interacting sequential circuit s treated as a single isolated FSM, then proving the completeness of the algorithm reduces to showing that applying the algorithm results in the prevention of any of the redundancies described in Section 5.2 from occurring.

It was shown in Lemma 5.5.3 that isomorphism redundancies in the overall FSM could only occur if isomorphism occurred in a single embedded FSM. It was shown in Section 5.5.3 how isomorphism could be prevented in each individual FSM. Therefore, no isomorphism-SRFs can occur in the overall FSM.

By virtue of Lemma 5.5.2, an invalid-SRF cannot occur in the overall FSM if the don't-cares due to unreachable states/untraversed edges in each individual FSM have been exploited during logic optimization. By Lemma 5.5.4, the procedure for searching for unreachable states/untraversed edges presented in Section 5.5.4 is complete. Therefore, no invalid-SRF can occur in the overall FSM.

Because a single fault can corrupt an edge in only one embedded FSM, not *all* equivalent states in the overall FSM can cause an equivalent-SRF to occur in the overall machine. Only those pairs of equivalent states in the overall machine that are formed from the same states in all but one of the embedded FSMs can cause an equivalent-SRF in the overall FSM. Effectively, an equivalent-SRF can occur in the overall FSM only due to compatible states in the individual embedded FSMs. By Lemma 5.5.1 and Lemma 5.5.5 the procedure used in Section 5.5.4 identifies all possible unconditionally/conditionally compatible states in each individual FSM. When these are exploited as described in Section 5.5.1, no redundancy can exist due to compatible states in single embedded FSMs, and therefore no equivalent-SRF can exist in the overall FSM.

Since the algorithm ensures that no sequential redundancy can exist in the overall interacting sequential circuit considered as an single FSM, it produces an irredundant, interacting sequential circuit. □

5.6 Results

Results based on the optimization procedures of Section 5.5 are presented. There are two basic advantages in operating from a hierarchical or distributed representation of sequential circuits:

1. The representation of the total behavior in terms of multiple interacting STGs is much more compact than the STG of the overall lumped machine. This is primarily because the state inputs to a Submachine **A**, from another Submachine **B**, need not be explicitly enumerated as minterms, since these lines are effectively primary inputs to **A**. Similarly, if the present state lines from **A** feed **B**, these are allowed to be don't-cares in the cubes in the STG of **B**. It has been shown in previous sections that a sequential circuit can be made irredundant by carrying out a series of *local* operations on the

Example	# L	PI	PO	Literal Count		CPU Time[1]		Mem.[1] (Mb)	
				Initial	Final	Per Pass	#Passes	Distr.	Flat
bbara	5	4	2	132	123	7 sec.	3	2.4	0.5
scf	8	27	54	1859	1859	31 min.	1	8.5	2.5
arbseq	9	6	8	650	650	22 min.	1	24.9	24.9
cascade.1	10	3	19	909	610	23 min.	3	8.5	82.0
tlc.12	10	3	5	185	134	82 min.	3	8.6	>150
tlc.34	11	11	6	780	173	117 min.	4	33.1	*
cascade.2	16	7	10	1584	1516	237 min.	2	74.1	*
cascade.3	16	27	54	2867	2302	717 min.	6	23.2	*
multiple.1	21	9	11	965	307	199 min.	4	41.8	*
multiple.2	32	27	64	4451	3818	954 min.	6	97.5	*
cascade.p	49	7	10	1584	1516	237 min.	2	74.1	*
multiple.p	64	27	64	4451	3818	954 min.	6	97.5	*

The final Fault-Coverage for all machines was close to 100%.
[1]On a DECstation 3100 running Ultrix.
* : STG of the product machine could not be obtained due to memory constraints.

Table 5.1: Full testability via optimal synthesis of interacting sequential machines

interacting FSMs. Based on this observation, it has been possible to propose efficient procedures for redundancy removal that can operate from a distributed representation. Thus, even if it were possible to flatten the representation of the interacting sequential circuit, there is no inherent necessity for doing so.

2. The number of states in each submachine is smaller than in the composite machine. Fewer compatibility checks have to be carried out and the basic state minimization procedure is faster when operating on each submachine separately.

The results of the optimization procedure operating on distributed representations of circuits are presented in Table 5.1. The examples **bbara** and **scf** are FSMs from the MCNC Logic Synthesis Workshop benchmark set [76]. The examples **arbseq**, **tlc.12** and **tlc.34** are industrial examples, of which **tlc.12** and **tlc.34** are cascades. Examples **cascade.*** and **multiple.*** were obtained by interconnecting the MCNC benchmark set FSMs in various ways.

In the final column of the table, the memory requirements of the proposed procedure are compared against a procedure operating on flattened representations of the same

circuits. The circuits following **tlc.12** in Table 5.1 (**tlc.34, cascade.2** ...) are large enough that the STG of the overall flattened machine cannot be extracted. The circuit **tlc.12** is of moderate size. Even though the STG of the lumped machine can be extracted for it, it is too large for state minimization to be viable. Thus, a full redundancy removal operating on the flattened representation is not possible. When it is only required to enhance the testability of the machines close to 100%, it is possible to handle machines with much larger numbers of latches. It should be noted that even this is not possible via the conventional approach of extracting the STG of the overall machine.

5.7 Conclusions

In this chapter, relationships between redundant logic and don't-care conditions in interacting sequential circuits were explored. Redundancies in non-scan sequential circuits may be testable from a combinational viewpoint, but may produce a faulty State Transition Graph (STG) that is equivalent to the STG of the true machine.

A classification of redundant faults in sequential circuits composed of single or interacting finite-state machines was presented. Don't-care sets can be defined for each class of redundancy and optimally exploiting these don't-care conditions results in the implicit elimination of any such redundancies in a given circuit. It has been shown that interacting sequential circuits can be made fully testable by carrying out a series of local operations on a distributed-style specification of circuit behavior. While a general optimization procedure can be based on this classification, algorithms can be tailored for particular forms of interaction between FSMs to improve efficiency.

Redundancy identification using sequential test generation algorithms can be viewed as a competing approach to the don't-care exploitation. In the test generation approach, in the worst case, the product of the fault-free and faulty machines have to be traversed repeatedly for each redundant fault. Such repeated traversal corresponds to computing the same information multiple times. When the machine has a large number of redundancies, this approach can prove to be inefficient. On the other hand, don't-care exploitation requires a few traversals of the machine being synthesized for every pass of the synthesis loop. A large number of redundancies can be removed during each pass. The disadvantages of the don't-care exploitation approach are that the logic optimization becomes complicated, and that it is practically difficult to ensure 100% testability because of the difficulty in extract-

ing and exploting the complicated don't-cares. Therefore, a viable approach to sequential synthesis for test is to use the don't-care exploitation approach for initial synthesis, and subsequently use test generation algorithms for removing the final few redundancies.

Chapter 6

Synthesis for Multiple-Fault Testability

6.1 Introduction

While the single-fault model was adequate in detecting malfunctioning chips in the past, the increasing density of components on integrated circuits has made the testing of chips for the presence of multiple faults imperative [81].

Multiple faults are difficult to test for because of the sheer number of multiple-fault conditions that are possible even for moderately sized circuits. Given a circuit with n wires, the total number of possible stuck-at multiple faults is $3^n - 1$. This number is much larger than the $2n$ single stuck-at-fault conditions that are possible on the same circuit. A large body of work has existed for a number of years for the test generation and testability analysis of combinational circuits for multiple faults [67, 22, 77, 98]. But, the large number of possible multiple faults has prevented the application of these methods to all but extremely small circuits. In addition, the results in [67] are applicable only to two-level circuits. Recent work has resulted in theoretical results that can predict the effects of certain operations commonly used by multilevel logic optimization algorithms on the multiple-fault testability of combinational circuits [56]. As a result, the effects of multiple faults on combinational circuits are now understood well enough that synthesis techniques have been proposed recently that can realize fully multiple-fault testable combinational circuits without excessive area penalties [56, 33, 34].

The analysis of sequential circuits, which are made up of latches in addition to combinational circuit elements, in the presence of multiple-fault conditions is a more difficult task. Very little work has been done in this area in the past. Nonscan testing of sequential circuits for the presence of multiple faults is complicated by the limited controllability and observability of the combinational circuit implementing the next-state and primary output logic. Multiple faults can exist that alter the functionality of the combinational portion of the circuit without changing the sequential behavior. In this work, the effects of multiple stuck-at faults on sequential circuits are analyzed. It is shown that the effects of multiple stuck-at faults on the state graph of a sequential circuit can be much more dramatic than the effects of single stuck-at faults. Based on the study of these effects, methods are proposed for the synthesis of sequential circuits for full or high multiple-fault testability.

Synthesis-for-testability techniques for single-fault testability of sequential circuits are reviewed briefly in Section 6.2. A review of previous work in the multiple-fault analysis of combinational circuits is presented in Section 6.3. The effects that multiple stuck-at faults can have on the state graph of a sequential circuit are presented in Section 6.4. In Sections 6.5 and 6.6, procedures for the synthesis of fully and highly multiple-fault testable sequential circuits are presented.

6.2 Synthesis Procedures for Nonscan Single-Fault Testability

A number of synthesis procedures have been proposed recently for obtaining fully single-fault testable non-scan sequential circuits [38, 39, 31]. Obtaining a fully single-fault testable circuit requires the removal of all redundancies. A general model of a sequential circuit S, implementing a single FSM is shown in Figure 6.1(a). Gates in the combinational network may be in the cone of the output logic, the next-state logic, or both. The State Transition Graph corresponding to one such machine is shown in Figure 6.1(b).

Redundant faults in S may be combinationally redundant (CRFs) or sequentially redundant (SRFs). Combinationally redundant CRFs can be eliminated via combinational logic optimization alone [14, 10, 56]. Sequentially redundant faults can be classified into three categories [39].

1. **equivalent-state faults:** The fault causes the interchange/creation of equivalent

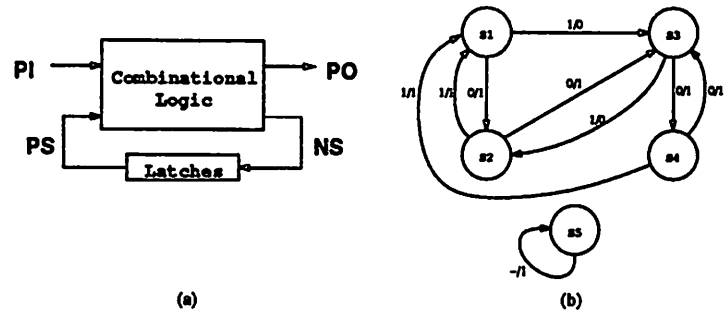


Figure 6.1: A sequential circuit

states in the STG.

2. **invalid-state faults:** The fault does not corrupt any fanout edge of a valid state in the STG, but does corrupt the fanout edge of an invalid state.

3. **isomorphic faults:** The fault results in a faulty machine that is isomorphic (with a different encoding) to the original machine.

In [39], it was shown that any sequential redundancy must fall into one of these classifications.

The removal of SRFs in the presence of single stuck-at faults can be done by the use of two broad classes of synthesis procedures:

- Optimal synthesis procedures.
- Constrained synthesis procedures.

Optimal synthesis procedures [39] define a don't-care set corresponding to the redundancies and optimize the combinational logic under that don't-care set. The attraction of such a method is that full single-fault testability is achieved with no area penalty. Constrained synthesis procedures [38] impose restrictions on the structure of the STG and the logic implementing it so that redundancies do not occur. Typically, some area penalty is involved with such methods. It is also possible to devise synthesis procedures that are neither purely optimal synthesis procedures nor constrained procedures but possess the beneficial properties of both [31].

Outlined below are brief descriptions of how the synthesis-for-testability procedures remove sequential redundancies. Invalid state SRFs are removed by the use of invalid

states as don't-cares during logic optimization. They can also be removed by ensuring that all the 2^n (n being the encoding length) states in the sequential circuit are reachable. SRFs due to isomorphism are removed either by a locally optimal state assignment or by ensuring that the combinational logic is either implemented in two-levels or is algebraically optimized starting from a two-level description. Equivalent-state SRFs are more complicated and a number of methods have been proposed for removing them. One technique involves partitioning the next-state logic so that the faulty and true next-states only differ in a single bit [38]. State assignment can then be used to ensure that the codes for equivalent states differ in at least two bits. Another method of avoiding equivalent-state SRFs is to optimize the logic under an equivalent-state external don't-care set (or Boolean relations [16] in general) [39]. One problem with these methods is that a single fault could cause a faulty state that was not equivalent to the true state in the fault-free machine to become equivalent to the true state in the faulty machine. A way of getting around this problem is to realize the next-state and output logic via different blocks. Another technique, proposed in [31], is to ensure that that the logic implementing the FSM satisfies the property that if a fault corrupts an edge to a different next-state, at least one sequence that distinguishes between the true and faulty next-states remains uncorrupted.

6.3 Multiple-Faults in Combinational Circuits

It was shown in [67] that a 100% single-fault testable two-level single output combinational circuit is also 100% multiple-fault testable. This result was extended in [56] to multilevel single output circuits obtained via algebraic optimization techniques [20]. In particular it was shown that for every multiple fault in the algebraically optimized circuit, there is an equivalent multiple fault in the multiple-fault irredundant two-level circuit. A corollary of this property of algebraically optimized circuits is that a complete multiple-fault test set for a single output two-level circuit is also a complete multiple-fault test set for the multilevel circuit obtained from it by algebraic optimization.

The basis for the results on the multiple-fault testability of sequential circuits in this chapter is the following lemma which is a corollary of Theorem 2 in [67].

Lemma 6.3.1 *For any multiple stuck-at fault in a two-level single output circuit, there exists a single stuck-at fault such that the test set for the single fault is a subset of the test set for the multiple fault.*

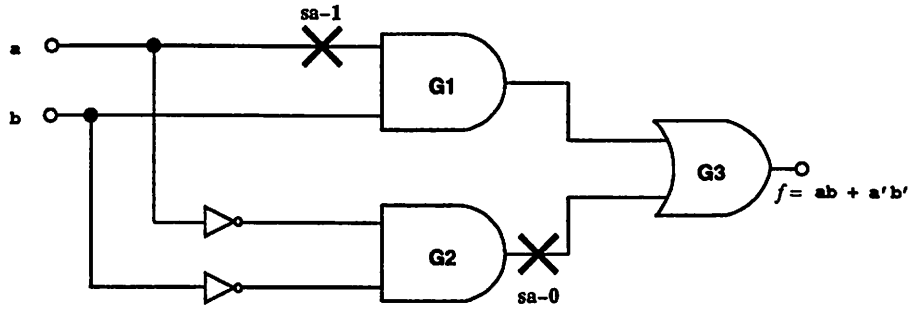


Figure 6.2: A multiple fault in a two-level circuit

To ensure that a multiple output two-level or algebraically optimized circuit is multiple-fault testable it is sufficient to ensure that the cone of every output is fully single-fault testable on its own. Essentially, this requires the separate optimization of the logic driving each output.

6.4 Multiple-Faults in Sequential Circuits

The effects that a multiple fault can cause in the STG of a FSM are similar to the effects caused by a single fault. Again, the faulty STG may be isomorphic to the fault-free STG, or the only edges corrupted may fan out of invalid states, or a true next-state may be corrupted to an equivalent faulty next-state. While a fully multiple-fault testable combinational portion of a fully single-fault testable sequential circuit is sufficient to ensure high sequential testability, two basic problems are encountered in handling multiple faults.

While techniques similar to those used in the single-fault case can be used to remove invalid state and equivalent state SRFs in the sequential circuit in the multiple-fault case also, the techniques for removing isomorphism redundancies in the case of single faults cannot be used in the multiple-fault case. It can be shown that if the combinational logic in a sequential circuit is two-level or algebraically optimized multilevel, isomorphism cannot occur due to a single fault [39] because all test vectors for any single fault in such a circuit produce either a D or a $\overline{D}$ at some output, but never both. For isomorphism to occur, the codes of state have to be exchanged and consequently there must exist a pair of test vectors such that one of them produces a D and the other a $\overline{D}$ at some next-state line. It is easily shown that this property is not true for two-level or algebraically optimized circuits in the presence of multiple faults. Consider the example of the multiple fault in

the two-level circuit shown in Figure 6.2. When the minterm $a'b$ is applied as a test vector, a $\overline{D}$ is produced at the output, whereas when the minterm $a'b'$ is applied as a test vector a D is produced at the output. A similar example can be illustrated for the case of an algebraically optimized circuit also. Therefore, implementing the combinational logic as two-level or algebraically optimized multilevel logic is no longer sufficient to prove that isomorphism does not occur in the presence of multiple faults. [1] On the other hand, it may be argued that while the removal of isomorph SRFs in the presence of multiple faults is important from the point of view of ensuring full testability, the multiple faults that cause isomorphism in actual state graphs with a large number of outputs and state bits are rare and do not reduce the multiple-fault coverage a great deal.

A second, and probably more important factor in making the problem of multiple-fault testability of sequential circuits difficult is the extremely large number of multiple faults. The large number of multiple faults makes the multiple-fault simulation of circuits extremely difficult, if not impossible. Even though significant progress has been made recently [77], only moderately sized combinational circuits can be multiple-fault simulated currently. What this implies in the context of sequential circuits is that test generation for multiple faults in sequential circuits is impossible. Therefore, any procedure that synthesizes a sequential circuit for full multiple-fault testability without providing the multiple-fault test set for the circuit is meaningless. To obtain the test set as a by-product of the synthesis, further restrictions on the STG and its implementation become necessary. Well established test generation procedures for single stuck-at faults are available today [78, 50]. An approach to obtaining the multiple-fault test set is to synthesize the FSM in such a manner that the *complete* multiple-fault test set can be derived directly from the single-fault test set. This approach is explored further in the sequel.

6.5 Fully Multiple-Fault-Testable Sequential Circuits

The results of [67] and [56] can be used to show that for a certain class of sequential circuits, complete single-fault testability implies complete multiple-fault testability. Specifically, it can be shown that the set of test sequences for testing multiple faults in these sequential circuits can be derived directly from the test sequences for single faults. It is important to note that no explicit test generation is required for obtaining the multiple-

[1] However, it is conjectured that this is indeed sufficient.

fault test set for the sequential circuits synthesized in this manner. The following theorem is a statement of a sufficient set of requirements on the class of FSMs and on the synthesis procedure so that full multiple-fault coverage is obtained with the multiple-fault test set generated as a by-product of the synthesis:

Theorem 6.5.1 *In a combinationally multiple-fault testable two-level or algebraically optimized implementation of a FSM in which all next-state lines are outputs, one is guaranteed that for any multiple fault there exists a single fault such that at least the sequence that can be used for detecting that single fault can also be used to detect the multiple fault.*

Proof. The proof follows from Lemma 6.3.1 which relates the single and multiple-fault test sets for combinational circuits. By Lemma 6.3.1 one is guaranteed that given a multiple fault forced on the logic, at least one of the edges corrupted in the STG is also corrupted when some single fault is forced on the logic. Say that the sequence $T = \{i_1@s_1, i_2@s_2, ..., i_n@s_n\}$ detects that single fault in the sequential circuit. Since all the next-state lines are outputs and since T is a test sequence for the single fault, the edge $i_n@s_n$ is obviously corrupted in the circuit with the single fault. Therefore, this edge is also corrupted by the multiple fault. If the sequence T is applied in the presence of that multiple fault, one is guaranteed that either the edge $i_n@s_n$ or some edge in T prior to it will be corrupted. In either case, the multiple fault is detected. □

The following corollary follows directly from Theorem 6.5.1:

Corollary 6.5.1 *A* 100% *single-fault testable sequential circuit with all the next-state lines being the outputs and with the two-level or algebraically optimized combinational logic* 100% *multiple-fault testable is* 100% *multiple-fault testable with a test set that is equivalent to the single-fault test set.*

The above corollary can be proven for the case where all the present state lines are primary outputs as well. While this synthesis procedure is interesting in that it relates the single-fault testability of a certain class of FSMs to multiple-fault testability, arbitrary FSMs cannot be transformed into FSMs in which next states are outputs. In the following section, a synthesis procedure that removes the restriction that next-state be outputs at the expense of a minor reduction in the multiple-fault coverage that can be guaranteed is explored.

6.6 Highly Multiple-Fault-Testable Sequential Circuits

In this section, a synthesis procedure for obtaining high multiple-fault coverage in sequential circuits is outlined.

Theorem 6.6.1 *Consider a sequential circuit with the following properties. The next-state logic and the primary output logic are realized by distinct blocks. The STG for the sequential circuit has 2^k states (k is the number of encoding bits or flip-flops), and any two states have a different output for at least one input combination. The combinational logic blocks, which are either two-level or algebraically optimized multilevel circuits, are fully testable for multiple faults. If such a sequential circuit is fully testable for single faults, it is fully testable for those multiple faults which are contained completely in either the next-state logic block or the output logic block.*

Proof. Consider a multiple fault m_o that is completely contained in the output logic block. It is known that all single faults in the output logic block are fully testable. It is also known from Lemma 6.3.1 that there exists a single fault such that if the output on edge $i_i@s_i$ is corrupted by that single fault, it is also corrupted by m_o. Since the state s_i is justifiable in the true machine and since the next-state logic block is untouched by m_o, the state s_i is also justifiable in the machine with the multiple fault. Therefore, the sequential circuit is fully testable for multiple faults contained entirely in the output logic block.

Now consider a multiple fault m_n contained entirely within the next-state logic block. It is known that there exists a single fault such that all the edges corrupted as a result of that single fault are also corrupted by m_n. Say that the test sequence $T = \{i_1@s_1, i_2@s_2 \ldots, i_n@s_n'\}$ was used to detect the single fault. This implies that the next-state was corrupted on the edge $i_{n-1}@s_{n-1}$, say from s_n to s_n'. The faulty state s_n' was distinguished from the true state by means of the input i_n.

There are two possibilities when the multiple fault m_n is forced on the circuit. The first is that only the edge $i_{n-1}@s_{n-1}$ in T is corrupted by m_n (this is guaranteed by Theorem 6.3.1). The justification sequence $J = \{i_1@s_1, i_2@s_2, ..., i_{n-1}@s_{n-1}\}$ can then be used to reach the state s_{n-1} and the true next-state s_n and the faulty next state s_n'' can be differentiated in a single cycle since the output logic block is not corrupted by the logic. But since it is not known which state the edge $i_{n-1}@s_{n-1}$ is corrupted to by m_n, and because multiple-fault simulation to find out the corrupted next-state is too time-consuming, one has

to attempt to distinguish between s_n and all the possible corrupted next-states. Therefore, in the case that only the edge $i_{n-1}@s_{n-1}$ in T is corrupted, a set of $2^k - 1$ test sequences is required, in the worst case, to guarantee that the multiple fault m_n is detected. Typically, a single vector can be used to differentiate s_n and several other next-states.

The second possibility is that some edge prior to $i_{n-1}@s_{n-1}$ in T may have been corrupted. In general one has to assume that any edge in T may have been corrupted, and (since multiple-fault simulation is impossible) based on that assumption an attempt has to be made to distinguish between the true next-state at the end of any edge and all the other states in the STG. If that is done, the multiple fault m_n is guaranteed to be detected. □

The STG in the above procedure is required to have 2^k states so that a multiple fault in the next-state logic block can only corrupt an edge to a valid state. The restriction that an edge be corrupted only to a valid state is required because typically only the outputs of valid states are specified. This restriction on the number of valid states can be relaxed since the next-state and output logic blocks are implemented by distinct blocks. The following synthesis procedure illustrates a method for realizing highly multiple-fault testable sequential circuits:

1. The FSM is assumed to be one in which each pair of valid states assert a different output for at least one input combination. If a pair does not satisfy this condition and the STG is incompletely specified, edges are added to make the FSM satisfy the property. If the STG is completely specified, a new input can be added to make the STG unspecified and the necessary edges can then be added to make the FSM satisfy the required property.

2. The states of the FSM are encoded for area optimality. The next-state and output logic blocks are realized separately.

3. The output logic block (implemented in two-levels) is then optimized for area with the invalid states used as don't-cares and the combinational logic is made fully multiple-fault irredundant. Once the optimization of the output logic block is complete, the largest set Φ of invalid states is found such that each state in Φ has a different output for some input combination from every valid state.

4. Once the output logic has been minimized, the output asserted by every state for any input combination is known exactly. The invalid states not in Φ are located. Each of the invalid states not in Φ is made valid by using the edges in the STG with unspecified next-states. The primary outputs asserted on these edges are left unchanged. If no edge with unspecified next-states is available, go back to Step 1, add an extra input and repeat the entire procedure.

5. If this is the first pass through the procedure, go back to Step 1 with the new set of valid and invalid states. If it is not the first pass, then go to Step 6 if Φ is unchanged from the previous pass. If Φ is found to have changed, return to Step 1.

6. The next-state logic (implemented in two levels) is then optimized with the invalid states in Φ used as don't-cares, and the combinational logic is made multiple-fault irredundant.

7. Multilevel representations are obtained for the output and next-state logic blocks by algebraic optimization techniques.

Theorem 6.6.2 *The procedure outlined above results in a sequential circuit that is fully testable for multiple faults contained entirely in either the next-state or output logic blocks.*

Proof. Since invalid states are used as don't-cares in the optimization of the output logic, no invalid state is required as a test vector in order to test a multiple fault in the output logic block. Therefore, since the output logic block is combinationally fully multiple-fault irredundant, it is also sequentially fully multiple-fault irredundant.

At the end of the synthesis procedure, the only invalid states are those that have a different output from every valid state for at least one input combination. A multiple fault in the next-state block either corrupts an edge to a valid state (in which case the faulty and true next-states can be distinguished in the manner explained in the proof of Theorem 6.6.1) or to an invalid state. Since the invalid state is guaranteed to be distinguishable from the true state, the multiple fault is guaranteed to be detectable. □

The testing scenario under the above synthesis procedure is the following. Once the synthesis procedure has completed, the single-fault test set for the sequential circuit is obtained using STEED [50]. In addition, the primary input vectors that distinguish between

each pair of valid states and the primary input vectors that distinguish between each valid state and each invalid state are obtained. In the worst case, the total number of such vectors is $N_v(N_v - 1)/2 + N_{iv}N_v$, where N_v is the number of valid states and N_{iv} is the number of invalid states. The testing of the circuit is done in the following manner. For every test sequence in the single-fault test set, the true next-state on every edge is known. Therefore, after the application of every edge in a test sequence, all the primary input vectors that distinguish between the true valid next-state and the other valid/invalid states are applied. Essentially, this implies that each sub-sequence in a test sequence has to be applied $2^{N_b} - 1$ times in the worst case, where N_b is the number of encoding bits. However, in practice, this number is significantly less than 2^{N_b} since one can differentiate a given state from several other states with the same input vector. In this manner, at the expense of increased testing time and a larger test set, one can ensure close to 100% multiple-fault coverage.

It remains to clearly define how close the fault coverage that can be guaranteed by this procedure is to 100% . One is guaranteed that the multiple faults contained entirely in either the next-state or the primary output logic blocks are fully testable by this procedure. Also detectable are those multiple faults spanning both logic blocks that are *disjoint.* A disjoint multiple fault is a fault such that the part of the multiple fault in the output logic block does not affect at least one edge that is used to distinguish between some true and faulty next-state pair that arises as a result of the part of the multiple fault in the next-state logic block. While it is conjectured that this would lead to 100% multiple-fault coverage in practice, the conjecture cannot, unfortunately, be verified due to the inability to do comprehensive multiple-fault simulation.

6.7 Conclusions

In this chapter, the property that for any multiple fault in a two-level or algebraically optimized multilevel circuit there is a corresponding single fault such that all the tests for the single fault are a subset of the tests for the multiple fault was exploited. Using this property, synthesis procedures for fully and highly multiple-fault testable sequential circuits were devised. The synthesis procedures are such that the test-set is generated as a by-product and no explicit test generation is required.

While the synthesis procedure works for arbitrary FSMs, a synthesis procedure that would obviate the need for pairs of states to be distinguishable in a single clock cycle

is definitely more desirable. Developing such a procedure is a topic for future research.

Chapter 7

Logic Verification Using General BDDs

7.1 Introduction

Reduced, ordered Binary Decision Diagrams (OBDDs) [23] have gained widespread use in the areas of combinational and sequential logic verification (*e.g.* [80, 28]) due to their canonicality and easy manipulability.

Since OBDDs are a canonical form, verifying the equivalence of two combinational logic functions involves selecting a common input ordering for the two circuits, constructing OBDDs for each of the circuits, and checking to see if the two OBDDs are isomorphic. Satisfiability checking simply corresponds to comparing the OBDD of a given circuit, under any input ordering to the constant 0 OBDD.

Finding a good input ordering that produces OBDDs of manageable size is a difficult problem, and has received considerable attention (*e.g.* [80, 46, 11]). However, there are classes of combinational circuits, notably multipliers, for which OBDDs, under any possible input ordering, have a provably exponential size. Circuits, other than multipliers, have been encountered where finding an input ordering to obtain an OBDD of manageable size has not been possible thus far. In fact, it is quite easy to construct simple examples, where OBDD sizes grow exponentially with the number of inputs to the example.

There has been some work in the generalization of BDDs to verify larger classes of circuits. The method of Friedman [45] uses pBDDs which are BDDs where variables can

appear in any order along a path from the root to a leaf. Equivalence checking for pBDDs is NP-complete, and the CPU time required by his method appears to grow exponentially in the order of the output bit that is verified in a multiplier. The method of Simonis [101] uses constraint logic programming to verify classes of multipliers. It does not appear that this method can be easily generalized to verify circuits that contain multipliers. Burch, in [24], showed that the replication of inputs to a $n \times n$ multiplier to obtain a circuit with $2n^2$ inputs would result in an OBDD of $O(n^3)$ size under a high-to-low input ordering. The work of [24], however, requires that there be a correspondence between the replicated inputs of the logic-level implementations of the circuits that are being checked for equivalence. This is not generally possible in a synthesis scenario, where the logic may be restructured quite dramatically.

In this chapter, it is shown how general BDDs, *i.e.* BDDs where input variables are allowed to appear multiple times along any path in the BDD, can be used to check for Boolean satisfiability. General BDDs are constructed by replicating inputs in a given circuit η to obtain a new circuit η'. After choosing an appropriate ordering for the inputs to η', OBDD construction algorithms are used to obtain a general BDD for η.

The satisfiability checking strategy presented here is based on an *input smoothing* operation on general BDDs. After all the inputs have been smoothed away, if the function reduces to a constant 1, then it means that the original function was satisfiable. In order to verify the equivalence of two functions, f_1 and f_2, $f_1 \oplus f_2$ is checked for satisfiability.

Various ways of smoothing inputs in general BDDs are described. One input smoothing strategy is based on cofactoring the general BDD with respect to cubes and OR'ing the results. This strategy has to be embellished in order not to be memory intensive. A general depth-first branching strategy to smooth the inputs of a general BDD is presented. Using this strategy, it is possible to trade off CPU time versus memory requirements. The order of smoothing input variables is also important and has to be intelligently chosen. A second strategy is based on a circuit transformational method, wherein a multiplexor-based circuit derived from a BDD is modified, and a new BDD corresponding to the smoothed function is reconstructed from the modified circuit.

It is shown how compact transition relations based on general BDD representations can be constructed for sequential circuits, and use the general-BDD input smoothing strategy to traverse the state space of a sequential machine. This state space traversal technique can be used to verify different logic-level implementations of sequential circuits.

General BDDs are not a canonical form and are a much more powerful representation than OBDDs. For example, it has been shown that a general BDD of $O(n^3)$ size can be constructed for a $n \times n$ multiplier. Using general BDDs, it has been able to perform equivalence checking on combinational logic implementations of the most significant bit of a parallel 16 × 16 multiplier, a single output modified Achilles heel function and a complex ALU, *without requiring any additional information* other than the given logic-level descriptions. It was possible to traverse the state space of sequential logic implementations corresponding to a parallel 16 × 16 multiplier, again without requiring any additional information other than the given logic-level description. The memory requirements to verify other classes of circuits is also dramatically reduced using general BDDs, as opposed to OBDDs.

The basic notation used is described in Section 7.2. In Section 7.4, the basic strategy for satisfiability checking and combinational logic verification using general BDDs is described. Input smoothing in general BDDs is treated in Section 7.5. State space traversal using transition relations that are based on general BDDs is the subject of Section 7.6. Input replication and ordering algorithms are presented in Section 7.7. Experimental results are presented in Section 7.8.

7.2 Basic Definitions

7.2.1 Binary Decision Diagrams

The definition of reduced, ordered Binary Decision Diagrams as described in [23] is presented below.

Definition 1: A Binary Decision Diagram is a rooted, directed graph with vertex set V containing two types of vertices. A *nonterminal* vertex v has as attributes an argument index $index(v) \in \{1, \cdots, n\}$ and two children $low(v)$, $high(v) \in V$. A *terminal* vertex v has as attribute a value $value(v) \in \{0, 1\}$.

The correspondence between Binary Decision Diagrams and Boolean functions is defined as follows:

Definition 2: A Binary Decision Diagram G having root vertex v denotes a function f_v defined recursively as:

1. If v is a terminal vertex:

(a) If $value(v) = 1$, then $f_v = 1$.

(b) If $value(v) = 0$, then $f_v = 0$.

2. If v is a nonterminal vertex with $index(v) = i$, then f_v is the function:

$$f_v(x_1, \cdots, x_n) = \overline{x_i} \cdot f_{low(v)}(x_1, \cdots, x_n) + x_i \cdot f_{high(v)}(x_1, \cdots, x_n)$$

x_i is called the *decision variable* for vertex v.

The following additional properties are required in reduced, ordered Binary Decision Diagrams:

1. In Definition 1, place the restriction that for any nonterminal vertex v, if $low(v)$ is also nonterminal, then one must have $index(v) < index(low(v))$. Similarly, if $high(v)$ is also nonterminal, then one must have $index(v) < index(high(v))$.

2. A *reduced* BDD is one in which $low(v) \neq high(v)$ for any vertex v and no two subgraphs in the BDD are identical.

Note that Property 1 ensures that in any path from root to leaf in the BDD, a variable can appear at most once. The above two properties result in reduced, ordered Binary Decision Diagrams being a canonical form (OBDDs) [23].

7.2.2 Binary Decision Diagram Operations

Let $f : B^n \rightarrow B$ be a Boolean function, and $x = (x_1, \ldots, x_k)$ a set of input variables of f. The smoothing of f by x is defined as:

$$\begin{aligned} S_x f &= S_{x_1} \ldots S_{x_k} f \\ S_{x_i} f &= f_{x_i} + f_{\overline{x_i}} \end{aligned}$$

where f_a designates the cofactor [18] of f by the literal a.

The smoothing operator can be computed very efficiently on OBDDs. For details the reader is referred to [28]. The concern here is with input smoothing on general BDDs, which is more complicated.

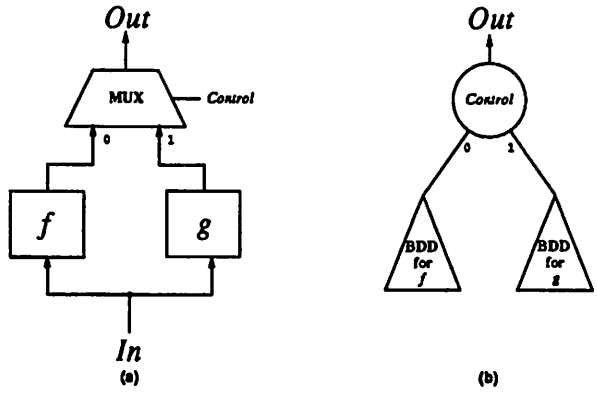


Figure 7.1: Relaxing the ordering constraint

7.3 General BDDs

In General BDDs a variable is allowed to appear multiple times along a path from the root of the BDD to the leaves. There are two basic reasons why this freedom leads to a more compact representation. The two reasons are illustrated in this section by means of examples.

A corollary of allowing a variable to appear multiple times along a path in the BDD is that the ordering constraint does not exist any more. Therefore, the same set of variables can appear in a different sequence along different paths in the BDD. Consider the example shown in Figure 7.1. The output of the circuit in Figure 7.1(a) is equal to the function f when the *control* input is 0, and is equal to the output of g when the *control* input is 1. The functions f and g have common support. Say that f requires a completely different variable ordering from the variable ordering required for g in order that the OBDD size be polynomial for both f and g, and also that there is no single variable ordering that would realize a reasonable sized OBDD for both f and g. In order to build an OBDD for *out*, one has to effectively build the OBDD for both f and g under the same variable ordering. Since that is not possible, an OBDD cannot be built for *out*. On the other hand, when making a general BDD for the circuit, one has the freedom of choosing the ordering for f independently of the ordering chosen for g. As a result, it is possible to make a general BDD for *out*.

The second reason that general BDDs lead to compact representations has to do specifically with the fact that a variable is allowed to appear multiple times along a path in

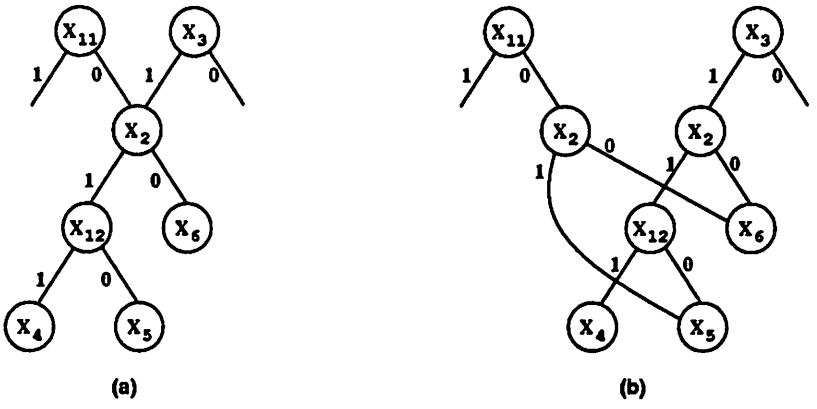


Figure 7.2: Multiple instances of a variable along a BDD path

the BDD. Consider the example of the BDD fragment shown in Figure 7.2(a). The variable x_1 appears twice along one of the paths in the BDD. The multiple instances of x_1 in that path are labeled x_{11} and x_{12}. Focus attention on the node for variable x_2. This node is reached from two other nodes in the BDD, *viz* from the node x_{11} and the node x_3. When x_2 is reached from node x_{11}, the path from x_2 to x_4 is meaningless because in reaching x_2 from x_{11}, a decision has already been made to set x_1 to a 0. But, the same path (from x_2 to x_4 through x_{12}) is required when x_2 is reached from x_3. If the node for x_{12} were not present, then the node for x_2 would have to be duplicated as shown in Figure 7.2(b). In the BDD fragment shown here, x_2 is reached from only two other paths. In general, a node could have very large fanin and not allowing a variable to appear multiple times could cause a large amount of duplication of nodes.

7.4 Satisfiability/Equivalence Checking Via Input Smoothing

One is given a logic circuit f to check for satisfiability. Assume that one cannot construct a manageable OBDD for f, due to memory or CPU time restrictions. The strategy in such a case is simple. The inputs to f are replicated to obtain a different circuit f', keeping track of what inputs to f' are derived from the same input to f. Replicating the inputs to an example circuit is shown in Figure 7.3 where the input x_3 in Figure 7.3(a) has been replicated to $x_{3,0}$ and $x_{3,1}$ in Figure 7.3(b).

If enough inputs are replicated and a good ordering of replicated inputs is found,

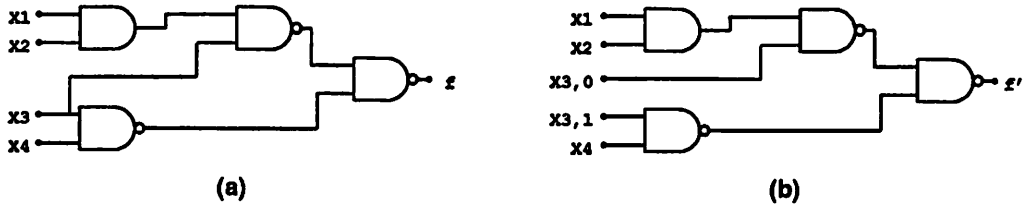


Figure 7.3: Example of input replication

one can use OBDD construction algorithms [23] to construct an OBDD for f', which can be viewed a general BDD, G, for the circuit f (if the replicated inputs are coalesced). Note that the general BDD is *not* a canonical form for the function f. For example, function f may not be satisfiable, but G may be quite large.

The inputs to f' can now be sequentially smoothed away. An input to f, say x_i, is chosen. Assume that x_i has been replicated n times in f', and say the replicated inputs correspond to $x_{i,0}$, ..., $x_{i,n-1}$. One has to smooth away the $x_{i,0}$, ..., $x_{i,n-1}$ inputs to f', making sure that they have the same values (0 or 1). The smoothing of f' by x_i (whose replicated instances are $x_{i,0}$, ... $x_{i,n-1}$) is defined as:

$$S_{x_i} f' = f'_{x_{i,0} \cdot x_{i,1} \cdots x_{i,n-1}} + f'_{\overline{x_{i,0}} \cdot \overline{x_{i,1}} \cdots \overline{x_{i,n-1}}}$$

Essentially, one is cofactoring f' with respect to the cube corresponding to all the $x_{i,j}$'s set to 1, and with the cube corresponding to all the $x_{i,j}$'s set to 0 and OR'ing the results.

As before one can define:

$$S_x f' = S_{x_0} \ldots S_{x_{k-1}} f'$$

to sequentially smooth away the k sets of inputs to f', corresponding to the distinct inputs to f.

This smoothing strategy can use highly efficient OBDD manipulation algorithms. If during the smoothing, or after smoothing away all the inputs, a constant 1 function corresponding to $S_x f'$ is obtained, then it means that f is satisfiable. This is because an input combination has been found that sets f' to a 1 under the constraint that the replicated inputs have the same consistent values. This in turn means that an input combination that sets f to a 1 has been found. On the other hand, if a constant 0 is obtained function, then it means that f was not satisfiable.

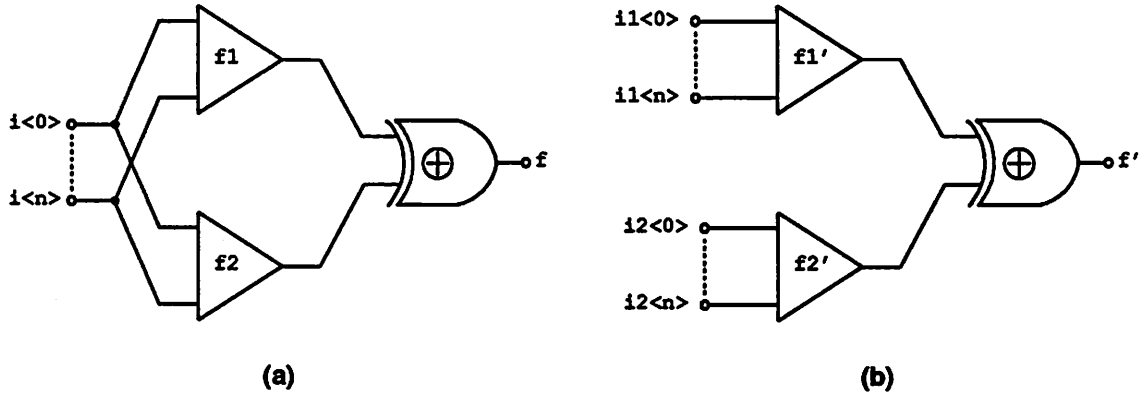


Figure 7.4: Equivalence checking

7.4.1 Equivalence Checking

In order to check for the equivalence of two logic functions f_1 and f_2, $f_1 \oplus f_2$ is constructed as shown in Figure 7.4(a). This composite function is treated as the f function above, and the inputs are replicated as needed to construct a general BDD for f. The very first step of replication will break up the inputs to f_1 and f_2 so they have disjoint support in f'. This is shown in Figure 7.4(b). This implies that if OBDDs were constructible for f_1 and f_2, a general BDD will be constructible for $f_1' \oplus f_2'$, under an input ordering where the (ordered) set of inputs to f_1' is followed by the (ordered) set of inputs to f_2'. Of course, if OBDDs are constructible for f_1 and f_2, there is no need to use general BDDs to verify the equivalence of f_1 and f_2. The use of general BDDs, however, allows the further replication of the inputs to f_1' and f_2' so a manageable-sized general BDD can be constructed for f', even in the case where OBDDs cannot be constructed for f_1 or f_2.

7.5 Input Smoothing in General Binary Decision Diagrams

Algorithms for input smoothing in general BDDs are presented in this section.

7.5.1 A Branching Strategy for Smoothing Replicated Inputs

While input smoothing in an OBDD is not guaranteed to reduce the size of the OBDD, efficient algorithms exist [104] that can smooth away a set of inputs in an OBDD by making a single pass over the OBDD. Input smoothing in general BDDs is more complicated, and can increase the size of the general BDD. If there are unreplicated inputs in the general

BDD, then these are smoothed away first to obtain, in most cases, a smaller BDD, which is then checked for satisfiability.

Experimental evidence indicates that given a set of inputs to be smoothed, (where each input has been replicated), the order in which the inputs are smoothed can have a significant effect on the resulting general BDD. Further, the size of a general BDD that is obtained after smoothing away even a single set of replicated inputs can be substantially larger than the original BDD. In general, a massive replication of inputs in a general BDD makes smoothing more difficult. This is intuitive, since the smoothed BDD will be substantially different from the original general BDD. Therefore the general branching algorithm, shown below, that operates in a depth-first manner was devised.

In the routine **satisfiable**(), G is the general BDD, and X is the set of inputs to be smoothed away, so as to check for satisfiability. Each $x_i \in X$ consists of $x_{i,0}, \ldots, x_{i,n_i}$ replicated inputs. This routine can be seen as an attempt to interleave two methods for satisfiability checking, BDDs and SAT.

satisfiable(G, X):
{
    if ($X = \phi$) {
        if ($G \equiv 0$) **return**(FALSE) ;
        else if ($G \equiv 1$) **return**(TRUE) ;
    }
    x_p = **select-input** (G, X) ;
    Compute $G_{x_p} = G_{x_{p,0} \cdot x_{p,1} \cdots x_{p,n_p}}$;
    Compute $G_{\overline{x_p}} = G_{\overline{x_{p,0}} \cdot \overline{x_{p,1}} \cdots \overline{x_{p,n_p}}}$;
    if (G' = **bdd-or** (G_{x_p}, $G_{\overline{x_p}}$, **param1**)) {
        return (**satisfiable**(G', $X - x_p$)) ;
    }
    else {
        if (**satisfiable**(G_{x_p}, $X - x_p$)) **return**(TRUE) ;
        else if (**satisfiable**($G_{\overline{x_p}}$, $X - x_p$)) **return**(TRUE) ;
        else return(FALSE) ;
    }
}

The parameter **param1** controls the amount of branching that can take place versus the size of the smoothed BDDs, and can be set based on the amount of memory available. During the **bdd-or**() operation the size of the resulting BDD is monitored, and if its size exceeds **param1**, the **bdd-or**() operation is terminated. If the **bdd-or**() has to be

terminated, branching on the cofactors takes place. This ability to branch gives the general BDD approach the capacity to trade off CPU time for memory usage.

The routine **select-input()** selects a set of replicated inputs in the general BDD to be smoothed. Currently, a simple heuristic that has to do with the fraction of the general BDD that is affected by the smoothing is used. Smoothing with respect to x_i only affects the portion of the general BDD below $x_{i,min}$, where $x_{i,min}$ corresponds to the replicated instance of x_i that has the lowest index (is closest to the root of the general BDD) among all the replicated instances of x_i. The input x_p is selected such that $x_{p,min}$ has the maximum index among all $x_{i,min}$.

7.5.2 Smoothing by Addition of Extra Variables

One approach to avoiding a blowup during input variable smoothing is to defer the computation of the OR in

$$S_{x_i} f \;=\; f_{x_{i,0} \cdot x_{i,1} \cdots x_{i,n-1}} \;+\; f_{\overline{x_{i,0}} \cdot \overline{x_{i,1}} \cdots \overline{x_{i,n-1}}}$$

until as late as possible. This can be done by recognizing that

$$S_{x_j}\, S_{x_i} f \;=\; S_{x_j}(f_{x_{i,0} \cdot x_{i,1} \cdots x_{i,n-1}} + f_{\overline{x_{i,0}} \cdot \overline{x_{i,1}} \cdots \overline{x_{i,n-1}}}) \;=\; S_a\, S_{x_j}(a \cdot f_{x_{i,0} \cdot x_{i,1} \cdots x_{i,n-1}} + \overline{a} \cdot f_{\overline{x_{i,0}} \cdot \overline{x_{i,1}} \cdots \overline{x_{i,n-1}}})$$

where a is the extra variable whose addition makes it possible to defer the computation of the OR. Also, a is ordered so that it becomes the root of the BDD for $a \cdot f_{x_{i,0} \cdot x_{i,1} \cdots x_{i,n-1}} + \overline{a} \cdot f_{\overline{x_{i,0}} \cdot \overline{x_{i,1}} \cdots \overline{x_{i,n-1}}}$. The size of the BDD after x_i is smoothed using this method is bounded by the sum of the sizes of the BDDs for $f_{x_{i,0} \cdot x_{i,1} \cdots x_{i,n-1}}$ and $f_{\overline{x_{i,0}} \cdot \overline{x_{i,1}} \cdots \overline{x_{i,n-1}}}$. The actual change could be much lower than the sum of the sizes if there is a large amount of sharing of nodes between the BDDs for $f_{x_{i,0} \cdot x_{i,1} \cdots x_{i,n-1}}$ and $f_{\overline{x_{i,0}} \cdot \overline{x_{i,1}} \cdots \overline{x_{i,n-1}}}$.

7.5.3 Smoothing Inputs Using Circuit Transformations

In the case where unreplicated inputs exist in the general BDD, a strategy based on circuit transformations (different from the algorithm of [104]) sometimes works much better. This strategy is as follows: Given a general BDD, G, a multiplexor-based circuit η corresponding to G is derived by replacing all nodes in G by 2-input multiplexors whose control input is the decision variable corresponding to the node. In η, all the multiplexors

corresponding to the unreplicated inputs are replaced by OR [1] gates to obtain η'. Now, one can choose an ordering for the remaining inputs to η', (and coalesce replicated inputs if necessary) to construct a new general BDD for η'.

In a general BDD, one cannot simply replace each node corresponding to an instance of a replicated input by an OR gate. Doing so may make paths in the general BDD corresponding to conflicting values for different instances of a replicated variable sensitizable in the derived circuit, thus destroying functionality. However, a replicated input which does not appear more than once along any path in the general BDD can be smoothed away by replacing all its nodes by OR gates in the derived circuit.

In the case of replicated inputs appearing several times along a path, the following strategy can be used. Assume one has a general BDD G, and the input x_i has been replicated k times, the replicated instances being $x_{i,0}$, $x_{i,1}$, ..., $x_{i,k-1}$. It is now desired to transform G into G' so that all paths in G' that correspond to conflicting values for the different instances of the replicated variable x_i have zero-terminal vertices. Other than this modification, G' has the same functionality as G. One first obtains G_1 from G by making all the paths in G that correspond to a *zero* value for any of the replicated instances of x_i have zero-terminal vertices. Similarly, G_2 is obtained from G by making all the paths in G that correspond to a *one* value for any of the replicated instances of x_i have zero-terminal vertices. The OR of $G1$ and $G2$ satisfies the property required of G'. One can now obtain a circuit from G' by replacing all the nodes in the G' corresponding to the replicated instances of the variable x_i by OR gates and all the other nodes by 2-input multiplexors. The replicated primary input x_i has effectively been smoothed away in this derived circuit. A (possibly general) BDD can now be constructed from the derived circuit after picking an ordering for the remaining variables.

This technique can perform significantly better than a straightforward smoothing algorithm because of the degree of freedom in choosing a different ordering for the general BDD of η'. For example, assume that the variable x (with f being the function associated with its BDD node) is being smoothed. Straightforward smoothing within the BDD forces the BDD for $f_x + f_{\overline{x}}$ to be constructed under a global variable-ordering that is suitable for f_x and $f_{\overline{x}}$ but not necessarily for $f_x + f_{\overline{x}}$, leading to a much larger BDD after smoothing.

[1] The use of complemented nodes [13] in a BDD makes this transformation slightly more complicated. A process of pushing the inversions all the way back to the terminal vertices while deriving the circuit is necessary, and the derived circuit can be twice as large as the original BDD.

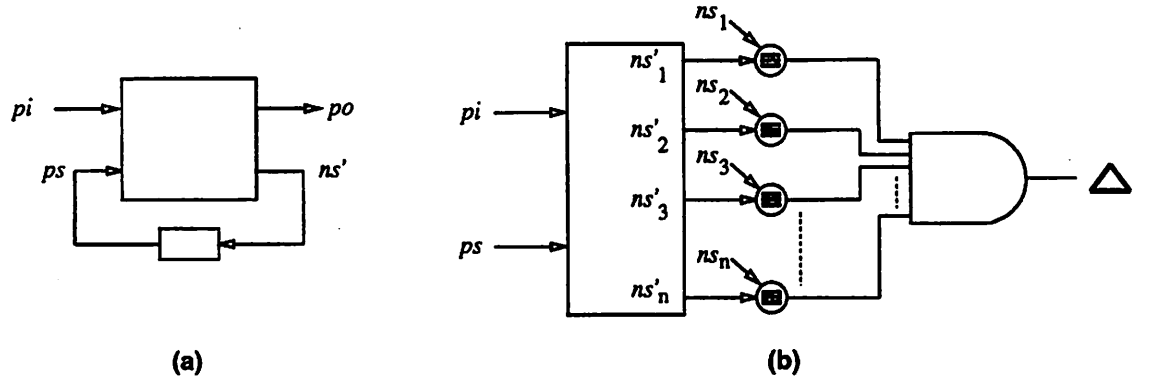


Figure 7.5: A Sequential circuit and its transition relation

While this problem is usually not encountered in the case where no replicated inputs are present, it assumes great significance when they are.

7.6 Implicit State Space Traversal Using General BDDs

Verifying the equivalence of two sequential machines, A and B, involves traversing the state space of the product machine, $A \times B$. An efficient approach to sequential machine state space traversal using ordered Binary Decision Diagrams (OBDDs) was presented in [25, 28]. The use of a *transition relation* allowed the implicit breadth-first enumeration of states in a sequential machine, enabling the state traversal of circuits of significant size. Building the transition relation is often the bottleneck in this class of approaches – under an OBDD representation it is sometimes not possible to build the transition relation, even if OBDD representations of the combinational logic of the sequential machine are of manageable size.

In this section, after first reviewing the transition relation method, it is shown how general BDDs can be used to traverse the state space of a sequential machine.

7.6.1 The Transition Relation Method

A sequential circuit is shown in Figure 7.5(a). The next state lines ns'_i, $0 \leq i < N$ are functions of the pi_k, $0 \leq k < M$ and the ps_j, $0 \leq j < N$. The transition relation $\Delta(pi,\ ps,\ ns)$ is a single-output Boolean function shown in Figure 7.5(b). It is a function that is a 1 if and only if a primary input pi, a present state ps and a pseudo next state ns are applied such that the next state ns' produced by the primary input and present state

is equal to the pseudo next state ns.

Assume one has a set of states S_i and one wishes to compute the set of states S_{i+1} that are reachable in one clock cycle from S_i. The operations performed on Δ are as follows:

1. Smooth away the pi variables from $\Delta(pi,\ ps,\ ns)$ to obtain $\Delta'(ps,\ ns)$.
2. $\Delta'' = \Delta'(ps,\ ns) \cap S_i(ps)$, *i.e.* intersect the transition relation with the given present states. It may improve efficiency to intersect Δ' with $\overline{S_i(ns)}$, *i.e.* $\Delta'' = \Delta'(ps,\ ns) \cap S_i(ps) \cap \overline{S_i(ns)}$. One is not interested in reaching the $S_i(ns)$ states again, since it is known that they are reachable. The support of the $S_i(ps)$ lines is changed to be the pseudo next state lines, in order to obtain $S_i(ns)$.
3. Smooth away the ps variables from $\Delta''(ns)$ to obtain $S_{i+1}(ns)$. $i = i + 1$. Go to Step 2.

Since smoothing in OBDDs generally reduces the size of the OBDD, if the transition relation is constructible, one can typically traverse the entire state space of the sequential machine. The entire state space is traversed when at some iteration n, $S_n = S_{n+1}$, or no new state is encountered at some iteration.

7.6.2 Using General Binary Decision Diagrams

One can use general BDDs to perform an efficient state traversal of sequential circuits. For certain classes of circuits OBDDs may not be constructible for some next state lines under any ordering of the primary input and present state variables. In other circuits, it may not be possible to construct OBDDs for all the next state lines using the same ordering of the primary input and present state variables. Further, since the function Δ tends to be very complex, even if OBDD representations are available for the next state lines, an OBDD representation for Δ may not be constructible.

The use of general BDDs leads to the following useful property. Given general BDDs of the next state lines, one can always construct a general BDD for Δ because the primary inputs and present state lines can be replicated *across* the different next state functions, ns'_i, $0 \leq i < N$. The size of the general BDD corresponding to the product of the N XNOR's of the pseudo next-state lines with the associated next state functions is then of the order of the sum of the sizes of the general BDDs corresponding to the various

$ns'_i \equiv ns_i$, $0 \leq i < N$, given an appropriate variable ordering for the replicated inputs. However, as mentioned earlier, such massive replication of inputs should be used with caution since it can make the smoothing operation on general BDDs more time consuming. Typically, a moderate replication of inputs across the next state lines results in a manageable general-BDD representation of the Δ function on which input smoothing can be performed. The above strategy of replicating inputs across the next state lines, proves useful when OBDD representations can be constructed for the next state lines, but the Δ function is too large for efficient state traversal.

The fact that the pseudo next state lines are unreplicated and the present state lines replicated is handled in the following manner in Step 2 of the traversal algorithm above. At Step 2, one has to intersect an OBDD representation of $S_i(ps)$ (either obtained from the previous iteration by a change of variables from ns to ps, or given as a reset state), with the $\Delta'(ps, ns)$ general BDD, wherein the ps lines may have been replicated. This is trivially accomplished by implicitly expanding the support of the $S_i(ps)$ OBDD to the ps support of the general BDD for $\Delta'(ps, ns)$. For example, consider a circuit with three ps lines, ps_0, ps_1 and ps_2. Assume that an OBDD representation of the states 100, 010 in $S_1(ps)$ is given. Also assume that each of the ps lines has been replicated twice in $\Delta'(ps, ns)$, under the ordering $ps_{1,0}$, $ps_{2,0}$, $ps_{1,1}$, $ps_{3,0}$, $ps_{2,1}$, $ps_{3,1}$. By expanding the support of $S_1(ps)$, one will effectively have obtained the states $10--0--$, $01--0--$ in general BDD form. These states can now be intersected with $\Delta'(ps, ns)$. Note that one are only interested in the states 10100, 010010. However, during the smoothing of the replicated ps lines in Step 3. the constraint that $ps_{i,0} == ps_{i,1}$, $1 \leq i \leq 3$ will be imposed.

Once the general BDD for the transition relation has been obtained, the input smoothing algorithms described in the previous sections for replicated and unreplicated inputs can be used to traverse the machine. In some cases building the complete transition relation even in the form of a general BDD may not be feasible or may require excessive replication (making the smoothing of the replicated variables cumbersome). In such situations, techniques similar to those described in [104] for avoiding the explicit construction of the transition relation and for smoothing variables as early as possible can be applied.

7.6.3 Variant Methods

It is possible that OBDDs are constructible for each next state line and/or primary output of a sequential circuit under different orderings of each output, but not under the restriction that the same ordering be used for each output. Further, even if OBDDs are constructible under the same ordering for each of the next state lines, the transition relation may be too large to construct. In either of these cases, the following method to construct a general BDD for the transition relation is viable.

1. Construct OBDDs under possibly different orderings for each next state line ns'_i, $0 \leq i < N$.

2. Find a compatible ordering for all the ns'_i, by replicating pi or ps lines *across* the ns'_i.

3. Compute the transition relation $\Delta(pi, ps, ns)$ as a general BDD under the compatible ordering.

Finding a compatible ordering simply corresponds to replicating inputs whenever there is a conflict in the ordering of any set of inputs across the next state lines. Replication can be kept to a minimum by only considering major conflicts between the ordering of sets of inputs, and disregarding minor interchanges in input orders. For instance, consider the lines ns'_1 and ns'_2, which have as inputs pi_1, pi_2 and ps_1, ps_2. Assume that the optimal order for ns'_1 is pi_1, ps_1, ps_2, pi_2, and that for ns'_2 is pi_2, ps_1, ps_2, pi_1. One can obtain a compatible ordering for the two lines by replicating pi_1, and obtaining two lines $pi_{1,0}$ and $pi_{1,1}$. The compatible ordering is $pi_{1,0}$, ps_1, ps_2, pi_2, $pi_{1,1}$. ns'_1 will have $pi_{1,0}$, pi_2, ps_1, ps_2 as its support and ns'_2 will have $pi_{1,1}$, pi_2, ps_1, ps_2 as its support. The size of the ns'_i general BDDs will be the same as the size of the OBDDs under an optimal ordering. Further, one can manipulate these general BDDs simultaneously; for instance, one can AND these general BDDs to compute the transition relation.

7.7 Replicating and Ordering Circuit Inputs

7.7.1 Replicating Inputs to a Multiplier

First the replication strategy for a $n \times n$ parallel multiplier presented in [24] is reviewed. It was shown there that replicating each of the $2n$ inputs n times could result in

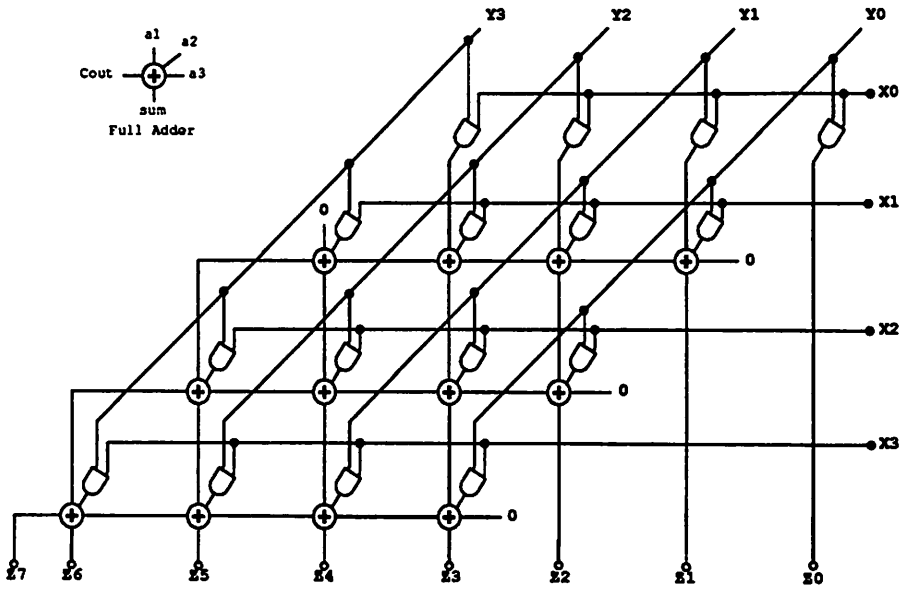


Figure 7.6: A 4×4 multiplier

a general BDD of $O(n^3)$ size under a particular input ordering.

The original multiplier and the replicated-input multiplier are shown in Figure 7.6 and Figure 7.7 respectively.

Burch [24] verifies multipliers by replicating the inputs to the two given multipliers m_1 and m_2 in a certain way, to obtain m_1' and m_2'. He then constructs OBDDs for m_1' and m_2'. Verification entails checking for isomorphism between the general BDDs, and therefore an *exact* correspondence needs to be established between the replicated inputs of multiplier m_1' and the replicated inputs to multiplier m_2'. This correspondence can, typically, only be established if m_1 and m_2 initially have a constrained and similar structure.

There is no need of a correspondence between replicated inputs in the general-BDD approach – one can use different replications for the m_1 and m_2 functions, and one only needs to keep track during replication of what replicated inputs correspond to the original inputs (information needed during smoothing). This approach does not depend on the canonical nature of the OBDDs, it merely uses efficient OBDD manipulation algorithms to check $m_1 \oplus m_2$ for satisfiability. It should be noted that it is not necessary need to find exactly the same replication of Figure 7.7, nor exactly the high-to-low ordering that was used in [24]. Finding a reasonably close replication and ordering will result in a general BDD of manageable size, though it may not be of $O(n^3)$ size.

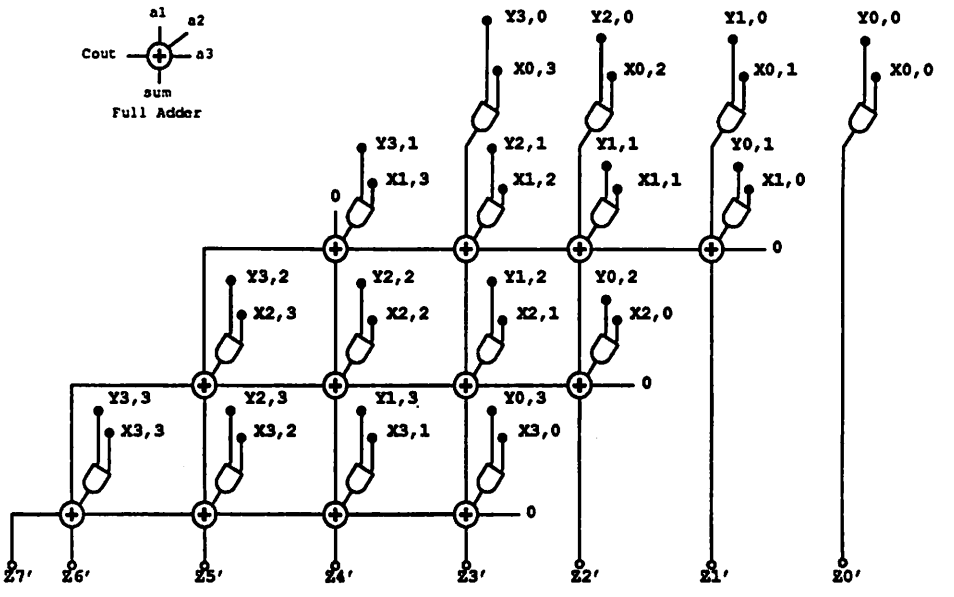


Figure 7.7: A 4 × 4 multiplier with inputs replicated

7.7.2 A General Algorithm to Replicate and Order Inputs

In many cases, OBDDs are completely different structurally and much larger than the circuit itself. In such cases, the use of appropriately constructed general BDDs can lead to graphs that are significantly smaller than the OBDDs and are structurally closer to the circuit.

The goal of attempting to make the BDD-based circuit structurally as close to the given circuit as possible guides the replication and ordering of the inputs. This is best illustrated with an example. Consider the **adder-subtractor** circuit shown in Figure 7.8. Figure 7.8(a) shows an area-inefficient implementation in which two distinct blocks are used for addition and subtraction. The desired output is then selected by the output multiplexor based on the value of the **ADSB** line. This is exactly the structure of the OBDD that would be obtained if the **ADSB** input were at the root of the OBDD. The area-optimum implementation of the **adder-subtractor** on the other hand is shown in Figure 7.8(b). This circuit is much smaller and structurally quite different from the circuit in Figure 7.8(a). A general BDD whose structure is close to that of Figure 7.8(b) can be derived in the following manner:

1. **ADSB** is replicated n times, where n is the bit-width. The replicated **ADSB** lines are labeled so that **ADSB**$< i >$ corresponds to the fanout path from **ADSB** that is

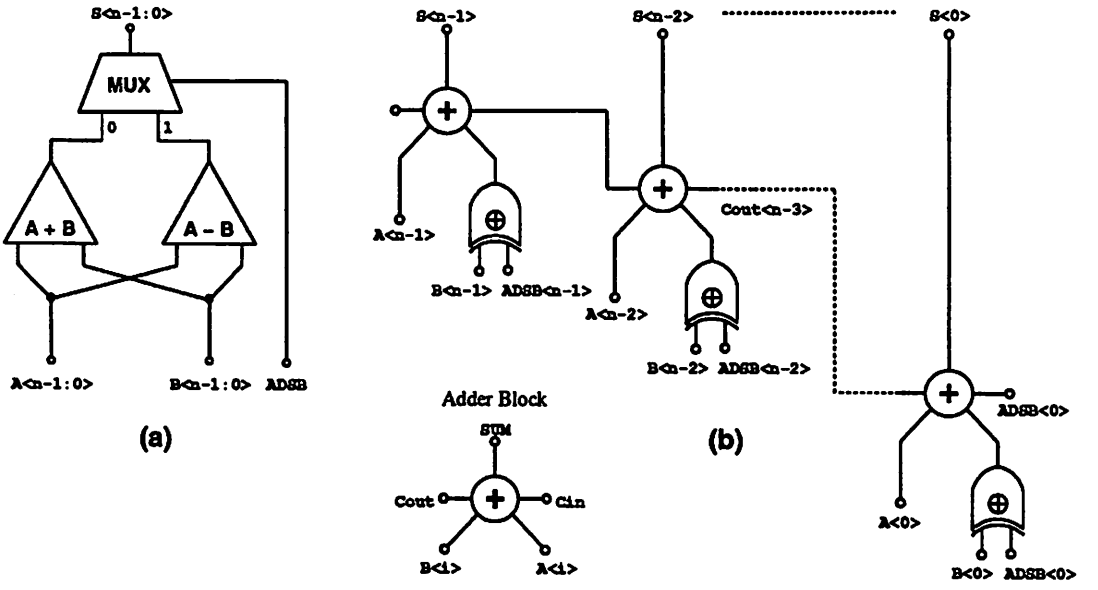


Figure 7.8: Two versions of the adder-subtractor

XOR'ed with **B**$< i >$ in Figure 7.8(b).

2. An OBDD is constructed with the ordering **B**$< n-1 >$, **ADSB**$< n-1 >$, **A**$< n-1 >$, **B**$< n-2 >$, **ADSB**$< n-2 >$, **A**$< n-2 >, \cdots,$ **B**$< 0 >$, **ADSB**$< 0 >$, **A**$< 0 >$. As a result of the replication, each bit-slice of the adder-subtractor in the general BDD has the **ADSB** input available to it (just like the optimal logic-level implementation of the adder-subtractor). Hence, the general BDD with the **ADSB** input replicated will have the same structure as the circuit of Figure 7.8(b).

The example of the **adder-subtractor** circuit leads us to a general procedure for replicating and ordering inputs. To begin with, a possibly area-optimized version of the circuit is obtained to guide the replication. In the optimized circuit, an input with a large depth (where depth is defined as the number of gates between the input and the primary output) and which fans out multiple times so that each fanout path is used at different depths in the circuit is a good candidate for replication (The **ADSB** input in the **adder-subtractor** satisfies these requirements). Such an input is replicated as many times as it fans out. Once all inputs deemed to be good candidates are replicated, either the ordering strategy of [80] or some other strategy dependent on the circuit structure can be used to construct the OBDD. When a large number of inputs have been replicated, the circuit tends to assume the form described in Lemma 1 of [80] wherein the output f can be expressed as

EX	#I	#O	Output Verified	GBDD		OBDD	
				Time[1]	BDD Size	Time[1]	OBDD Size
ach32	69	1	Output # 1	61s	7073	*exc*	*exc*
ach32-p	64	1	Output # 1	118s	1	*exc*	*exc*
add-shift32	69	32	Output # 31	123s	10721	*exc*	*exc*
			Output # 15	80s	5089	*exc*	*exc*
mult8-add16	32	16	Output # 15	10m	180125	*exc*	*exc*
mult16	32	64	Output # 31	533m	15246	*exc*	*exc*
			Output # 15	25h	3894	*exc*	*exc*

[1] On a DECstation 5000/200. *exc* implies that the limits were exceeded.

Table 7.1: Equivalence checking applied to combinational circuits not amenable to OBDD representation

$f = g(f_1, g_1(f_2, g_2(...g_{k-1}(f_k, f_{k+1})...)))$ and each f_i has a support that is disjoint from the others. In such a case, the optimum ordering for f is the concatenation of the optimum orderings for the f_i's. The orderings for the f_i's can either be concatenated in the order from $k+1$ to 1 or from 1 to $k+1$ depending on the circuit.

The above strategy works well for parallel multipliers also. Each reconvergence of a primary input is easily detected in a given logic-level implementation and the input can then be replicated.

7.8 Results

7.8.1 Combinational Circuit Verification

The viability of the general-BDD approach for equivalence checking of combinational circuits is illustrated by the results shown in Table 7.1 for four examples. The significance of the results lies in that a canonical representation in the form of an OBDD could not be obtained for any of the three functions and that none of the three circuits are collapsible into two levels of logic. Therefore, the general-BDD approach is the only viable approach known to us for verifying these circuits, without requiring additional information other than the given logic-level descriptions.

In Table 7.1, **#I** and **#O** correspond to the total number of inputs and outputs in the circuits, respectively. **Output Verified** corresponds to the index of the output for

which equivalence checking data is provided in the table. The CPU time (**Time**) involved in the equivalence checking is provided in the table. The CPU time includes the time for creating the general BDD as well as for smoothing the variables. The size (in terms of the number of nodes in the graph) of the general BDD corresponding to the XOR of the two functions being verified is provided under the column **BDD Size**. The equivalence checking was performed on a DECstation 5000/200. The system was configured with 120 Mb of main memory and 325 Mb of swap space.

ach32 in Table 7.1 is a single-output modified 32-bit Achilles-heel function. The following equation describes the circuit. $f = \overline{mux_0} \cdot \overline{mux_1} \cdot \overline{mux_2} \cdot \overline{mux_3} \cdot \overline{mux_4} \cdot f_0 + mux_0 \cdot \overline{mux_1} \cdot \overline{mux_2} \cdot \overline{mux_3} \cdot \overline{mux_4} \cdot f_1 + \ldots + mux_0 \cdot mux_1 \cdot mux_2 \cdot mux_3 \cdot mux_4 \cdot f_{31}$ where the f_i are defined as follows:

$$f_0 = x_0 \cdot y_0 + x_1 \cdot y_1 \cdots + x_{31} \cdot y_{31}$$

$$f_{i, 1 \leq i \leq 31} = \prod_{j=0}^{31} (x_j + y_{((j+i) \ modulo \ 31)})$$

It can be shown that the best ordering for constructing an OBDD for f would require the mux signals to appear at the root of the OBDD. The children of the subgraph made by the mux signals would then be the OBDDs for the various f_i, as shown in Figure 7.9. Different f_i require different variable orderings in order to achieve non-exponential OBDD sizes. For example, f_0 requires that all the x_j and y_j be adjacent in the ordering, while f_7 requires that all the x_j and $y_{(j+7) \ modulo \ 31}$ be adjacent. It doesn't seem that any ordering exists that results in viable OBDDs for all the f_i at the same time. [2] Since building an OBDD for f entails building OBDDs for all the f_i under the same variable ordering, it is not possible to build an OBDD for f for large n. This was verified for $n = 32$ using the BDD package in MIS-II [15], using several different orderings. The replication of inputs and the variable ordering for creating the general BDDs were both done automatically for this example. It is possible to verify this circuit in about a minute. Note that f has been chosen so that neither f nor $\overline{f}$ is collapsible to a two-level sum-of-products representation. **ach32-p** is the same as **ach32** except that the various f_i are XOR'ed instead of being multiplexed. While it was not possible to verify two instances of **ach32-p** using OBDDs, verification using general BDDs required about 2 minutes.

[2] The proof for the conjecture that any OBDD would be exponential in size with respect to the number of inputs to the f_i is under development.

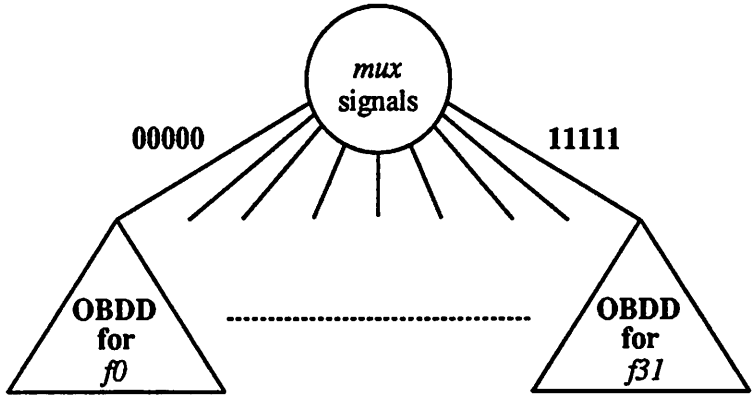


Figure 7.9: An outline of the OBDD for ach32

The second circuit in Table 7.1 is **add-shift32**. This circuit is similar in nature to **ach32** in that it basically performs one of 32 functions based on the *mux* control signals. The output of the circuit is equal to the arithmetic addition of the input A and the input B rotated by an amount given by the *mux* signals. For reasons similar to those for **ach32**, an OBDD cannot be built for this circuit. Also, this circuit is not collapsible to two-levels of logic. The automatic replication and smoothing strategies, on the other hand, are able to perform equivalence checking for the most complex output in 2 minutes. The remaining outputs all required fewer times.

mult8-add16 is a cascade of an 8-bit multiplier and a 16-bit adder, with the outputs of the multiplier feeding one set of inputs to the adder. It was not possible to build the OBDD for the most significant output of this circuit using the OBDD package in MIS-II. On the other hand, using the strategy of replicating inputs across the cone for each output of the multiplier, it was possible to make the general BDD for the most significant output of the circuit and subsequentaly check for equivalence using the smoothing techniques described in this chapter in of about 10 minutes.

The next circuits for which verification was attempted were parallel combinational multipliers. Multipliers are a class of commonly occurring circuits that have long defied attempts at equivalence checking. Multipliers are probably some of the most difficult circuits to verify because of the tremendous reconvergent fanout present (both internally and at the primary inputs) and the fact that each x input interacts non-trivially with all the y inputs and vice versa. The 12-bit multiplier is complex enough that equivalence checking cannot be performed for most of the higher order outputs using OBDDs. Using the general-BDD

approach, on the other hand, it has been possible to perform equivalence checking on all the outputs of a 12-bit multiplier in finite amounts of time. In making the general BDD for the XOR circuit for verifying the equivalence of two multipliers (*cf* Section 7.4.1), each input to each of the two n bit multiplier was replicated n times. This results in a circuit with $4n^2$ inputs, and makes smoothing difficult. It has been possible to verify the most significant bit of a 16×16 multiplier (**mult16** in Table 7.1) using the branching algorithm (*cf* Section 7.5) in about 8 CPU hours. However, it is estimated that the branching algorithm would require 25 hours of CPU time to verify the middle output (Output 15). I believe the reason for this failure is the massive initial replication of the input variables. A reasonable general-BDD representations for a multiplier wherein a smaller amount of input replication is done is under investigation.

7.8.2 Sequential Circuit Traversal

The viability of the general-BDD approach has been shown for state-space traversal in sequential circuits by means of two examples that are notable in the fact that transition relations cannot be constructed for them using OBDDs. Both the examples chosen are simple to traverse, primarily because they are arithmetic circuits. While the choice of such examples illustrates that the conventional approaches for state-space traversal are liable to fail even for examples that are simple to traverse, the choice also allowed us to illustrate the efficacy of the general-BDD approach while work on fully automating the general-BDD-based traversal algorithm is still in progress.

The first example was derived from the **add-shift32** in Section 7.8.1 by feeding back the 32 outputs to one of the inputs. The resulting sequential circuit is simple to traverse since all states are reachable in a single cycle from any initial state. That is, the transition relation becomes a tautology after the primary inputs are smoothed away. Using the general-BDD approach, it was possible to construct the general BDD for the transition relation (with the present state variables of the circuit replicated). It was then possible to smooth the primary inputs away to show that every state was reachable from every other state in a single cycle. The same conclusion could not have been reached using the techniques of [104] since the transition relation for this sequential circuit cannot be built using OBDDs.

The second example was derived from a **16-bit multiplier** by feeding the most

significant 16 outputs back to one set of inputs. If the reset state is now chosen to be the state that has the decimal value 1, then all the states will be reachable in the first cycle. In other words, the BDD representing the set of states reached after the first cycle will be a tautology. Again, this conclusion cannot be arrived at using any of the techniques *of* [104] since the transition relation for this sequential circuit cannot be built using OBDDs. While it was possible to build the transition relation for this circuit using general BDDs, it was found that smoothing the primary inputs was a problem because of the massive replication involved. Therefore, rather that build the transition relation explicitly, the technique suggested in [104] of first ANDing the transition functions for the next state lines (*cf* Figure 7.5(b)) with the general BDD for the initial state before taking the product was applied. The resulting general BDD is greatly simplified and allows us to conclude that all the states are reachable from the given reset state in the first cycle.

7.9 Conclusions

It has been shown in this chapter that a representation of logic functions, namely general Binary Decision Diagrams (BDDs), can be used in conjunction with efficient ordered Binary Decision Diagram (OBDD) manipulation algorithms, to check the satisfiability of, and verify combinational and sequential logic circuits that were not verifiable using previous techniques. In particular, it has been possible to verify large and complex arithmetic functions, for which sum-of-products or OBDD representations could not be constructed.

Future work will address improving the efficiency of the input smoothing operation on general BDDs, and the replication/ordering of inputs.

Chapter 8

Conclusions and Future Work

The focus of this dissertation has been on the investigation of logic synthesis techniques targeting optimal implementations of interacting sequential circuits. A number of aspects of the synthesis problem were addressed.

Algorithms for FSM decomposition [7, 5] were described in Chapter 3. The main point of these algorithms was to show that state partitioning can be easily tied in with recent results in state encoding [86, 41] in order to target FSM decompositions with logic-level optimality. While good results were obtained using these algorithms, there is scope for improving certain aspects. In particular, it is possible that combining factorization [40] with symbolic-output partitioning for deriving the decompositions can lead to superior results. This approach is currently being investigated.

The procedures presented in Chapter 3 and most other current sequential synthesis procedures require some form of state transition graph representation to operate from. Recognizing that, procedures for efficient state transition graph extraction from logic-level descriptions [8] were described in Chapter 4. Using these procedures, it is possible to increase the size of the sequential circuits for which current sequential logic synthesis strategies are viable.

While a large number of synthesis programs available today use either a STG representation or a two-level multiple-valued representation, it is clear that a number of synthesis operations can also be performed while operating on a BDD-based characteristic-function representation [28]. For many of these operations, it is never necessary to resort to either a STG or two-level representation of the machine, *i.e.*, it is never necessary to explicitly consider each individual state separately (sets of states are represented implicitly

by Boolean equations). It is for such operations that the BDD-based characteristic-function representation clearly subsumes the other two. The BDD-based characteristic-function representation is attractive because it is practical even for controllers that contain some arithmetic circuitry, and therefore for a much larger class of circuits. One of the important avenues for further research in sequential synthesis that can lead to a significant increase in the size of FSMs for which sequential optimization is practical is to investigate the extent to which the BDD-based characteristic-function representation can be used to address synthesis problems. At any level of abstraction, the ability to simulate at that level is the key (a necessary condition) for being able to synthesize at that level. The fact that the BDD-based characteristic-function representations are viable for traversing FSMs was presented in [104]. The application of BDD-based characteristic-function representations for detecting equivalent-state pairs was illustrated in [75]. It remains to be seen whether the BDD-based characteristic-function representation can be used effectively for FSM decomposition and state encoding.

There are two major problems with synthesizing from symbolic descriptions of sequential circuits. The first is that the cost function is dependent on the particular synthesis methodology that will be used at the combinational-level after encoding. The second, as described above, is the fact that the fanout of each state may have to be explicitly enumerated to achieve any useful optimizations. Given that the number of states is exponential in the number of latches, this fact alone is sufficient to make synthesis from symbolic descriptions impractical for large circuits. One approach is to use retiming and resynthesis to optimize the sequential circuit at the logic level [79]. Unfortunately, the techniques proposed so far have been ineffective in being able to realize substantial optimizations on controller-type sequential circuits. A possible alternative approach to sequential synthesis at the logic-level is the following. The circuit to be optimized can be considered as a set of interacting sequential circuits, partitioned so that the logic to be optimized lies in one of the subcircuits. The circuits are assumed to interact through their present states. Strong division [19, 42] can now be used to *re-encode a set of wires* that feed the subcircuit to be optimized at the same time. The decoding logic block can then be appropriately substituted into the necessary subcircuits to maintain functionality. While this is guaranteed to improve the performance of the subcircuits from which the logic was extracted, the effect of the resubstitution is not very deterministic. It may be worth pursuing this approach further.

It is desirable that a logic-level implementation of a sequential circuit be highly

testable. In case the initial implementation is not testable to the desired degree, synthesis procedures can be applied that make it highly testable. If an implementation was realized as a network of interacting sequential circuits (possibly using the algorithms presented in Chapter 3), it may be necessary to maintain the original structure for various reasons while applying the synthesis-for-test procedures. To that effect, procedures [6] were presented in Chapter 5 for synthesizing non-scan irredundant interacting sequential circuits. By virtue of these procedures, it is possible to generate don't-cares for each component submachine based on its interaction with the environment, and subsequently to optimize each submachine under its don't-care set, thereby maintaining the overall structure.

In recent work, it was shown that functional-level information at the register transfer level could be used to simplify the test generation and synthesis for test problems for sequential [51]. These algorithms will be incorporated into FLAMES in the near future. Computing sequential don't cares is just one half of the synthesis for test problem. The other half has to do with exploiting them optimally during combinational optimization. Image computation methods, originally developed for FSM verification [28], have recently been used for computing local don't-cares in multilevel combinational circuits [97]. These techniques have made it possible to use external don't-cares effectively for multilevel circuits without having to first collapse the circuit, thereby making the exploitation of sequential don't-cares practical. Chips for which pure non-scan testing is impractical, the partial scan approach where selected latches are made scannable, should be investigated.

The increasing density of transistors on chips has limited the effectiveness of the simple-minded single stuck-at fault model. An alternative approach is to use the multiple stuck-at fault model. The main problem with the multiple stuck-at fault model is the sheer number of fault combinations that must be considered. Synthesis procedures [4] were presented in Chapter 6 for obtaining highly multiple-stuck-at-fault-testable sequential circuits, with the multiple-fault test set generated as a by-product of the synthesis procedure. While the synthesis procedure works for arbitrary FSMs, a synthesis procedure that would obviate the need for pairs of states to be distinguishable in a single clock cycle is definitely more desirable. Developing such a procedure is a topic for future research.

An alternative comprehensive fault model for digital circuits is the delay-fault model. It has been shown [33] that the path-delay-fault model subsumes the single and multiple stuck-at fault models. In recent work, practical synthesis approaches have been proposed for obtaining fully-testable robustly path-delay-fault testable combinational cir-

cuits [3, 32]. Unfortunately, there are a number of problems with extending these approaches to sequential circuits. Special techniques must be used even under standard scan because of the requirement that two vectors be applied to test each fault [27]. It is possible to augment the scan registers so that they store pairs of bits at a time, but only at the expense of an area penalty in the form of larger registers. Delay-test generation for sequential circuits without making the registers scannable is obviously an even more difficult problem since a sequence of inputs that contains a pair of vectors which excites the delay fault has to be applied. The remaining portion of the sequence then has to propagate the fault effect to the primary outputs. With increasing transistor density, the path-delay-fault model is bound to become increasingly necessary. Test generation and synthesis for test for delay faults for sequential circuits is therefore a worthwhile research topic.

Logic verification is key if the complex transformations involved in sequential synthesis are to be validated. It was shown in Chapter 7 that a compact representation of logic functions, namely the general Binary Decision Diagram, can be used in conjunction with efficient ordered Binary Decision Diagram (OBDD) manipulation algorithms to check for satisfiability, and to verify larger combinational and sequential logic circuits than previously possible [9]. In particular, it has been possible to verify large and complex arithmetic functions for which sum-of-products or OBDD representations could not be constructed. Also, using this approach it has been possible to traverse FSMs for which the conventional OBDD-based characteristic-function representation could not be obtained.

A synthesis system called FLAMES has been developed which incorporates some of the algorithms described in this dissertation, some algorithms described in the dissertation of Abhijit Ghosh [48], and other algorithms commonly used in sequential synthesis. From the start, FLAMES was designed as a sequential synthesis system that would be able to manipulate networks of interacting sequential circuits. The organization and the main features of FLAMES are described in Appendix A.

Appendix A

FLAMES

A.1 Introduction

FLAMES is a program intended to provide a framework for experimenting with algorithms for manipulating interacting synchronous sequential circuits. It represents an integration of various algorithms for sequential synthesis and test developed by Srinivas Devadas, Abhijit Ghosh, Tony Ma and myself since 1987. The MIS-II system for combinational logic synthesis developed in the CAD group at Berkeley provided a good starting point for FLAMES. There were many reasons why starting with MIS-II as the core was attractive. In many ways, MIS-II represents a well engineered, modular piece of software designed with the goal of making the incremental addition of new packages easy. Additionally, given that MIS-II is a system for combinational logic synthesis, it already contains all the basic data-structures and the core algorithms necessary for manipulating multi-level combinational circuits, making the addition of sequential synthesis algorithms that much easier. Details of the organization of MIS-II can be found in Richard Rudell's doctoral dissertation [92]. A parallel effort in the development of a sequential synthesis system at Berkeley is the SIS (Sequential Interactive Synthesis) system [100]. Some of the initial work done in terms of deciding the organization of SIS was instrumental in making the task of developing FLAMES considerably easier.

A.2 Organization of FLAMES

The modularity of MIS-II manifests itself in the form of the relative mutual independence of the core packages and the clean functional interface to each of them. It has

Package	Author	Lines	Description
assign	P. Ashar, S. Devadas	1700	State encoding
cube_enum	A. Ghosh, P. Ashar	2440	ISTG extraction, minimization
dbfs	A. Ghosh	2890	BDD-based implicit FSM traversal
fdecom	S. Devadas, P. Ashar	2005	Subroutine extraction in FSMs
gbdd	P. Ashar	1500	General-BDD manipulation
hdecom	P. Ashar, J. Kukula	1785	FSM decomposition by GPIs
latch	E. M. Sentovich, K. J. Singh	135	Latch data structure
steed	A. Ghosh	2000	Sequential test generation
subnetwork	P. Ashar	790	Interacting FSM framework

Table A.1: Organization of FLAMES

been attempted to maintain the same philosophy for each of the packages added in FLAMES. The various packages added in FLAMES with their authors and size are shown in Table A.1. The size of a package is the approximate number of lines of code (computed by counting the number of semi-colons in the `.c` files) in the package. The organization of FLAMES is described in this section.

FLAMES is written in the C language on top of MIS-II. The basic data structure in MIS-II for representing a circuit is the **network** structure. In the typical situation, all the attributes of a circuit can be accessed given a pointer to the **network** structure for that circuit. Gates in the circuit are represented by the **node** data structure, and the set of nodes in a circuit is maintained as a linked list attached to the **network** structure. The basic data structures for representing combinational logic circuits and the source code for maintaining them are in the **network**, **node** and **espresso** packages. The reading and writing of Boolean networks is handled by the `io` package. BLIF (Berkeley Logic Interchange Format), Berkeley PLA and eqntott equation format are the supported I/O formats. The `command`, `io`, `main` and **network** packages in MIS-II had to be modified in FLAMES.

Sequential behavior was incorporated in FLAMES by associating latches with the network inputs corresponding to present state lines and network outputs corresponding to next state lines. This was done by the addition of the `latch` package that was developed for SIS. The latch input and latch output (which are both **node** structures), and the initial and current states of a latch are stored as attributes in the `latch` structure. Pointers to `latch` structures in a network are stored in a linked list attached to the **network** structure. All the latch input and the corresponding `latch` pointer pairs, and the latch output and corresponding `latch` pointer pairs are stored in a hash table attached to the **network**

structure.

Many sequential synthesis algorithms require a State Transition Table (STT) or State Transition Graph (STG) representation. It was decided to use an ESPRESSO-type two-level cover as the symbolic representation of the sequential circuit. The two-level cover and other attributes associated with the STG, including the start-state information and the code for each symbolic state, are stored in a structure attached to **network**. The maintenance the two-level STT representation for the network is done by the **cube_enum** package.

The `io` package had to be modified in order to read in and write out sequential circuits. STT representations of FSMs in the Berkeley KISS format can be read in and written out. For logic-level representations, the same modified BLIF representation as used for SIS was utilized. The presence of a latch in the circuit is indicated by the `.latch` keyword followed in sequence by the name of the input to the latch, the output to the latch and the initial state of the latch.

The major thrust of FLAMES was to allow sequential circuit partitioning and the manipulation of networks of FSMs. The next step in FLAMES was therefore, the incorporation of the framework which would allow that. This was done by modifying the command line interpreter in MIS-II so that multiple networks could be handled. A record of all the networks currently in a MIS-II session is maintained by means of linked list attached to each network. At any time in a FLAMES session, there is a single *active* network. Unless a command in FLAMES allows the network on which it is to operate to be named, it operates on the active network by default. Each **network** has a flag attached to it which indicates whether it is active. Obviously, only one network can have its active flag turned on at any time. It is possible to switch context to the network of choice by means of the **activate_network** command in which the network of choice is specified by its name. The names of all the networks in the current FLAMES session can be obtained using the **print_network_names** command. The **-a** option in the **print_network_names** command gives the name of the current active network. The source code for this framework is in the **subnetwork** package; whenever networks are transformed, created, or removed, the record keeping is done by the **subnetwork** package. Each subnetwork has an unambiguous parent network, and the parent of the root network is the null network. It is up to the user to interpret the hierarchy of subcircuits. Recursive partitioning is supported, and whenever a network is partitioned, the subcircuits generated are considered to be the children of the original circuit. Whenever a network is freed, all its children are also freed. The available commands for manipulating

sets of networks are `merge_networks`, `copy_network`, `free_network`, `subtract_network` and `extract_interface`. `merge_networks` combines the named networks into a single network, `subtract_network` creates a new network by removing a subnetwork from another. `extract_interface` extracts the portion of the subnetwork (corresponding to the logic feeding the next state lines) that is used by (communicates with) other named networks. This is useful for maintaining the overall functionality of the interacting FSMs when the encoding of a single FSM is changed independently of the other FSMs.

Networks of FSMs can be obtained in a number of ways. Multiple networks can be read in in the same FLAMES session. The parent of a network being read in is specified by the `.parent` keyword in the input file. If the parent is not specified, the root network is the default parent. Once read in, a circuit can be partitioned using one of a number of partitioning commands. One way is by extracting the sequential/combinational functional cone of the output/next state lines desired in each subcircuit. This facility is useful for extracting out the logic corresponding to, for example, outputs that lie on the critical path or outputs whose logic is difficult to test. This technique is also useful for automatically finding cascade/parallel decompositions corresponding to the current encoding. Partitioning can also be performed using the FSM decomposition algorithms implemented in the `hdecom` package.

A number of sequential circuit manipulation algorithms have been incorporated into FLAMES. As mentioned, algorithms for FSM decomposition described in Chapter 3 have been implemented in the `hdecom` package. FSM decomposition algorithms based on subroutine extraction [40] have been implemented in the `fdecom` package. A state minimization program based on cube-based FSM enumeration has been included in the `cube_enum` package. Algorithms for FSM traversal and verification, including implicit breadth-first traversal algorithms [104] and depth-first/breadth-first algorithms [49] have been implemented in the `dbfs` package. State encoding algorithms including a MUSTANG-type algorithm, a KISS-type algorithm and a decomposition-based state encoding algorithm have been implemented in the `assign` package. Test generation and redundancy removal algorithms for sequential circuits [50] have been implemented in the `steed` package. The general BDD generation and manipulation algorithms described in Chapter 7 have been implemented in the `gbdd` package.

A.3 A Synthesis Strategy in FLAMES

In the typical synthesis/re-synthesis scenario, a sequential circuit or a set of interacting sequential circuits is read into FLAMES for optimization. The logic that needs to be optimized is identified by first identifying, for example, the critical paths or the difficult to test faults in the circuit. The sequential circuit is then partitioned accordingly and the symbolic representation for the subcircuits to be optimized is extracted. The various sequential and combinational optimization strategies are then applied to achieve the desired specifications. Finally, the optimized circuit is verified against the original circuit for equivalence.

Bibliography

[1] V. Agrawal and K. Cheng. A complete solution to the partial scan problem. In *Proceedings of the International Test Conference*, pages 44–51, September 1987.

[2] D. B. Armstrong. A programmed algorithm for assigning internal codes to sequential machines. In *IRE Transactions on Electron Computers*, volume EC-11, pages 466–472, August 1962.

[3] P. Ashar, S. Devadas, and K. Keutzer. Testability properties of multilevel logic networks derived from binary decision diagrams. In *Proceedings of the Santa Cruz Conference on Advanced Research in VLSI*, pages 35–54, March 1991.

[4] P. Ashar, S. Devadas, and A. R. Newton. Multiple fault testable sequential circuits. In *Proceedings of the International Symposium on Circuits and Systems*, pages 3118–3121, May 1990.

[5] P. Ashar, S. Devadas, and A. R. Newton. A unified approach to the decomposition and re-decomposition of sequential machines. In *Proceedings of the 27th Design Automation Conference*, pages 601–606, June 1990.

[6] P. Ashar, S. Devadas, and A. R. Newton. Irredundant interacting sequential machines via optimal logic synthesis. In *IEEE Transactions on CAD*, volume 10, pages 311–325, March 1991.

[7] P. Ashar, S. Devadas, and A. R. Newton. Optimum and heuristic algorithms for an approach to finite state machine decomposition. In *IEEE Transactions on CAD*, volume 10, pages 296–310, March 1991.

[8] P. Ashar, A. Ghosh, S. Devadas, and A. R. Newton. Implicit State Transition Graphs: Applications to logic synthesis and test. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 84–87, November 1990.

[9] P. Ashar, A. Ghosh, S. Devadas, and A. R. Newton. Combinational and sequential logic verification using general binary decision diagrams. In *International Workshop on Logic Synthesis*, May 1991.

[10] K. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. Multi-level logic minimization using implicit don't cares. In *IEEE Transactions on CAD*, volume 7, pages 723–740, June 1988.

[11] C. L. Berman. Ordered Binary Decision Diagrams and Circuit Structure. In *Proceedings of the International Conference on Computer Desgin*, pages 392–395, October 1989.

[12] D. Bostick, G. Hachtel, R. Jacoby, M. Lightner, P. Moceyunas, C. Morrison, and D. Ravenscroft. The Boulder optimal logic design system. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 62–65, November 1987.

[13] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th Design Automation Conference*, pages 40–45, June 1990.

[14] D. Brand. Redundancy and don't cares in logic synthesis. In *IEEE Transactions on Computers*, volume C-32, pages 947–952, October 1983.

[15] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A multiple-level logic optimization system. In *IEEE Transactions on CAD*, volume 6, pages 1062–1081, November 1987.

[16] R. Brayton and F. Somenzi. Boolean relations. In *International Workshop on Logic Synthesis*, May 1989.

[17] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, Boston, Massachusetts, 1984.

[18] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[19] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. Multilevel logic synthesis. In *Proceedings of the IEEE*, volume 78, pages 264–300, February 1990.

[20] R. K. Brayton and C. McMullen. The decomposition and facorization of Boolean expressions. In *Proceedings of the International Symposium on Circuits and Systems*, pages 49–54, Rome, May 1982.

[21] R. K. Brayton and F. Somenzi. Boolean relations and the incomplete specification of logic networks. In *IFIP International Conference on Very Large Scale Integration*, pages 231–240, August 1989.

[22] M. A. Breuer and A. D. Friedman. *Diagnosis and Reliable Design of Digital Systems.* Computer Science Press, 1976.

[23] R. Bryant. Graph-based algorithms for Boolean function manipulation. In *IEEE Transactions on Computers*, volume C-35, pages 677–691, August 1986.

[24] J. Burch. Using bdds to verify multipliers. In *Proceedings of 1991 International Workshop on Formal Methods in VLSI Design*, January 1991.

[25] J. Burch, E. Clarke, K. McMillan, and D. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th Design Automation Conference*, pages 46–51, June 1990.

[26] A. Casotto, editor. *Oct Tools Distribution 5.0.* Electronics Research Laboratory, University of California, Berkeley, March 1991.

[27] K-T. Cheng, S. Devadas, and K. Keutzer. Robust delay-fault test generation and synthesis for testability under a standard scan design methodology. In *Proceedings of the 28th Design Automation Conference*, June 1991. 80-86.

[28] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using functional Boolean vectors. In *IFIP Conference*, November 1989.

[29] J. Darringer, D. Brand, J. Gerbi, W. Joyner, and L. Trevillyan. LSS: A system for production logic synthesis. IBM *J. Res. Develop.*, 28(5):537–545, September 1984.

[30] S. Devadas. Approaches to multi-level sequential logic synthesis. In *Proceedings of the 26th Design Automation Conference*, pages 270–276, June 1989.

[31] S. Devadas and K. Keutzer. Boolean minimization and algebraic factorization procedures for fully testable sequential machines. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 208–211, November 1989.

[32] S. Devadas and K. Keutzer. Design of integrated circuits fully testable for delay faults and multifaults. In *Proceedings of the International Test Conference*, pages 284–293, October 1990.

[33] S. Devadas and K. Keutzer. Necessary and sufficient conditions for robust delay fault testability. In *Proceedings of the Sixth M.I.T. Conference on Advanced Research in VLSI*, pages 221–238, April 1990.

[34] S. Devadas, K. Keutzer, and S. Malik. A synthesis-based approach to test generation and compaction for multifaults. In *Proceedings of the 28th Design Automation Conference*, pages 359–365, June 1991.

[35] S. Devadas, H-K. T. Ma, and A. R. Newton. On the verification of sequential machines at differing levels of abstraction. In *IEEE Transactions on CAD*, volume 7, pages 713–722, June 1988.

[36] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. MUSTANG: State assignment of finite state machines targeting multi-level logic implementations. In *IEEE Transactions on CAD*, volume 7, pages 1290–1300, December 1988.

[37] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. Optimal logic synthesis and testability: Two faces of the same coin. In *Proceedings of the International Test Conference*, pages 3–13, September 1988.

[38] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. A synthesis and optimization procedure for fully and easily testable sequential machines. In *IEEE Transactions on CAD*, volume 8, pages 1100–1107, October 1989.

[39] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. Irredundant sequential machines via optimal logic synthesis. In *IEEE Transactions on CAD*, volume 9, pages 8–18, January 1990.

[40] S. Devadas and A. R. Newton. Decomposition and factorization of sequential finite state machines. In *IEEE Transactions on CAD*, volume 8, pages 1206–1217, November 1989.

[41] S. Devadas and A. R. Newton. Exact algorithms for output encoding, state assignment and four-level Boolean minimization. In *Proceedings of the Twenty Third Hawaii International Conference on the System Sciences*, volume I, pages 387–396, January 1990.

[42] S. Devadas, A. R. Wang, A. R. Newton, and A. Sangiovanni-Vincentelli. Boolean decomposition in multi-level logic optimization. In *Journal of Solid State Circuits*, pages 399–408, April 1989.

[43] T. A. Dolotta and E. J. McCluskey. The coding of internal states of sequential machines. In *IEEE Transactions on Electronic Computers*, volume EC-13, pages 549–562, October 1964.

[44] E. B. Eichelberger and T. W. Williams. A logic design structure for LSI testability. In *Proceedings of the 14th Design Automation Conference*, pages 462–468, June 1977.

[45] S. Friedman. *Data Structures for Formal Verification of Circuit Designs.* PhD thesis, Princeton University, January 1990. Technical Report CS-TR-236-90.

[46] M. Fujita, H. Fujisawa, and N. Kawato. 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, November 1988.

[47] M.R. Garey and D.S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W.H. Freeman, New York, second edition, 1979.

[48] A. Ghosh. *Techniques for testing and verification of sequential circuits.* PhD thesis, University of California, Berkeley, 1991.

[49] A. Ghosh and S. Devadas. A Mixed Depth-First/Breadth-First Traversal Technique for Sequential Logic Verification. In *International Workshop on Logic Synthesis*, May 1991.

[50] A. Ghosh, S. Devadas, and A. R. Newton. Test generation for highly sequential circuits. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 362–365, November 1989.

[51] A. Ghosh, S. Devadas, and A. R. Newton. Sequential test generation at the register transfer and logic levels. In *Proceedings of the 27th Design Automation Conference*, pages 580–586, June 1990.

[52] A. Ghosh, S. Devadas, and A. R. Newton. Verification of interacting sequential circuits. In *Proceedings of the 27th Design Automation Conference*, pages 213–219, June 1990.

[53] S. Ginsburg. A synthesis technique for minimal state sequential machines. In *IRE Transactions on Electronic Computers*, volume EC-8, pages 13–24, March 1959.

[54] P. Goel. An implicit enumeration algorithm to generate tests for combinational logic circuits. In *IEEE Transactions on Computers*, volume C-30, pages 215–222, March 1981.

[55] D. Gregory, K. Bartlett, A. de Geus, and G. Hachtel. Socrates: A system for automatically synthesizing and optimizing combinational logic. In *Proceedings of the 23rd Design Automation Conference*, pages 79–85, June 1986.

[56] G. D. Hachtel, R. M. Jacoby, K. Keutzer, and C. R. Morrison. On the properties of algebraic transformations and the multifault testability of multilevel logic. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 422–425, November 1989.

[57] G. D. Hachtel, R. M. Jacoby, K. Keutzer, and C. R. Morrison. On the relationship between area optimization and multifault testabilty of multilevel logic. In *Proceedings of the International Workshop on Logic Synthesis*, June 1989.

[58] J. Hartmanis. Symbolic analysis of a decomposition of information processing. In *Inform. Control*, volume 3, pages 154–178, June 1960.

[59] J. Hartmanis. On the state assignment problem for sequential machines I. In *IRE Transactions on Electronic Computers*, pages 157–165, June 1961.

[60] J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines.* Prentice-Hall, Englewood Cliffs, N. J., 1966.

[61] F. C. Hennie. *Finite-State Models for Logical Machines.* Wiley, New York, New York, 1968.

[62] S. J. Hong, R. G. Cain, and D. L. Ostapko. MINI: A heuristic approach for logic minimization. In *IBM journal of Research and Development*, volume 18, pages 443–458, September 1974.

[63] D. A. Huffman. The synthesis of sequential switching circuits. In *J. Franklin Institute*, volume 257, no. 3, pages 161–190, 1954.

[64] D. A. Huffman. The synthesis of sequential switching circuits. In *J. Franklin Institute*, volume 257, no. 4, pages 275–303, 1954.

[65] R. M. Karp. Some techniques for the state assignment of synchronous sequential machines. In *IEEE Transactions on Electron Computers*, volume EC-13, pages 507–518, October 1964.

[66] J. Kim and M. M. Newborn. The simplification of sequential machines with input restrictions. In *IEEE Transactions on Computers*, volume C-20, pages 1440–1443, December 1972.

[67] I. Kohavi and Z. Kohavi. Detection of multiple faults in combinational logic networks. In *IEEE Transactions on Computers*, volume C-21, pages 556–568, June 1972.

[68] Z. Kohavi. Secondary state assignment for sequential machines. In *IEEE Transactions on Electron Computers*, volume EC-13, pages 193–203, June 1964.

[69] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill Book Company, New York, New York, second edition, 1978.

[70] K. Krohn and J. Rhodes. Algebraic theory of machines. In *Proceedings Symposium on Mathematical Theory of Automata.* Polytechnic Press, N.Y., 1962.

[71] L. Lavagno, S. Malik, R. Brayton, and A. Sangiovanni-Vincentelli. MIS-MV: Optimization of multi-level logic with multiple-valued inputs. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 560–563, November 1990.

[72] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing synchronous circuitry by retiming. In *Proceedings of Third CalTech Conference on VLSI*, March 1983.

[73] B. Lin. July 1989. Personal communication.

[74] B. Lin and A. R. Newton. Synthesis of multiple-level logic from symbolic high-level description languages. In *IFIP International Conference on Very Large Scale Integration*, pages 187–196, August 1989.

[75] B. Lin, H. Touati, and A. R. Newton. Don't care minimization of multi-level sequential networks. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 414–417, November 1990.

[76] R. Lisanke, editor. *FSM Benchmark Suite.* Microelectronics Center of North Carolina, Research Triangle Park, North Carolina, 1987.

[77] H-K. T. Ma. *Technologies for Logic Validation of Digital Circuits.* PhD thesis, University of California, Berkeley, 1989.

[78] H-K. T. Ma, S. Devadas, A. R. Newton, and A. Sangiovanni-Vincentelli. Test generation for sequential circuits. In *IEEE Transactions on CAD*, pages 1081–1093, October 1988.

[79] S. Malik, E. M. Sentovich, R. Brayton, and A. Sangiovanni-Vincentelli. Retiming and resynthesis: Optimizing sequential circuits using combinational techniques. In *IEEE Transactions on CAD*, volume 10, pages 74–84, January 1991.

[80] S. Malik, A. R. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. 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, November 1988.

[81] W. Maly. Realistic fault modelling for VLSI testing. In *Proceedings of the 24th Design Automation Conference*, pages 173–180, June 1987.

[82] M. P. Marcus. Derving maximal compatibles using Boolean algebra. In *IBM Journal*, pages 537–538, November 1964.

[83] E. J. McCluskey. Minimization of Boolean functions. *Bell Lab. Technical Journal*, 35, Nov. 1956.

[84] G. H. Mealy. A method of synthesizing sequential circuits. In *Bell System Tech. J.*, volume 34, pages 1045–1079, September 1955.

[85] G. De Micheli. Symbolic design of combinational and sequential logic circuits implemented by two-level macros. In *IEEE Transactions on CAD*, volume 5, pages 597–616, September 1986.

[86] G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment of finite state machines. In *IEEE Transactions on CAD*, volume 4, pages 269–285, July 1985.

[87] G. De Micheli, A. Sangiovanni-Vincentelli, and T. Villa. Computer-aided synthesis of PLA-based finite state machines. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 154–156, November 1983.

[88] E. F. Moore. Gedanken-experiments on sequential machines. In *Automata Studies*, pages 129–153. Princeton University Press, Princeton, N.J., 1956.

[89] M. C. Paull and S. H. Unger. Minimizing the number of states in incompletely specified sequential circuits. In *IRE Transactions on Electronic Computers*, volume EC-8, pages 356–357, September 1959.

[90] W. V. Quine. A way to simplify truth functions. *Am. Math. monthly*, 62, Nov. 1955.

[91] S. M. Reddy and R. Dandapani. Scan design using standard flip-flops. In *IEEE Design and Test of Computers*, volume 4, pages 52–54, February 1987.

[92] R. Rudell. *Logic Synthesis for VLSI Design.* PhD thesis, University of California, Berkeley, 1989.

[93] R. Rudell and A. Sangiovanni-Vincentelli. Exact minimization of mutiple-valued functions for PLA optimization. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 352–355, 1986.

[94] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. In *IEEE Transactions on CAD*, volume 6, pages 727–751, September 1987.

[95] A. Saldanha, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli. A framework for satisfying input and output constraints. In *Proceedings of the 28th Design Automation Conference*, pages 170–175, June 1991.

[96] G. Saucier, M. C. Depaulet, and P. Sicard. ASYL: A rule-based system for controller synthesis. In *IEEE Transactions on CAD*, volume CAD-6, pages 1088–1097, November 1987.

[97] H. Savoj, R. K. Brayton, and H. J. Touati. Use of image computation techniques in extracting local don't cares and network optimization. In *International Workshop on Logic Synthesis*, May 1991.

[98] D. Scherz and G. Metze. A new representation for faults in combinational digital circuits. In *IEEE Transactions on Computers*, August 1972.

[99] M. Schulz and E. Auth. Advanced automatic test patten generation and redundancy identification techniques. *Proceedings of the 18th International Symposium on Fault-Tolerant Computing*, June 1988.

[100] E. M. Sentovich. SIS: An interactive system for the synthesis of sequential logic circuits. 1991. Unpublished document.

[101] H. Simonis. Formal verification of multipliers. In *Proceedings of 1991 International Workshop on Formal Methods in VLSI Design*, January 1991.

[102] R. E. Stearns and J. Hartmanis. On the state assignment problem for sequential machines II. In *IRE Transactions on Electronic Computers*, pages 593–604, December 1961.

[103] A. Stolzle. A VLSI wordprocessing subsystem for a real time large vocabulary continuous speech recognition system. In *MS Thesis*, September 1989.

[104] H. Touati, H. Savoj, B. Lin R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 130–133, November 1990.

[105] J. H. Tracey. Internal state assignment for asynchronous sequential machines. In *IRE Transactions on Electronic Computers*, pages 551–560, August 1966.

[106] T. Villa and A. Sangiovanni-Vincentelli. NOVA: State assignment of finite state machines for optimal two-level logic implementations. In *IEEE Transactions on CAD*, volume 9, pages 905–924, September 1990.

[107] W. Wolf, K. Keutzer, and J. Akella. A kernel finding state assignment algorithm for multi-level logic. In *Proceedings of the 25th Design Automation Conference*, pages 433–438, June 1988.

[108] S. Yang and M. Ciesielski. On the relationship between input encoding and logic minimization. In *Proceedings of the Twenty Third Hawaii International Conference on the System Sciences*, volume I, pages 377–386, January 1990.

[109] M. Yoeli. The cascade decomposition of sequential machines. In *IRE Transactions Electronic Computers*, volume EC-10, pages 587–592, April 1961.

[110] M. Yoeli. Cascade parallel decomposition of sequential machines. In *IRE Transactions on Electronic Computers*, volume EC-12, pages 322–324, June 1963.

[111] H. P. Zeiger. *Loop-free Synthesis of Finite-State Machines.* PhD thesis, Massachusetts Institute of Technology, Cambridge, 1964.

vitunic@ic.Berkeley.EDU

JOB 937

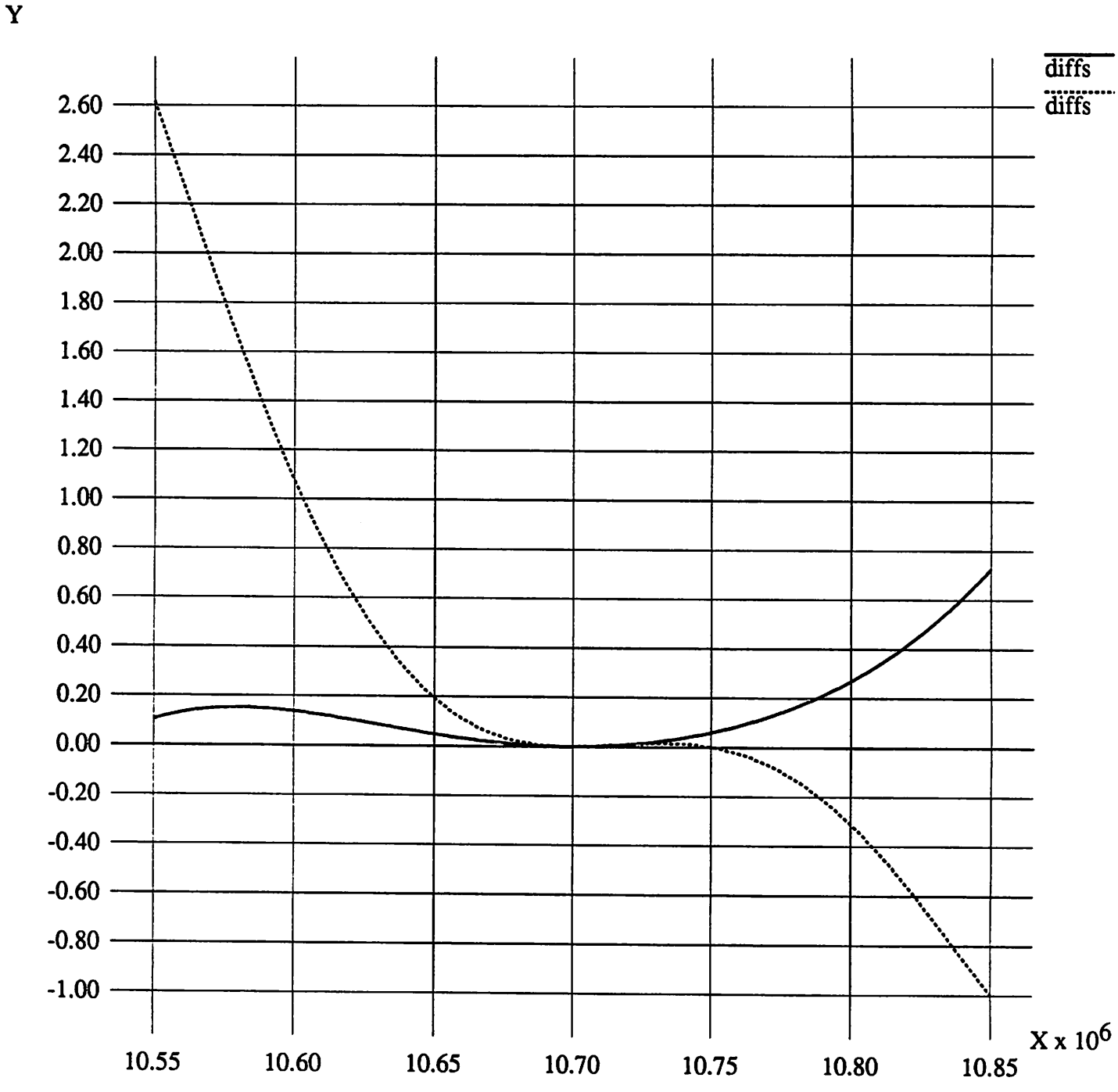
stdin

Printer queue: lps20a
Started: Wed May 27 13:31:07 1992

Digital Equipment Corporation
ULTRIX™

PrintServer 20

X Graph


Y
diffs
diffs
2.60
2.40
2.20
2.00
1.80
1.60
1.40
1.20
1.00
0.80
0.60
0.40
0.20
0.00
-0.20
-0.40
-0.60
-0.80
-1.00
10.55
10.60
10.65
10.70
10.75
10.80
10.85
X x 10^6

kennish@kabuki.berkeley.edu

JOB 237

nerd_report

Printer queue: lps20a
Started: Wed May 27 13:40:22 1992

Digital Equipment Corporation
ULTRIX™

PrintServer 20

A Quantitative Measure of Differential Social Abilities: Nerd Test Results *

Linda A. Kamas
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley

Abstract

This report investigates and documents the percent nerdity *of a sample of 51 subjects from the Electrical Engineering and Computer Science graduate student community at the University of California at Berkeley.*

1 Introduction

It has been postulated that those whose chosen avocation is in an area such as Electrical Engineering (EE) and Computer Science (CS) tend to have certain differential social abilities. This differential set of social abilities is often termed "nerdity" and thus, he or she who posesses such qualities is often termed a "nerd." This term has gone in and out of style over time with both positive and negative connotations; the positive connotations being that one is particularly adept at using a computer or has other, usually technical, knowledge; the negative being that one will neglect one's hygiene, underdevelop other interpersonal skills and and de-emphasize verbal skills and thus speak in long and convoluted sentences in exchange for the time invested in developing technical knowledge.

In order to quantify these qualities, a "Nerd Test" (see Appendix A) was developed by engineers at MIT, a reputed nerd Mecca. This report documents the results of applying this test to those in the Berkeley EECS graduate community and the reactions of some respondents.

2 Experimental Procedures

The Nerd Test was distributed through the "Happy Hour" mailing list, a list of over 100 people in the Berkeley EECS community who are at least interested in knowing about a social activity at least once a week. Given the description of subjects, this is admittedly a non-random sampling of the department and could result in a skewed data set. However, since there is no similar previous data, this simple procedure will at least give some insight. (And besides, sociologists do this *all* the time, except they only survey their friends and sometimes they don't even keep data. They then write a report based on their *feelings* about the results. Often, a major USA newspaper hears about the results and it gets printed.)

The interpretation of questions could result in a tolerance of about plus or minus 3 points. In particular, it was not clear whether to count the Star Trek movies as episodes. Also, some respondents were not sure exactly what a rhetorical question was.

Results were received by electronic mail and in person over a period of approximately three weeks starting on April 2, 1992.

It is not known how many respondents printed

*This work has been supported by cooperation of the "Happy Hour" group.

out the test and tabulated by paper, but at least two respondents were documented to have taken and tabulated their scores by computer. One, whom we will call by the fictitious name "Henryc" to protect his identity, responded as follows.

> Results from nerd test taken, 19 July 1991.
> After answering "yes" or "no" to each question.
> % grep "^ yes" nerd | wc -w
> 48
> And I have confirmed that grepping for "no" gives me 52.

One respondent, whom we will fictitiously call "Kennish" made the following guidelines:

> The central committee on tests and social graces has determined that the minimum acceptable score for an EE major is 14, and that of a respectable EE major is 25. Furthermore, the committee requests that anyone scoring over 90 seek help.

3 Experimental Results

With the data obtained from 51 respondents, (see Appendix B), we have the following statistical results:

> average: 47.69
> median: 49
> standard deviation: 9.98
> range: 32 to 72.

However, those tested in a social setting (party) provided by one of the Happy Hour founders had the following statistical results:

> average: 43.82
> median: 40
> standard deviation: 9.81
> range: 32 to 71

compared to those not polled in a social setting:

> average: 50.89
> median: 51
> standard deviation: 9.11
> range: 32 to 72,

with the standard definitions of average, median, and standard deviation [1]. One respondent in the social setting, who was thought to have received an excessively low score, 25, was audited and was found to have had an actual score of 33. This implies that perhaps the difference in statistical data from those in a social setting could be attributed to environmental social pressures causing one to answer "no" to borderline questions more frequently than respondents being tested elsewhere. It is also possible that the group of respondents who would actually engage in social activity such as a party would have lower nerd scores. Also, respondents exhibited a certain amount of denial about their nerdiness, which could affect scores from those tested in all settings.

As a reference, the average of 21 respondents at the Carnegie-Mellon University graduate psychology department was 35 with a range from 20 to 51. However, the CMU Nerd Test distributor comments:

> We're having great fun assessing the nerdiness of the psych. dept. However, we've concluded that this test only taps the techno-geek aspects of nerdiness.

There were 6 respondents who were not in the CMU group and not in the EECS community. Their average score was 23.33

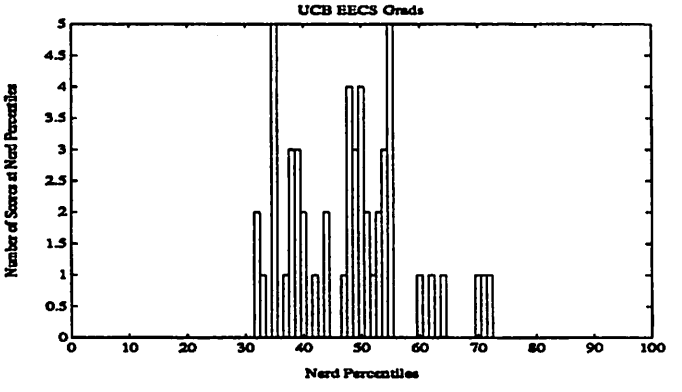


Figure 1: All EECS Grads Tested

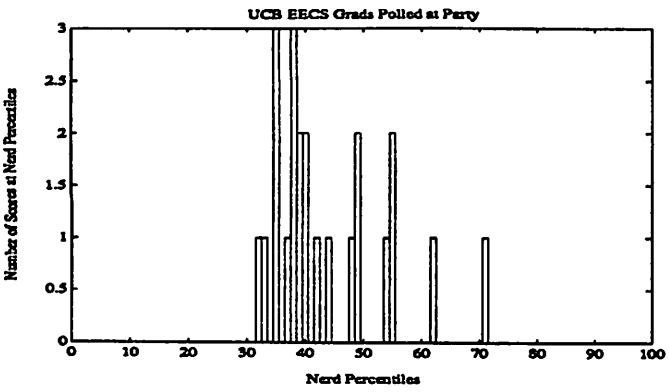


Figure 2: EECS Grads Tested in a Social Setting

4 Conclusions

The 51 subjects from the Electrical Engineering and Computer Science graduate community at UC Berkeley had an insignificantly higher nerd score average (47.69) than the graduate psychology department at Carnegie-Mellon University (35) and a slightly higher average than the others tested that were not in the EECS community (23.33). This implies that nerd-like tendencies are only slightly more prevalent in the EECS department. However, other factors, such as denial about nerdiness could have caused respondents to report an artificially lower score.

A The Nerd Test

Score one point for each YES. The score is % nerdity.
1) Have you ever used a computer? If the answer is no, try taking the Baker House Purity Test.
2) Have you ever programmed a computer?
3) Have you ever built a computer?
4) Done #2 continuously for more than four hours?
5) Done #2 continuously for more than eight hours?
6) ? (For those non-MIT students out there, this translates as, "Do/did you major in electrical engineering or computer science?")
7) Do you wear glasses?
8) Are your glasses broken (e.g. taped)?
9) Is your vision worse than 20/40?
10) Worse than 20/80?
11) Are you legally blind?
12) Have you ever asked a question in lecture?
13) Have you ever answered a question in lecture?
14) Have you ever corrected a professor?
15) Have you ever answered a rhetorical question?
16) Do you sit in the front row?
17) Do you take notes in more than one color?
18) Have you ever worn a calculator?
19) Do you read science fiction?
20) Have you ever used a microscope?
21) Have you ever used a telescope?
22) Have you ever used an oscilloscope?
23) Is your weight less than your IQ?
24) Have you ever done #2 on Friday, Saturday, and Sunday of the same weekend?
25) Have you ever done #2 past 4 am?
26) Have you ever done #2 with someone of the appropriate (either or both, your choice) sex (besides your consultant)?
27) Have you ever done #2 for money?
28) Do you have a Rubik's Cube?
29) Can you solve it?
30) Without the book?
31) Without looking?
32) Do you have acne?
33) Do you have greasy hair?
34) Are you unaware of it?
35) Have you ever bought anything from Radio Shack?
36) From Heathkit?
37) Do you know trigonometry?
38) Do you know calculus?
39) Do you know Maxwell's Equations?
40) Do you have them on a t-shirt?
41) Have you ever dissected anything?
42) Do you know pi past five decimal places?
43) Do you know e past five decimal places?
44) Do you own more than $ 500 in electronics (excluding stereo)?
45) More than $ 1000?
46) More than $ 2500?
47) Have you ever built more than $ 2500 worth of electronics?
48) Have you ever watched Dr. Who?
49) More than three times in the same night?

50) Have you ever read *The Hitchhiker's Guide to the Galaxy*?
51) Was your SAT math score more than 300 points higher than your verbal?
52) Have you ever done homework on a Friday night?
53) Have you ever pulled an all-nighter?
54) Have you ever redesigned a major household appliance?
55) Have you ever played a computer game?
56) Done #55 in the last three months?
57) Done #55 in the last three weeks?
58) Have you ever written a computer game?
59) Are your pants too short?
60) Do your socks mismatch?
61) Have you used a chemistry set?
62) After the age of 13?
63) Have you ever played D & D (or any other role-playing game)?
64) Since high school?
65) Have you ever entered a science fair?
66) Did you win?
67) Do you own a digital watch?
68) Does it play music?
69) Does it have a calculator?
70) Have you ever used a rare earth element?
71) Do you own a CRC?
72) Do you own a CRT?
73) Do you know RPN?
74) Do you own a laser (over 1 mw)?
75) Were you ever on a chess team?
76) A debate team?
77) Do you know more than three programming languages?
78) More than eight?
79) Have you ever made a technical joke?
80) Did no one get it?
81) Can you name more than ten Star Trek episodes?
82) Are you socially inept?
83) Do you own a pencil case?
84) Do you wear it?
85) Do you know Schrodinger's Equation?
86) Have you ever solved it?
87) Have you ever used the word "asymptotic"?
88) Can you count in binary?
89) Have you ever broken into a computer system?
90) A government system?
91) Have you ever changed your bank account?
92) Changed someone else's?
93) Done #92 for money?
94) Have you ever inhaled helium?
95) Do you know the Latin name for the fruit fly?
96) Do you own anything that is radio controlled?
97) Have you ever interpolated?
98) Have you ever extrapolated?
99) Have you ever used a modem?
100) Have you ever reached sexual climax while doing #2?

B Raw Data

Percent Nerdity of EECS grads:

Malik Audeh	39#
Steve Burgett	50
Tim Callahan	39#
Ben Bonham	44#
Bart Bombay	35#
Lisa Buckman	33# *
Ed Chang	50
Henry Chang	55**
Phyllis Chang	38#
Lennard Chen	42#
Cormac Conroy	48#
Mike Daumer	35#
Allen Downey	40#
Eric Felt	50
John Gamelin	38#
Heather Harkness	39
Soren Hein	48
Kevin Heppell	55#
Timothy Hu	72
Anna Ison	32#
Gary Jones	48
Gani Jusuf	60
Alan Kamas	55
Linda Kamas	54
Peter Kennedy	49#
John Kohl	62#
John Krick	35#
Josh Lack	38#

Sherry Lee	35
Chris Lennard	53***
Ethan Miller	54
Danny Miranda	49
Robert Neff	48
Ken Nishimura	52
Mark Noworolowski	53
Fred Obermeier	70
Bruce Parnas	37# ****
Ruth Rosenholtz	54#
Steven Scole	51
Ben Sheng	43
Henry Sheng	51
Costas Spanos	50
Susan Streisand	44
Kiran Thakare	71#
Dawn Tilbury	35
Greg Uehara	32
Greg Walsh	55
Andrew Wang	64
Todd Weigandt	40 #
Denise Wolf	47
Wayne Wonchaba	55 #
Others:	
Peter Beerel	44 (Stanford Grad)
Greta Gize	22 (B.A. Psych)
Betty Kamas	21 (B.A. Psych)++
Chuck Kamas	61 (BSEE)
CMU Psych Dept.	35 (the average of 21 respondents)
Eleen Kamas	36 (B.A. Psych, B.A. Math)++
Rafi Laufer	25 (Math Grad)
Lisa Nathanson	15 (B.A. Psych)
Hieu Nguyen	37 (Math Grad, but did B.S. in EE)
Dave Ofelt	49 (Stanford Grad)
Pora Park	20 (B.A. Honors, Rhetoric)++
Eric Taylor	36 (BSEE)

#: Polled at party

*: Received 25 before getting audited.

**: "To my horror, I scored 7 points more than I did just 9 months ago."

***: "He puts his pencils in a tennis ball can case, does that count as a pencil case? I did see him wearing it in his pocket today. And I did ask him, "Is that a tennis ball can case in your pocket, or are you just..."

****: Answered "yes" to question 100.

++: Has immediate family or is married to an engineer.

References

[1] C. W. Helstrom, *Probability and Stochastic Processes for Engineers.* MacMillan Publishing Co., 1984.