Copyright © 1994, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

LOGIC OPTIMIZATION OF INTERACTING COMPONENTS IN SYNCRHONOUS DIGITAL SYSTEMS

by

Yosinori Watanabe

Memorandum No. UCB/ERL M94/32

29 April 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

Logic Optimization of Interacting Components in Synchronous Digital Systems

by

Yosinori Watanabe

Doctor of Philosophy in
Electrical Engineering and Computer Sciences
University of California at Berkeley
Professor Robert K. Brayton, Chair

In optimizing digital systems, manual designs sometimes use information derived from other components to identify a functional flexibility at a particular component. This thesis addresses how to identify such a flexibility as well as how to use it in the optimization of synchronous digital systems.

We first focus on the case where the system realizes a combinational logic behavior, and propose a procedure for computing a set of permissible functions at each component, i.e. the set of functions that can be realized there while preserving the behavior of the entire system. The identified set of functions is represented by a single relation between the inputs and the outputs of the component. We then address the problem of finding an optimum permissible function. The problem is reduced to the minimization of relations, and we develop a heuristic procedure for the problem.

The second half of the thesis performs an analogous investigation for sequential logic behaviors. We consider a synchronous system in which the behavior of each component is modeled by a finite state machine, and show that the complete set of its permissible sequential behaviors can be represented by a single non-deterministic finite state machine, which we call the E-machine. We give a fixed point computation for deriving the E-machine. We then consider the problem of finding an optimum sequential behavior, which is achieved by minimizing the E-machine. We show that the E-machine is a special type of non-deterministic finite state machine, and use this property effectively in the minimization. We develop both exact and heuristic procedures for the problem.

R. K. Brayton
Professor Robert K. Brayton
Thesis Committee Chairman

Contents

List of Figures

List of Tables

Acknowledgments

Many people supported me in various ways during the time I spent at Berkeley. Without the help and encouragement, my will to graduate might have faded away in an early stage of the Ph.D. program, and this thesis would not exist. I sincerely express gratitude to all of them, especially to the following people.

My advisor, Robert K. Brayton. I owe to Bob absolutely everything about research. Professor Brayton provided me all the technical skills that I currently have; how to set up the research goal, how to formulate and attack a problem, how to write a paper, and how to organize a technical presentation. Among his outstanding abilities, I was especially impressed with his attitude of conducting research with a full of respect toward the truth. With this attitude, Bob demonstrated that research is an activity of proceeding on a trail of truth, rather than marching down into a desired direction. I learned that this attitude is the origin of courage, indispensable when stuck at an unexpected obstacle, to further proceed on the trail even though it leads to an unpleasant outcome. It was also this attitude that convinced me of the significance of mathematical rigor and precision in order to follow the right path. I admired Professor Brayton so much that I simply mimicked his style. It was always fun when I felt that my technical abilities improved, and most pleasant when I realized that once a research has been accomplished, it is often the case that the result can be stated in a simple statement. This has been a useful check to see whether I have fully reached the end of the trail, and also a great guidance to deliver a clear presentation. I will treasure what I learned from Professor Brayton for the rest of my life.

I am grateful to Alberto L. Sangiovanni-Vincentelli and Shmuel Oren, the second and the third members of my thesis committee. Professor Sangiovanni provided fruitful comments on the research, especially for the second half of this thesis. Professor Oren kindly agreed to be a member of both the qualification and the thesis committees, even though his research interests are not directly related to the focus of the thesis. I also wish to thank Professor Katherine A. Yelick for

her agreeing to be a committee member of my qualification examination.

Ernest Kuh has been thoughtful. As a family friend, Professor and Mrs. Kuh occasionally invited me to dinner, even for a private gathering, and extended generous hospitality. He congratulated me whenever I made an achievement, like on my passing the qualification examination, or on my first presentation at an international conference. Professor Kuh sometimes dropped in a seminar room while I was giving a talk, and gave positive feedback. I gratefully acknowledge his continuous support and encouragement over the years.

I enjoyed fruitful collaboration in conducting this research. The clustering procedure, presented in Section 2.5, is the result of collaboration with Lisa Guerra. Lisa also implemented a prototype of the procedures described in the chapter. I sincerely appreciate her effort. I also thank Alexander Saldanha for his interests in the work given in Chapter 4. Alex carefully read my notes, and immediately provided a different interpretation of the result. This work yielded the construction of non-deterministic E-machines, described in Section 4.6.

Significant cooperation was given by several people in order to complete the experiments presented in this thesis. I thank Abhijit Ghosh for providing the program of his procedure for minimizing a Boolean relation. I am indebted to Fabio Somenzi for his support on the research presented in Chapter 3, the minimization of relations. Whenever I sent an e-mail to ask a question about his exact method of minimizing Boolean relations, he provided a perfect answer promptly, usually in 30 minutes. Fabio also allowed me to use the program of his procedure for experiments. Timothy Kam and Tiziano Villa let me use their procedure for a covering problem, with which I was able to conduct additional experiments in Chapter 5. Huey-Yih Wang kindly offered me a set of examples of interacting finite state machines for the experiments given in the same chapter. I acknowledge their support. Special thanks are to Thomas Shiple, whom I always consulted on questions about binary decision diagrams. Tom was eager to understand my problems, and often came up with excellent solutions.

It was a great pleasure to be a member of the Berkeley CAD group. I thank every member of the group, including those who used to be here. I express special gratitude to Albert Wang and Hervé Touati. I learned from Albert art of (constructive) proof techniques, and from Hervé elegance of presentation. My thanks also go to Brad Krebs and Mike Kiernan for excellent support on facilities, to Flora Oviedo, Elise Mills, and Kia Cooper for daily assistance, and to Heather Brown, Carol Lynn Stewart, and Genevieve Thiebaut for administrative support of the graduate program.

My life at Berkeley would not have been so delightful, if it had not been for the local Japanese community. I owe the precious memory and experience to those who stayed at Berkeley

as visitors or students, as well as those who often visited the Bay area. They are, in alphabetical order, Kozo Bando, Takashi Fujii, Tomoyuki Fujita, Yoshihiro Fujita, Masahiro Fukui, Hiroaki Furuichi, Naoko Furukawa, Kenji Goto, Toshihiro and Keiko Hattori, Hiroshi Ichiryu, Nagisa Ishiura, Masamichi and Yoshiko Kawarabayashi, Shigeyoshi Kawarai, Takashi and Mami Komaya, Yuji Kukimoto, Tadahiro Kuroda, Naotaka Maeda, Yusuke and Taeko Matsunaga, Hitoshi Matsuo, Takashi Mitsuhashi, Yasuhiko Nakano, Tsuneo Nakata, Yoshihito Nishizaki, Yasushi Ogawa, Hidetoshi Onodera, Akira Onozawa, Yasuaki Sakina, Masao Sato, Masatoshi Sekine, Kei and Rikako Suzuki, Masayoshi Tachibana, Atsushi Takahara, Yoshio Takamine, Toru Toyabe, Atsuhisa and Chiori Yakawa, and Makiko Yoshida. I am also grateful to Masami and Nobuko Fujimoto for their support.

Finally, I thank my parents, Hitoshi and Reiko Watanabe. They regularly called me on weekends, and it was always fun to hear from them what's new there. I also liked telling them about my recent activities, especially when I had good news, because they always got excited more than I could.

Now, most emphatically, I thank my wife, Mika. I was not able to find the right word in my English dictionary to express how much I owe to her. *Hontoni doumo arigatou*, Mika. I intend to spend many years proving to her that I am worth the effort.

Chapter 1

Introduction

1.1 Optimization of Interacting Components

A *system* is a regularly interacting or interdependent group of items forming a unified whole[36]. It is common that each item, or *component*, has some flexibility in its function. Namely, one may alter the function of the component while preserving the behavior of the entire system. Such flexibility, associated with internal components, is the subject of this thesis. Specifically, the thesis addresses two issues; how to identify the flexibility, and how to use it.

Suppose one wants to design a system so that it realizes a given behavior. The design task can be viewed as an iteration of the following two major processes. The first process is to determine or modify the structure of the system, i.e. what kind of components are used and how they interact. The second is to optimize the behaviors of the components with respect to the structure defined above. This thesis deals with the second phase; for a given set of interacting components, optimize behaviors of the components so that the entire system realizes a desired behavior. We consider the problem in two steps. First, we identify a set of behaviors that can be realized at a component. Such a set defines the flexibility associated with the component, and we call it a set of *permissible* behaviors of the component. We then find, in the second step, an optimum behavior in the set according to a given cost function.

The type of systems we consider in this thesis is *digital* systems. We follow the conventional definition of digital systems [2, 4], i.e. one in which each component can be modeled so that it handles information in a discrete manner. By behavior, we mean logic behavior. Specifically, we consider two types of behaviors. One is a behavior modeled by a Boolean function, which is also referred to as a *combinational logic behavior*. In particular, we focus on a function with multiple

outputs. Namely, each component of the system has multiple inputs and multiple outputs, and implements a Boolean function between the inputs and the outputs. The other type of behaviors is *sequential logic behaviors*. In this case, each component, with multiple inputs and outputs, implements a function between the input sequences and the output sequences.

Among digital systems, we restrict our attention only to *synchronous systems* in this thesis. In a synchronous system, the operations of all the components of the system are synchronized by a global timing controller, or a *clock*, and each operation requires an integer number of clock intervals [5, 19]. For combinational logic behaviors, we assume that signals arrive at all the inputs of the system at the same time and it takes no time for each component to compute the corresponding output signals. For sequential logic behaviors, we assume that every component generates an output event simultaneously, one at a time for each clock interval. Under these assumptions, we consider, for each type of behaviors, how to compute a set of permissible behaviors, as well as how to find an optimum behavior.

1.2 Organization of the Thesis

This thesis consists of four main chapters. One can divide them in two ways. One way is based on the types of behaviors. In Chapter 2 and Chapter 3, we deal with combinational logic behaviors, while sequential logic behaviors are considered in Chapter 4 and Chapter 5. The other way of dividing the thesis is in terms of the types of problems. In Chapter 2 and Chapter 4, we consider the problem of identifying a set of permissible behaviors, while Chapter 3 and Chapter 5 are concerned with the problem of finding the best behavior. Specifically, each chapter is organized as follows.

Chapter 2 deals with the case where each component implements a Boolean function with multiple inputs and multiple outputs. Although extensive research has been made for the problem of finding permissible behaviors for combinational logic with a single output [3, 39, 50], little has been done for the case of multiple outputs. In this chapter, we first discuss the problem of finding the maximum set of permissible behaviors for a given component. Namely, fixing the behaviors of all the other components of the system, we compute the complete set of behaviors at a particular component that can be realized there while preserving the behavior of the entire system. A behavior here is a Boolean function with multiple outputs. Brayton and Somenzi showed in [11] that the complete set can be represented by a *Boolean relation*, a relation between the Boolean spaces spanned by the inputs and the outputs of the component respectively. Namely, a

function f is in the set if and only if for all elements $\mathbf{x}$ of the input domain, the pair $(\mathbf{x}, f(\mathbf{x}))$ is a member of the relation. A procedure for computing such a set was proposed by Savoj [48]. We first review the work, and then focus on another kind of permissible behaviors called a *compatible* set of permissible behaviors [39]. Unlike the maximum set, a compatible set of permissible behaviors is computed for every component of the system, and has the property that an arbitrary combination of behaviors of the sets over the components results in a desired behavior on the entire system. The property of compatibility enables one to process all the components in parallel for finding optimum behaviors in the succeeding step. We consider the problem of computing *maximally* compatible sets of permissible behaviors over the components, where compatible sets are maximal if there is no permissible behavior that can be newly added to any one of the sets without destroying the property of compatibility. We show that each of the maximally compatible sets can be also represented by a Boolean relation, and present a procedure for computing such sets. We also show that the maximum set of permissible behaviors at a particular component can be computed as a special case of this procedure.

In Chapter 3, we address the problem of finding an optimum combinational logic behavior for multi-output components. As a cost function, we use the number of product terms required in a sum-of-products expression representing a function. Therefore, the problem is that for a given relation representing a set of functions, find a sum-of-products expression with the minimum number of product terms over all the functions in the set. This problem is called the minimization of relations, and is a generalization of the minimization of functions, which has been fully investigated [8, 25, 34]. In this chapter, we consider the general case where an input variable of a function may assume more than two values. We present a heuristic procedure for the problem, which is analogous to Espresso [8], a well-developed heuristic approach for minimizing functions. We make a contrast between relations and functions by showing some special properties associated with relations not found in functions. These properties are carefully accounted for in the proposed heuristic procedure to effectively achieve high quality results. The results are compared against the exact method given in [53].

Chapter 4 and Chapter 5 perform an analogous investigation for sequential logic behaviors. Each component of the system now implements a sequential logic behavior, i.e. a function between the input sequences and the output sequences of the component. Under the assumption of synchronous systems, the behavior can be represented by a finite state machine [35]. Therefore, the entire system can be regarded as one in which a number of finite state machines are mutually interacting.

Chapter 4 considers the problem of finding the maximum set of permissible behaviors at a given component. The definition of the set is same as the combinational case, except that a behavior here is now sequential. Although extensive research has been done for the problem [18, 29, 43, 56], no method is known on how to capture the complete degree of freedom. We claim that the complete set of permissible sequential behaviors of a given component can be computed and represented by a single non-deterministic finite state machine, which we call the E-machine. A procedure to compute the E-machine is also provided.

In Chapter 5, we *minimize* the E-machine. The problem is to find an optimum permissible behavior given in the E-machine, where we use, as a cost function, the number of states required in a finite state machine to represent the behavior. This is a generalization of the problem known as the state minimization, or the state reduction, of deterministic finite state machines, for which the research is well-matured [1, 23, 24, 27, 38, 40, 41, 44]. The chapter first presents a theoretical analysis, where we consider how the conventional concepts developed for deterministic machines can be generalized. We show that the E-machine is a special type of non-deterministic machine, and this property can be effectively used for solving the problem. We propose both exact and heuristic procedures, and conduct experiments to demonstrate the effectiveness of taking into account the interaction with other components of the system in optimizing systems of finite state machines.

In each chapter, we first clarify the focus of the chapter, and provide a background of the subject, such as related works in the literature or applications of the problem. Terminology used in the chapter is then defined so that each chapter is made self-contained. Remarks on the individual approaches presented in each chapter, such as evaluation of experimental results or technical limitations of the procedures, are noted at the end of the chapter. The entire thesis is concluded in Chapter 6.

Chapter 2

Permissible Logic Functions for Multi-Output Components

2.1 Introduction

We consider the case where each component of a system is combinational logic, i.e. it implements a Boolean function with multiple inputs and multiple outputs. We assume that no combinational loop exists in the system; in particular we assume that the connections of the components do not form a cycle[1]. Such a system can be represented by a directed acyclic graph, where a node corresponds to a component and an edge corresponds to a connection between components. A multi-output Boolean function is associated with each node, which is the one implemented by the corresponding component. We call the graph a *clustered Boolean network.*

The problem addressed is to find a set of permissible Boolean functions at each node of a given clustered Boolean network. The application we consider is the logic optimization of multi-level combinational logic circuits. One optimization technique is to optimize a logic representation of a function that can be realized at a given sub-portion of a circuit. This is generally referred to as a *local optimization* or *node optimization.* It can be achieved by (1) computing a set of functions allowed to be implemented at the node, and (2) finding one with a least-cost representation. A clustered Boolean network is a model for a multi-level combinational logic circuit, and the first step of this optimization scenario is the problem addressed in this chapter. Chapter 3 is concerned with the last step.

[1]It could happen that a cycle of components would not have a cyclic dependence, but we rule this out for ease of exposition. This can happen even if each component has only one output.

Conventionally, a multi-level combinational logic circuit is abstracted in a more restrictive way by using a model called a *Boolean network*, and the problem above has been extensively studied for this model [9]. A Boolean network is also a directed acyclic graph, where the Boolean function associated with each node must have a single output. This single-output requirement is considered a limitation, since experience with two-level (PLA) minimization demonstrates that better results are usually obtained by simultaneously optimizing a set of functions, rather than one at a time. In this chapter, we remove this restriction and see how node optimization can be done for the general case.

We consider two types of sets of permissible functions. One is the maximum set of permissible functions defined at a given node of a clustered Boolean network. Specifically, fixing the Boolean functions associated with the rest of the nodes as they are, one wants to compute the maximum set of Boolean functions that can be realized at the node while preserving the functionality of the entire network. Brayton and Somenzi showed in [11] that in case of multiple-output functions, such a set cannot be represented by a Boolean function with don't cares, and introduced a theory of Boolean relations. The claim is that the maximum set can be computed and represented by a Boolean relation, where the relation consists of all possible pairs of input and output values allowed to be realized at the node. A procedure to compute such a relation was presented by Savoj in [48]. We first review how a set of functions can be represented by a relation as well as why a relation is necessary to represent such a set. We then consider another type of set of permissible functions called *compatible* sets. For this type, a set of sets of Boolean functions is defined, one set for each node of a network, and has the property that simultaneous replacement of the functions associated with the nodes by an arbitrary combination of the functions, one from each of the respective sets, preserves the functionality of the entire network. The notion of compatible sets was first introduced by Muroga *et. al* [39] for the case where each node of the network implements a single-output NOR gate, along with a procedure to compute such sets. Savoj proposed a method for a more general case where, instead of a NOR gate, a node may implement an arbitrary single-output Boolean function [49]. Introduced also in [49] is a notion of *maximally* compatible sets of permissible functions, where compatible sets are maximal if there is no function that can be newly added to any one of the sets without destroying the property of compatibility. However, no method is known for computing such sets even for the case of single-output Boolean functions.

In this chapter, we consider the problem of computing *maximally compatible* sets of permissible functions for the case where a node may implement an arbitrary *multiple-output* Boolean function. We first show that each of such sets can be also computed and represented by a Boolean

relation. Then we present a procedure to compute such relations. Since we make no assumption on the number of outputs of a Boolean function associated with each node, the proposed method is valid even for the case of single-output functions.

This chapter is organized as follows. After defining terminology in Section 2.2, we review how to compute the maximum set of permissible functions for a clustered Boolean network in Section 2.3. Section 2.4 addresses the problem of finding maximally compatible sets of permissible functions, and presents a procedure to compute such sets for clustered Boolean networks. We also show in this section that the maximum set of permissible functions can be obtained as a special case of the proposed procedure. In Section 2.5, we present a procedure of composing a clustered Boolean network from a given Boolean network. Such a procedure is necessary when one performs optimizations based on a conventional Boolean network, and then wants to apply local optimization based on a clustered Boolean network. A clustered Boolean network is composed by grouping together a set of nodes of the original Boolean network. The procedures have been implemented and we present experimental results in Section 2.6. Section 2.7 concludes the chapter.

2.2 Preliminaries

We first define a Boolean network, the conventional model used for logic optimization of multi-level combinational logic circuits.

Definition: Boolean Network

A **Boolean network** $\eta = (N, E)$ is a directed acyclic graph. A node with no in-coming edges is called a **primary input node**. A Boolean network contains at least one node referred to as a **primary output node**. A node that is neither a primary input node nor a primary output node is called an **intermediate node**. The set of the primary input nodes, the primary output nodes, and the intermediate nodes are denoted by X, Z, and N_I, respectively. Given two nodes, s and t, s is a **fanin** of t if an edge $[s, t]$ exists in η. Conversely, t is a **fanout** of s in this case. Similarly, s is said to be a **transitive fanin** of t, if s is a fanin of t or there is a fanout of s that is a transitive fanin of t. The node t is said to be a **transitive fanout** of s in this case.

A Boolean variable is associated with each node of η. For each node s not a primary input, a single-output Boolean function is associated. This is a function of the variables associated with the fanins of s.

We make no distinction between a node of a Boolean network and the Boolean variable

associated with it; the variable of a node s is denoted by s.

For each node of a given Boolean network η, one can uniquely define a Boolean function represented in terms of the primary inputs. Such a function is referred to as a *global function* of the node. Specifically, the global function of a primary input node x is given by x itself, while the global function of a node s, not a primary input, is given by substituting each variable of the function defined at s with the global function of the node corresponding to the variable. Consider a multiple-output Boolean function g such that the function of the i-th output of g, g_i, is the global function of the i-th primary output node of Z. We call g the *functionality* of η.

We assume that for a given Boolean network η, a Boolean relation M, defined between the Boolean spaces spanned by the primary inputs and the primary outputs respectively, is given to specify the allowed functionalities of η. We represent the relation by its characteristic function, $M : B^{|X|} \times B^{|Z|} \rightarrow B$ with $B = \{0, 1\}$, and call it the *specification* of η. Namely, $M(\mathbf{x}, \mathbf{z}) = 1$ if and only if $(\mathbf{x}, \mathbf{z}) \in M$, and we make no distinction between a relation and its characteristic function in the sequel. The specification M provides all possible combinations of the values of the primary inputs and the primary outputs that are allowed to be realized on η, and we say that η *meets* the specification M if for all minterms $\mathbf{x} \in B^{|X|}$ of the primary inputs, $M(\mathbf{x}, g(\mathbf{x})) = 1$, where g is the functionality of η. We assume, without loss of generality, that the original functionality of η meets the specification M.

The model we use in this chapter is a clustered Boolean network, which is similar to a Boolean network except that the function associated with each node may have more than one output. We regard a clustered Boolean network as one composed from a Boolean network.

Definition: Clustered Boolean Network

Given a Boolean network $\eta = (N, E)$, a directed acyclic graph $\Gamma = (V, K)$ is a **clustered Boolean network** if the following properties hold:

1. $\forall v \in V : v \subseteq N_I \cup Z$
2. $\forall (u, v) \in V \times V : u \neq v \Rightarrow u \cap v = \phi$
3. $\bigcup_{v \in V} v = N_I \cup Z$
4. $\forall (u, v) \in V \times V, u \neq v : [u, v] \in K \Leftrightarrow \exists (s, t) \in u \times v : [s, t] \in E$

Γ is a partition of the nodes of η other than the primary inputs, where each $v \in V$ contains at least one node of η. A node $v \in V$ is called a *cluster*, and is distinguished from a node of the

original Boolean network. An edge [u, v] exists in Γ from a cluster u to a cluster v if and only if there exists a pair of nodes of η, $(s,t) \in u \times v$, such that an edge $[s,t]$ exists in η. Note that an edge in K is defined only for a pair of distinct clusters, and there is no self-loop for a single cluster. The definitions of fanins , fanouts , transitive fanins , and transitive fanouts follow those of Boolean networks.

For each cluster $v \in V$, a variable for a node $s \in N$ is an *input variable* of v if s is not in v and there is a node t in v such that an edge $[s,t]$ exists in η. A variable $s \in N$ is an *output variable* of the cluster v if s is in v and either s is a primary output node or there is a node t outside v such that an edge $[s,t]$ exists in η. The set of input variables and the output variables of a cluster v are denoted by I_v and O_v, respectively. Associated with v is a multi-output Boolean function from the Boolean space spanned by I_v to that by O_v. The function is given by the functions of η originally associated with the nodes of v.

As with a Boolean network, we can define the functionality of a clustered Boolean network as a multi-output Boolean function from the primary inputs X to the primary outputs Z that is given by composing the functions associated with the clusters of the network. We say the clustered Boolean network meets the specification in the same way as defined for Boolean networks.

2.3 The Maximum Set of Permissible Functions

We consider the problem of finding a set of functions that can be realized at a particular cluster of a clustered Boolean network so that the resulting functionality of the network meets a given specification, where the functions of the rest of the clusters are all fixed. We call each such function a *permissible function* of the cluster.

Definition: Permissible Function

Given a clustered Boolean network $\Gamma = (V, K)$ with a specification M, a function $f : B^{|I_v|} \rightarrow B^{|O_v|}$ is said to be **permissible** at a cluster $v \in V$, if the functionality of Γ given by replacing the function of v with f meets the specification M.

Brayton and Somenzi showed that the maximum set of permissible functions at a given cluster can be computed and represented by a single Boolean relation [11]. Here, a set $\mathcal{F}$ of Boolean functions, $\mathcal{F} \subseteq \{f \mid f : B^n \rightarrow B^m\}$, is said to be represented by a Boolean relation $F : B^n \times B^m \rightarrow B$, if $f \in \mathcal{F} \Leftrightarrow \forall \mathbf{x} \in B^n : F(\mathbf{x}, f(\mathbf{x})) = 1$, i.e. $\mathcal{F} = \{f \mid \forall \mathbf{x} \in B^n : F(\mathbf{x}, f(\mathbf{x})) = 1\}$.

We first clarify the condition under which a set of functions can be represented by a relation in the sense above. We start with the definition of the following operation.

Definition: Output Selection

The **output selection** is a non-deterministic operation defined over a pair of functions with the same input and output spaces, say $f : B^n \rightarrow B^m$ and $g : B^n \rightarrow B^m$, which returns one of the functions $h : B^n \rightarrow B^m$ such that for all $\mathbf{x} \in B^n$, $h(\mathbf{x})$ is equal to either $f(\mathbf{x})$ or $g(\mathbf{x})$.

Lemma 2.3.1 *A set of Boolean functions, $\mathcal{F} \subseteq \{f \mid f : B^n \rightarrow B^m\}$, is represented by a Boolean relation, if and only if $\mathcal{F}$ is closed under the output selection operation.*

Proof: Suppose that $\mathcal{F}$ can be represented by a Boolean relation $F : B^n \times B^m \rightarrow B$, i.e. $f \in \mathcal{F}$ if and only if $F(\mathbf{x}, f(\mathbf{x})) = 1$ for all $\mathbf{x} \in B^n$. We show that $\mathcal{F}$ is closed under the output selection operation in this case. Consider a pair of functions $(f, g) \in \mathcal{F} \times \mathcal{F}$, where f and g might be identical. Let h be a function obtained by output selection over f and g. Then for each $\mathbf{x} \in B^n$, $h(\mathbf{x})$ is equal to either $f(\mathbf{x})$ or $g(\mathbf{x})$, and thus $F(\mathbf{x}, h(\mathbf{x})) = 1$ since both f and g are members of $\mathcal{F}$. Hence h is a member of $\mathcal{F}$, and $\mathcal{F}$ is closed under the output selection operation.

Conversely, suppose $\mathcal{F}$ is closed under the output selection operation. We show that there exists a Boolean relation $F : B^n \times B^m \rightarrow B$ such that $f \in \mathcal{F}$ if and only if $F(\mathbf{x}, f(\mathbf{x})) = 1$ for all $\mathbf{x} \in B^n$. Consider a relation F such that $F(\mathbf{x}, \mathbf{y}) = 1$ if and only if there exists a function $f \in \mathcal{F}$ for which $\mathbf{y} = f(\mathbf{x})$. Then for an arbitrary function $f \in \mathcal{F}$, the relation F satisfies the property that $F(\mathbf{x}, f(\mathbf{x})) = 1$ for all $\mathbf{x} \in B^n$. The proof is complete if we show that for every function f with the property that $F(\mathbf{x}, f(\mathbf{x})) = 1$ for all $\mathbf{x} \in B^n$, f is a member of $\mathcal{F}$. We first extend the definition of the output selection operation so that it can be defined for more than two operands. Namely, for a given finite set of functions $\{f_1, \ldots, f_r\}$ which share the same input and output spaces, the extended output selection operation returns a function f such that for each input $\mathbf{x}$, there exists $i \in \{1, \ldots, r\}$ for which $f(\mathbf{x}) = f_i(\mathbf{x})$. It is easy to see that the closedness of the set $\mathcal{F}$ under the original definition of the output selection operation implies the closedness under the extended definition. Now, consider a function f with the property that $F(\mathbf{x}, f(\mathbf{x})) = 1$ for all $\mathbf{x} \in B^n$. We show that f is a member of the set $\mathcal{F}$. For an arbitrary input $\mathbf{x} \in B^n$, since $F(\mathbf{x}, f(\mathbf{x})) = 1$, the definition of the relation F implies that there exists a function in $\mathcal{F}$ whose output for the input $\mathbf{x}$ is equal to $f(\mathbf{x})$. Denoting such a function by $f^{(\mathbf{x})}$, we see that f can be obtained as a result of output selection over a set of functions $\{f^{(\mathbf{x})} \mid \mathbf{x} \in B^n\}$. Since $\mathcal{F}$ is closed under this operation, f is a member of $\mathcal{F}$, which completes the proof. ■

It is then claimed that the maximum set of permissible functions at a given cluster can be represented by a Boolean relation.

Theorem 2.3.1 *The maximum set of permissible functions defined at a cluster of a given clustered Boolean network with a specification can be represented by a Boolean relation.*

Proof: Let $\mathcal{F} \subseteq \{f \mid f : B^{|I_v|} \rightarrow B^{|O_v|}\}$ be the maximum set of permissible functions at a cluster v. We denote the clustered Boolean network by Γ and the specification by $M : B^{|X|} \times B^{|Z|} \rightarrow B$. By Lemma 2.3.1, the proof is done if we show that $\mathcal{F}$ is closed under output selection. Consider a pair of functions $(f, g) \in \mathcal{F} \times \mathcal{F}$, where f and g might be identical. Let h be a function given by output selection over f and g. We show that h is a member of $\mathcal{F}$. Suppose the contrary, i.e. $h \notin \mathcal{F}$. Since $\mathcal{F}$ is maximum, it implies that h is not a permissible function for the cluster v. Then there exists a minterm of the primary inputs, $\mathbf{x} \in B^{|X|}$, for which the value $\mathbf{z}$ obtained at the primary outputs of the network Γ by replacing the function of the cluster v by h does not meet the specification, i.e. $M(\mathbf{x}, \mathbf{z}) = 0$. Let $\mathbf{u} \in B^{|I_v|}$ be the value obtained at the input variables of the cluster v for such an $\mathbf{x}$. By definition of the output selection, $h(\mathbf{u})$ is equal to either $f(\mathbf{u})$ or $g(\mathbf{u})$. Suppose, without loss of generality, $h(\mathbf{u}) = f(\mathbf{u})$. Then if we replace the function of the cluster v by f in the network Γ and apply $\mathbf{x}$ to the primary inputs, the value obtained at the primary outputs is identical with the one obtained when h is realized at v. Since that value $\mathbf{z}$, obtained at the primary outputs, does not meet the specification M for the input $\mathbf{x}$, f is not a permissible function. This conflicts with the fact that f is a member of $\mathcal{F}$. Hence, our assumption that h is not a permissible function is incorrect. Therefore, the set $\mathcal{F}$ is closed under the output selection operation, and there exists a Boolean relation which represents the set. ■

We show by an example how the maximum set of permissible functions can be represented by a Boolean relation. Consider a clustered Boolean network shown in Figure 2.1 with two primary inputs and two primary outputs, where a circle represents a cluster in the figure. Suppose we want to compute the maximum set of permissible functions at the cluster v for the specification shown in Table 2.1-(a). The cluster v has two inputs x_1 and x_2, and two outputs v_1 and v_2. The function associated with each cluster other than v is given in Figure 2.1. For this example, the maximum set consists of four functions as shown in in Table 2.2, which are denoted by $\{f_1, \ldots, f_4\}$. This set can be represented by a Boolean relation F given in Table 2.1-(b). The table shows that a pair $(x_1x_2, v_1v_2) = (00, 10)$ is a member of F, $(00, 01)$ is another member of F, and so on. Namely, the relation F consists of all possible pairs of input and output values allowed at the cluster v to satisfy the specification. We first note that this set of functions cannot be represented by a Boolean

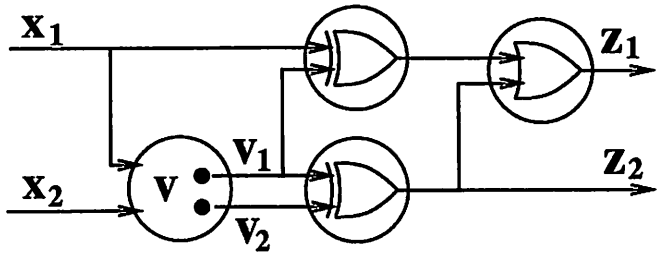

Figure 2.1: Clustered Boolean Network

x_1x_2	z_1z_2
00	11
01	00
10	00
11	11

(a)

x_1x_2	v_1v_2
00	10, 01
01	00
10	11
11	10, 01

(b)

Table 2.1: The Specification (a) and a Boolean Relation F (b)

	f_1	f_2	f_3	f_4
x_1x_2	v_1v_2	v_1v_2	v_1v_2	v_1v_2
00	10	10	01	01
01	00	00	00	00
10	11	11	11	11
11	10	01	10	01

Table 2.2: The Maximum Set of Permissible Functions for Cluster v

function with don't cares. Recall that in general, one can use don't care conditions to represent a set of functions. Namely, for a given Boolean function which may have multiple outputs, one specifies a set of input minterms that are treated as don't cares for each output of the function. It is then interpreted that for each output, the value of the output may be either 0 or 1 for those don't care inputs, while it must coincide with the value of the originally given function for the rest of the input minterms. In this way, one can represent a set of functions using don't care conditions. However, it is not the case for the set of functions shown in Table 2.2. This is because the set of don't care minterms must be specified for each output separately. Namely, we must say that for the output v_1, the minterms $x_1x_2 = 00$ and $x_1x_2 = 11$ are don't cares, since the value of v_1 may be either 0 or 1 for these input minterms. Similarly, $x_1x_2 = 00$ and $x_1x_2 = 11$ are don't cares for the output v_2. Then, since $x_1x_2 = 00$ is a don't care for both outputs, this implies that we can output an arbitrary minterm at v_1v_2 for this input, and thus $v_1v_2 = 00$ or $v_1v_2 = 11$ are also feasible outputs, which is a wrong conclusion. The point here is that since don't cares are specified for each output, in case of multi-output functions, disallowed output patterns might be included as feasible outputs, and thus representations using don't cares might be inadequate. Using a Boolean relation, one can explicitly specify which output pattern should be allowed for each input minterm. Specifically, for our example, the inclusion of a pair of input and output minterms in the relation is determined by checking if there is a permissible function that realizes the pair. However, as seen in the beginning of this section, not all sets of functions can be represented by using relations. Observe that in forming a relation for our example, the inclusion of one pair in the relation is independent of the inclusion of another. Conversely, for a given relation F, one can obtain a permissible function by choosing one pair for each input minterm, i.e. one output minterm for each row of Table 2.1-(b), where the choice made for one input minterm is independent of the choice for another. This property allows us to represent a set of functions by a relation. In fact, if we take a subset of the maximum set of permissible functions, we cannot represent it by a Boolean relation in general. The subset $\{f_1, f_2, f_3\}$ is one such example. This is because the subset is not closed under output selection; f_4 can be obtained as a result of the output selection over f_2 and f_3, but is not in the subset. In some sense, it is the maximality of the set of functions that makes it possible to use a relation for representing the set. We will see, in the following section, another type of set of functions that can also be represented by a relation. They also have a maximality property, in a different sense, and thus can be represented by a relation.

2.4 Maximally Compatible Sets of Permissible Functions

2.4.1 Maximally Compatible Sets and Boolean Relations

We now consider the problem of computing a set of permissible functions for every cluster of a network. The resulting set of sets of permissible functions, one set defined for each cluster, has the property of compatibility defined as follows:

Definition: Compatible Sets of Permissible Functions

Given a clustered Boolean network $\Gamma = (V, K)$ with a specification M, a set of sets of functions defined over the clusters of the network is said to be **a set of compatible sets of permissible functions**, if for an arbitrary selection of functions, one from each set, the functionality of Γ given by simultaneously replacing the function of each cluster with the selected function for that cluster meets the specification M.

Namely, one can choose an arbitrary function among the set defined for each cluster independently, and the property of compatibility guarantees that the resulting functionality of the network meets the specification.

Note that for the maximum set of permissible functions defined at a single cluster v, it was assumed that the functions of the rest of the clusters of the network were all fixed. Thus, the degree of freedom given at one cluster depends upon the others; if one changes the function associated with another cluster, the set of permissible functions originally computed at v is not necessarily valid, and one may have to re-compute it. This is not the case for compatible sets of permissible functions.

Compatible sets of permissible functions are said to be *maximal* if no new function can be added to any one of the sets without destroying the property of compatibility. Note that in general maximally compatible sets are not unique for a given clustered Boolean network with a specification; the sets are maximally compatible as long as strictly larger compatible sets do not exist. Maximally compatible sets are of interest because they can be computed and used independently, possibly by parallel processing. In this section, we present a procedure for computing a set of maximally compatible sets of permissible functions.

We first show that for any given set of maximally compatible sets, the set of permissible functions defined at each cluster can be represented by a Boolean relation.

Theorem 2.4.1 *For a set of maximally compatible sets of permissible functions defined over the clusters of a given clustered Boolean network with a specification, each set defined at a single*

cluster can be represented by a Boolean relation.

Proof: Let $\mathcal{F}_v \subseteq \{f \mid f : B^{|I_v|} \rightarrow B^{|O_v|}\}$ be the set defined at a cluster v. We denote the network by Γ and the specification by $M : B^{|X|} \times B^{|Z|} \rightarrow B$. By Lemma 2.3.1, the proof is done if we show that the set $\mathcal{F}_v$ is closed under the output selection operation. Consider an arbitrary pair of functions $(f, g) \in \mathcal{F}_v \times \mathcal{F}_v$, where f and g may be identical. Let h be a function given by output selection over f and g. We show that h is a member of $\mathcal{F}_v$. Suppose for contrary that $h \notin \mathcal{F}_v$. Since $\mathcal{F}_v$ is maximal, by definition of compatibility, there exists a combination of functions, where one function is chosen for each cluster from the corresponding set of the maximally compatible sets, such that if we replace the function of v by h and the function of each of the other clusters by the one in the combination, then the resulting functionality of Γ does not meet the specification. Specifically, there exists a minterm of the primary inputs $\mathbf{x} \in B^{|X|}$ such that for the corresponding minterm $\mathbf{z} \in B^{|Z|}$ of the primary outputs obtained by Γ under this replacement, $M(\mathbf{x}, \mathbf{z}) = 0$. Let $\mathbf{u} \in B^{|I_v|}$ be the value obtained at the input variables of the cluster v for this $\mathbf{x}$. By definition of output selection, $h(\mathbf{u})$ is equal to either $f(\mathbf{u})$ or $g(\mathbf{u})$. Suppose, without loss of generality, $h(\mathbf{u}) = f(\mathbf{u})$. Then if, instead of using h for the function of the cluster v in the replacement above, we replace the function of the cluster v by f, and if we apply $\mathbf{x}$ to the primary inputs of the network, then the same value $\mathbf{z}$ is obtained at the primary outputs. Since the value $\mathbf{z}$ does not meet the specification M for the input $\mathbf{x}$, the combination of the functions above, where f is chosen for the cluster v, is not permissible. This is conflict with the property of compatibility. Therefore h is a member of $\mathcal{F}_v$, and $\mathcal{F}_v$ is closed under output selection. Hence the set can be represented by a Boolean relation. ■

2.4.2 A Procedure for Computing Maximally Compatible Sets

We now present a procedure for computing a set of maximally compatible sets of permissible functions for a given clustered Boolean network $\Gamma = (V, K)$ with a specification $M : B^{|X|} \times B^{|Z|} \rightarrow B$. The proposed procedure sorts the clusters of Γ first, and processes one cluster at a time in this order. Therefore, the result of the procedure is order dependent. The order of the clusters satisfies the property that for all pairs of clusters $\{u, v\}$, if u is a fanin of v, i.e. $[u, v] \in K$, then v precedes u in the order. Therefore, the procedure starts with a cluster with only primary outputs, and when a cluster is processed, all the fanout clusters have been already processed.

We denote by σ the order of the clusters employed in the procedure, and assume that the clusters are processed in the increasing order of σ. Then when a cluster v is processed, a cluster

w with $\sigma(w) < \sigma(v)$ has been processed, and a set of functions $\mathcal{F}_w \subseteq \{f \mid f_w : B^{|I_w|} \rightarrow B^{|O_w|}\}$ has been already computed for w. The procedure computes a set of functions at v, $\mathcal{F}_v \subseteq \{f \mid f_v : B^{|I_v|} \rightarrow B^{|O_v|}\}$, such that $\mathcal{F}_v$ satisfies the following property with respect to the set of sets of functions computed for the already processed clusters:

Property 1: forward compatibility property

A function $f_v : B^{|I_v|} \rightarrow B^{|O_v|}$ is a member of $\mathcal{F}_v$, if and only if for an arbitrary combination of functions for the processed clusters, $C_v = \{f_w : B^{|I_w|} \rightarrow B^{|O_w|} \mid f_w \in \mathcal{F}_w$ and $\sigma(w) < \sigma(v)\}$, the functionality of Γ meets the specification by replacing the function of v with f_v and the function of w with $f_w \in C_v$ for each w such that $\sigma(w) < \sigma(v)$, where a function f_u at each cluster u such that $\sigma(u) > \sigma(v)$ is fixed to the one originally associated with the cluster.

Intuitively, we compute $\mathcal{F}_v$ so that the resulting sets of functions associated with all the already processed clusters are maximally compatible, in the sense that any combination of functions for these clusters, together with the original functions for the unprocessed clusters, leads to a functionality of the entire network which meets the specification, and no new function can be added to any of these already processed clusters without destroying this property. Then it immediately follows that at the end of the procedure, when all the clusters have been processed, we obtain maximally compatible sets of permissible functions.

Theorem 2.4.2 *Given a set of sets of functions $C = \{\mathcal{F}_v \mid v \in V\}$, suppose that $\mathcal{F}_v$ satisfies the forward compatibility property for each $v \in V$. Then C is a set of maximally compatible sets of permissible functions.*

Proof: By definition, it is easy to see that C is a set of compatible sets of permissible functions. We show the maximal compatibility of C. We first note that for each cluster v, the function originally associated with v in Γ is a member of $\mathcal{F}_v$. This is because when the cluster w which is immediately before v has been processed, an arbitrary combination of functions for the processed part, $C_v = \{f_w : B^{|I_w|} \rightarrow B^{|O_w|} \mid f_w \in \mathcal{F}_w$ and $\sigma(w) < \sigma(v)\}$, together with the original functionality for the rest of the clusters, leads to a functionality of Γ that meets the specification, and thus by definition of Property 1, the function originally associated with v is a member of $\mathcal{F}_v$.

Now, suppose for contrary that there exists a function f_v, at some cluster v, such that $f_v \notin \mathcal{F}_v$ and the set of sets of functions given by replacing $\mathcal{F}_v$ with $\mathcal{F}_v \cup \{f_v\}$ in C is still a set of compatible sets of permissible functions. We denote the resulting set of sets of functions by $\hat{C}$. Consider an arbitrary combination of functions $C_v = \{f_w : B^{|I_w|} \rightarrow B^{|O_w|} \mid f_w \in \mathcal{F}_w$ and

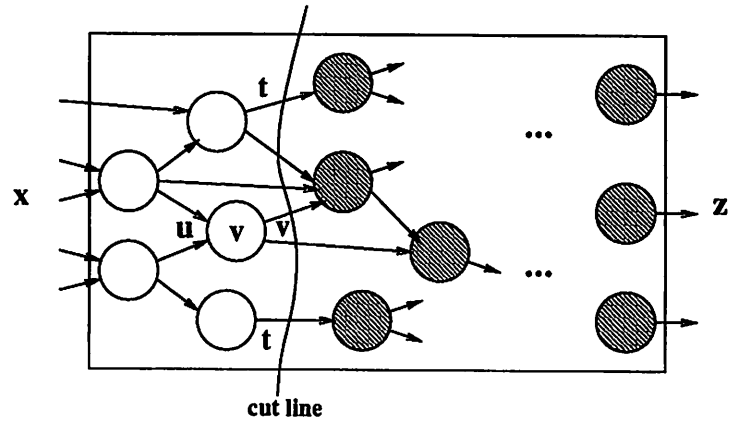

Figure 2.2: Computing Maximally Compatible Sets of Permissible Functions

$\sigma(w) < \sigma(v)\}$. Let U be the set of functions originally associated with the clusters u such that $\sigma(u) > \sigma(v)$. Note that $f_u \in \mathcal{F}_u$ for each $f_u \in U$, while $f_w \in \mathcal{F}_w$ for each $f_w \in C_v$. Suppose we realize $f_w \in C_v$ at each cluster w such that $\sigma(w) < \sigma(v)$, f_v at the cluster v, and $f_u \in U$ at each u such that $\sigma(u) > \sigma(v)$. Then, since $\hat{C}$ is compatible, the resulting functionality of the network Γ meets the specification. Since $\mathcal{F}_v$ satisfies the forward compatibility property, we see that f_v must be a member of $\mathcal{F}_v$, which is a contradiction. Hence, the set C is a set of maximally compatible sets of permissible functions. ■

We present how to compute such a set $\mathcal{F}_v$ for each cluster. Suppose a cluster v is being processed. First, define the cut line on the edges of Γ to partition the clusters into two classes; one is the set of clusters already processed, the other consists of all the unprocessed clusters. Such a cut is uniquely defined for a given cluster v being processed and an order σ. Figure 2.2 illustrates the situation, where the shaded clusters designate those already processed. Note that all the out-going edges of v crosses the cut. Let T be the set of Boolean variables which cross the cut but are not the output variables O_v of v. More specifically, a variable is in T if and only if it is an output variable of an unprocessed cluster other than v and either it is a primary output or is an input variable of some processed cluster. By definition, T and O_v do not intersect, and their union gives the complete set of Boolean variables crossing the cut.

Let $g^{(t)} : B^{|X|} \rightarrow B^{|T|}$ be a Boolean function between the primary inputs and the T variables defined above, which reflects the original functionality of the unprocessed clusters.

Namely, $g^{(t)}(\mathbf{x})$ provides the value that appears at the T variables in the original network Γ for the primary input $\mathbf{x}$; $g^{(t)}$ is the set of global functions for the T variables. Similarly, let $g^{(u)} : B^{|X|} \rightarrow B^{|I_v|}$ be a Boolean function between the primary inputs and the input variables of the cluster v such that $g^{(u)}(\mathbf{x}) = \mathbf{u}$ if and only if $\mathbf{u}$ is given at the input variables of v by the original functions associated with the unprocessed clusters for the primary input $\mathbf{x}$. Now, in order to compute $\mathcal{F}_v$, we use a Boolean relation $H : B^{|O_v \cup T|} \times B^{|Z|} \rightarrow B$ defined between the variables crossing the cut line, i.e. $O_v \cup T$, and the primary outputs Z. It satisfies the following property:

Property 2: existential property

$H(\mathbf{vt}, \mathbf{z}) = 1$ if and only if there exists a combination of functions for the already processed clusters, $C_v = \{f_w : B^{|I_w|} \rightarrow B^{|O_w|} \mid f_w \in \mathcal{F}_w \text{ and } \sigma(w) < \sigma(v)\}$, with which the value $\mathbf{z}$ is obtained at the primary outputs by setting the values of O_v and T to $\mathbf{v}$ and $\mathbf{t}$, respectively.

Recall that $\mathcal{F}_w$ is a set of functions computed for a cluster w, where such a set exists for each w with $\sigma(w) < \sigma(v)$. Intuitively, the relation H shows what values *can* appear at the primary outputs for a given value at the cut, and $H(\mathbf{vt}, \mathbf{z}) = 1$ as long as there exists a combination of functions for the processed part for which $\mathbf{vt}$ is mapped into $\mathbf{z}$. We then compute a Boolean relation $F_v : B^{|I_v|} \times B^{|O_v|} \rightarrow B$ defined at the cluster v such that $F_v(\mathbf{u}, \mathbf{v}) = 1$ if and only if

$$\forall(\mathbf{x}, \mathbf{z}) \in B^{|X|} \times B^{|Z|} : \mathbf{u} = g^{(u)}(\mathbf{x}) \text{ and } H(\mathbf{v}g^{(t)}(\mathbf{x}), \mathbf{z}) = 1 \Rightarrow M(\mathbf{x}, \mathbf{z}) = 1 \qquad (2.1)$$

It is then claimed that the set $\mathcal{F}_v$ of functions given by the relation F_v, i.e. $\mathcal{F}_v = \{f_v : B^{|I_v|} \rightarrow B^{|O_v|} \mid \forall \mathbf{u} : F_v(\mathbf{u}, f_v(\mathbf{u})) = 1\}$, satisfies the forward compatibility property (Property 1). Intuitively, we can interpret the computation of F_v above as follows. By the forward compatibility property, we want to compute a maximal set of functions at the cluster v so that, for an arbitrary combination of functions chosen from the sets computed so far, together with the original functionality for the unprocessed clusters, the resulting functionality of the entire network always meets the specification. Therefore, for a given pair of minterms $(\mathbf{u}, \mathbf{v}) \in B^{|I_u|} \times B^{|O_v|}$, we look at each primary input $\mathbf{x} \in B^{|X|}$ and each primary output $\mathbf{z} \in B^{|Z|}$, and include the pair $(\mathbf{u}, \mathbf{v})$ in the relation F_v if and only if $(\mathbf{x}, \mathbf{z})$ meets the specification for any possible primary output $\mathbf{z}$. A value $\mathbf{z}$ is *possible* if and only if the $\mathbf{u}$ value appears at the input variables of the cluster v for the minterm $\mathbf{x}$, and the minterm $\mathbf{vt}$ *can* be mapped into $\mathbf{z}$ by the already processed clusters, where $\mathbf{t} = g^{(t)}(\mathbf{x})$. A formal proof of this claim is found in Theorem 2.4.3.

In the actual computation of F_v above, each of the operands is represented by a binary

decision diagram (BDD) [12], and the operations are performed on BDD's. A BDD is a data structure to represent a single-output Boolean function, and thus the characteristic function of the relation H or M can be directly represented by BDD's. For a multi-output function, such as $g^{(u)} : B^{|X|} \rightarrow B^{|I_v|}$, this is done by defining a relation $G^{(u)} : B^{|X|} \times B^{|I_v|} \rightarrow B$ such that $G^{(u)}(\mathbf{x}, \mathbf{u}) = 1$ if and only if $\mathbf{u} = g^{(u)}(\mathbf{x})$. $G^{(u)}$ is represented by a BDD.

Once F_v is computed, the procedure moves on to the next cluster according to the order σ. At this point, we need to update the cut line, since now the cluster v has been processed. The new cut line is shown in Figure 2.3. Since the cut line has been updated, we also need to update the relation H. The new relation, say $\tilde{H}$, has the input part I_v and T, the output part Z, and satisfies the existential property (Property 2) where the cluster v is replaced by the one being processed in the statement. Specifically, such a relation $\tilde{H} : B^{|I_v \cup T|} \times B^{|Z|} \rightarrow B$ is defined as $\tilde{H}(\mathbf{ut}, \mathbf{z}) = 1$ if and only if

$$\exists \mathbf{v} \in B^{|O_v|} : F_v(\mathbf{u}, \mathbf{v}) = 1 \text{ and } H(\mathbf{vt}, \mathbf{z}) = 1 \tag{2.2}$$

Hence, the relation H is dynamically updated using the relation F_v just computed for the cluster v and the original relation H. In the beginning of the procedure, the output variables O_v of the cluster v processed first, i.e. the one on the top of the order σ, are all primary outputs. T variables defined in the beginning are also primary outputs, and the union of O_v and T is equal to the set of primary outputs Z. Hence, both the input part and the output part of the relation H given in the beginning are the primary outputs, and we initialize it as $H(\mathbf{vt}, \mathbf{z}) = 1$ if and only if $\mathbf{vt}$ and $\mathbf{z}$ are identical. The procedure terminates when all the clusters have been processed.

The correctness of the proposed procedure is claimed below.

Theorem 2.4.3 *The set of sets of functions given at the end of the proposed procedure over the clusters of* Γ *is a set of maximally compatible sets of permissible functions.*

Proof: We show by induction that the forward compatibility property (Property 1) and the existential property (Property 2) hold for the relations F_v and H, respectively.

For the base case, let v be the cluster on the top of the order σ. We first show that the forward compatibility property holds for the relation F_v computed by the formula (2.1). Since no cluster has been processed before, the following statement is equivalent to the property; a function f_v is a member of $\mathcal{F}_v$ if and only if the functionality of Γ meets the specification by replacing the function of v with f_v. Now, recall that the output variables O_v of v are all primary outputs, and T variables are initially defined as the rest of the primary outputs. Since the functionality of Γ meets

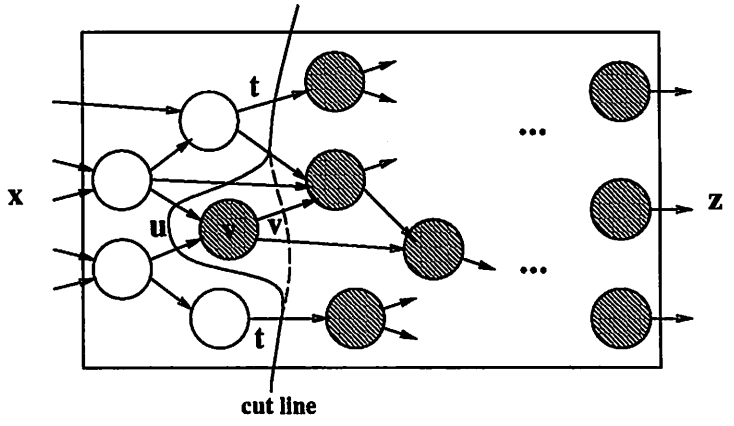

Figure 2.3: Updating the Cut Line during the Procedure

the specification with a function f_v if and only if for each $\mathbf{x} \in B^{|X|}$, the value $\mathbf{z}$ obtained at the primary outputs meets the specification, i.e. $M(\mathbf{x}, \mathbf{z}) = 1$, f_v has the property that $f_v(\mathbf{u}) = \mathbf{v}$ if and only if

$$\forall \mathbf{x} \in B^{|X|} : \mathbf{u} = g^{(u)}(\mathbf{x}) \Rightarrow M(\mathbf{x}, \mathbf{z}) = 1,$$

where $\mathbf{z} = \mathbf{v}g^{(t)}(\mathbf{x})$. Since the relation H is initialized so that $H(\mathbf{vt}, \mathbf{z}) = 1$ if and only if $\mathbf{vt}$ and $\mathbf{z}$ are identical, the formula above is equivalent to the formula (2.1). Therefore, $f_v(\mathbf{u}) = \mathbf{v}$ if and only if $F_v(\mathbf{u}, \mathbf{v}) = 1$. Hence, the functionality of Γ meets the specification for a function f_v if and only if f_v is a member of $\mathcal{F}_v$, and the forward compatibility property holds.

Suppose that this F_v and the initial relation H are used to update the relation H according to the formula (2.2). We prove that the resulting relation $\tilde{H}(\mathbf{ut}, \mathbf{z})$ satisfies the existential property. Since v is the only cluster processed so far, the property in this case is that $\tilde{H}(\mathbf{ut}, \mathbf{z}) = 1$ if and only if there exists a function $f_v \in \mathcal{F}_v$ with which the value $\mathbf{z}$ is obtained at the primary outputs by setting I_v and T to $\mathbf{u}$ and $\mathbf{t}$, respectively. Since O_v and T are the primary outputs, this is equivalent to saying that $\tilde{H}(\mathbf{ut}, \mathbf{z}) = 1$ if and only if there exists $\mathbf{v} \in B^{|O_v|}$ such that $F_v(\mathbf{u}, \mathbf{v}) = 1$ and $\mathbf{z} = \mathbf{vt}$. Due to the initialization of the relation H, it is equivalent to the formula (2.2), and thus the existential property holds.

Now consider the induction step. Let v be the cluster being processed. Suppose that the relation $H : B^{|O_v \cup T|} \times B^{|Z|} \rightarrow B$ has been computed so that it satisfies the existential property. We first show that the forward compatibility property holds for F_v given by the formula (2.1). Let f_v

be a function of $\mathcal{F}_v$. We show that for an arbitrary combination of functions for the clusters already processed, say $C_v = \{f_w : B^{|I_w|} \rightarrow B^{|O_w|} \mid f_w \in \mathcal{F}_w \text{ and } \sigma(w) < \sigma(v)\}$, the functionality of Γ meets the specification by replacing the function of v with f_v and the function of w with $f_w \in C_v$ for each w such that $\sigma(w) < \sigma(v)$. Consider an arbitrary such C_v. For an arbitrary primary input $\mathbf{x} \in B^{|X|}$, let $\mathbf{z}$ be the resulting minterm of the primary outputs obtained by this C_v and f_v. What we want to show is that this $\mathbf{z}$ meets the specification for the minterm $\mathbf{x}$. Let $\mathbf{u}$ be the value obtained at the input variables of the cluster v for this $\mathbf{x}$, i.e. $\mathbf{u} = g^{(u)}(\mathbf{x})$. Since f_v is a member of $\mathcal{F}_v$, $F_v(\mathbf{u}, f_v(\mathbf{u})) = 1$, and thus the formula (2.1) holds for this $\mathbf{u}$ and $\mathbf{v} = f_v(\mathbf{u})$. Due to the induction hypothesis, the relation H satisfies the existential property, and thus $H(\mathbf{v}g^{(t)}(\mathbf{x}), \mathbf{z}) = 1$. Therefore, the formula (2.1) implies that $M(\mathbf{x}, \mathbf{z}) = 1$, and thus the specification is met. Conversely, suppose that a given function $f_v : B^{|I_v|} \rightarrow B^{|O_v|}$ has a property that the network Γ meets the specification for this f_v with an arbitrary $C_v = \{f_w : B^{|I_w|} \rightarrow B^{|O_w|} \mid f_w \in \mathcal{F}_w \text{ and } \sigma(w) < \sigma(v)\}$. We prove that f_v is a member of $\mathcal{F}_v$. Specifically, we show that for all $\mathbf{u} \in B^{|I_v|}$, $F_v(\mathbf{u}, f_v(\mathbf{u})) = 1$. For a given $\mathbf{u} \in B^{|I_v|}$, consider an arbitrary pair of minterms of the primary inputs and the primary outputs, say $(\mathbf{x}, \mathbf{z})$, such that $\mathbf{u} = g^{(u)}(\mathbf{x})$ and $H(\mathbf{v}g^{(t)}(\mathbf{x}), \mathbf{z}) = 1$, where $\mathbf{v} = f_v(\mathbf{u})$. Since the relation H satisfies the existential property due to the induction hypothesis, there exists a combination of functions for the clusters already processed, say $C_v = \{f_w : B^{|I_w|} \rightarrow B^{|O_w|} \mid f_w \in \mathcal{F}_w \text{ and } \sigma(w) < \sigma(v)\}$, with which $\mathbf{z}$ is obtained at the primary outputs by setting the values of O_v and T to $\mathbf{v}$ and $g^{(t)}(\mathbf{x})$. Since f_v has a property that Γ meets the specification for an arbitrary such C_v, $M(\mathbf{x}, \mathbf{z}) = 1$. Hence, the formula (2.1) holds for these $(\mathbf{u}, \mathbf{v})$, and thus $F_v(\mathbf{u}, \mathbf{v}) = 1$. Therefore f_v is a member of $\mathcal{F}_v$. This concludes the proof that F_v satisfies the forward compatibility property.

What remains is to show that the relation $\tilde{H}$ updated by the formula (2.2) satisfies the existential property. Recall that the input part of $\tilde{H}$ consists of T and the input variables I_v of the cluster v, and the existential property for $\tilde{H}$ is that $\tilde{H}(\mathbf{ut}, \mathbf{z}) = 1$ if and only if there exists a combination of functions for the processed clusters, $C_v = \{f_w : B^{|I_w|} \rightarrow B^{|O_w|} \mid f_w \in \mathcal{F}_w \text{ and } \sigma(w) \leq \sigma(v)\}$, with which the value $\mathbf{z}$ is obtained at the primary outputs by setting the values of I_v and T to $\mathbf{u}$ and $\mathbf{t}$, respectively. Note that C_v contains the cluster v as well. Since the original relation H satisfies the existential property, such C_v above exists if and only if there exists a function f_v in $\mathcal{F}_v$ such that $H(f_v(\mathbf{u})\mathbf{t}, \mathbf{z}) = 1$. Equivalently, there exists $\mathbf{v} \in B^{|O_v|}$ such that $F_v(\mathbf{u}, \mathbf{v}) = 1$ and $H(\mathbf{vt}, \mathbf{z}) = 1$, which is identical with the formula (2.2). Hence the existential property holds for the updated relation $\tilde{H}$. ■

2.4.3 An Example

We illustrate how the proposed procedure works using the example shown in Figure 2.4. This example was introduced in [15] as a counterexample for which the method of computing compatible sets of permissible functions proposed in [49] fails to compute maximal sets.

Suppose the clusters have been composed so that every cluster consists of exactly one gate, i.e. there are three clusters u, v, and w. The order σ of the clusters is given as $w < v < u$. Suppose also that the specification is given as the functionality realized by the original network, i.e. $M(x_1x_2, z) = 1$ if and only if $z = x_1$.

First, we initialize the relation H as $H(w, z) = 1$ if and only if $w = z$.

1. Cluster w

   The fanin clusters of w are u and v, and $g^{(u)} = x_2$ while $g^{(v)} = x_1 \oplus x_2$, where $x_1 \oplus x_2$ designates the exclusive OR operation between x_1 and x_2. Using the formula (2.1), we obtain the relation F_w as $F_w(uv, w) = (w \equiv (u \oplus v))$, where $f \equiv g$ designates the exclusive NOR operation between f and g.

   Updating the relation H using the formula (2.2), we obtain the new relation $H(uv, z) = (z \equiv (u \oplus v))$.

2. Cluster v

   The fanin cluster of v is u, and $g^{(u)} = x_2$. The current cut line is cut 1 shown in the figure, and thus T variables consist of u as well. Using the formula (2.1), we obtain $F_v(x_1u, v) = (v \equiv (u \oplus x_1))$. Updating H, we obtain $H(x_1u, z) = (z \equiv x_1)$.

3. Cluster u

   The current cut line is cut 2 in the figure, and thus x_1 is the unique T variable in this case. Thus we obtain $F_u(x_2, u) = 1$ for all (x_2, u), i.e. F_u is tautologically one.

Since $F_u = 1$, we can replace u by an arbitrary function. By replacing u by a constant value 0, we can delete both v and w to obtain $z = x_1$.

2.4.4 Computing the Maximum Set of Permissible Functions

The key idea of the proposed procedure is that when a cluster v is processed, we compute a maximal set of functions that are compatible with those computed so far, as stated in the forward

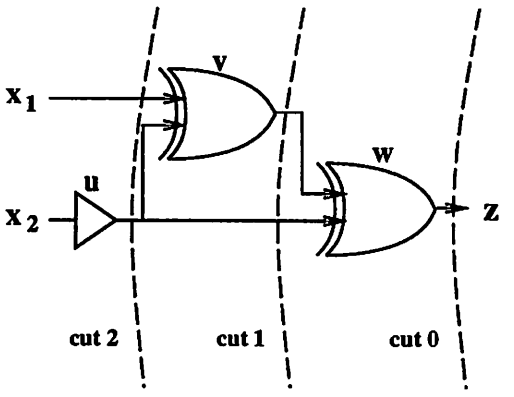

Figure 2.4: An Example for Computing Maximally Compatible Sets of Permissible Functions

compatibility property (Property 1). The relation F_v representing such a set $\mathcal{F}_v$ is computed with the help of the relation H that satisfies the existential property (Property 2).

Now, we notice that when we proved in Theorem 2.4.3 that such a relation H is dynamically computed using the formula (2.2), we did not use the fact that the relation F_w computed for a cluster already processed satisfies the forward compatibility property. This implies that we can interpret the proposed procedure as one that computes, at a give cluster v, a set of functions $\mathcal{F}_v$ that satisfies the forward compatibility property for a given set of sets of functions $\{\mathcal{F}_w\}$ for all the clusters w such that $\sigma(w) < \sigma(v)$, where $\mathcal{F}_w$ is simply a set of functions with the input space I_w and the output space O_w, and may not satisfy the forward compatibility property. Therefore, when the procedure processes each cluster w such that $\sigma(w) < \sigma(v)$, if we enforce the procedure to update the relation H using a set $\mathcal{F}_w$ which consists only of the function originally associated with the cluster w, instead of the one computed by the formula (2.1), then at the cluster v, the procedure computes the complete set of functions f_v defined at v such that the functionality of the network Γ given by replacing the original function of v by f_v, together with the original functionality of the rest of the clusters, meets the specification. Such a set is what we defined as the maximum set of permissible functions for the cluster v in Section 2.3. Hence, we can use the proposed procedure for computing the maximum set of permissible functions.

The argument above can be interpreted in a more general way. Suppose that when the procedure processes a cluster w such that $\sigma(w) < \sigma(v)$, we first compute a Boolean relation F_w

using the formula (2.1), and then replace it by another Boolean relation $\tilde{F}_w$ such that the set of functions represented by $\tilde{F}_w$ is a non-null subset of the one represented by F_w. In other words, for the resulting set $\tilde{\mathcal{F}}_w$ of functions for the cluster w, $\tilde{\mathcal{F}}_w \neq \phi$ and $\tilde{\mathcal{F}}_w \subseteq \mathcal{F}_w$. If we update the relation H using this new relation $\tilde{F}_w$, then when the procedure reaches the cluster v, it computes a set $\mathcal{F}_v$ of functions such that $f_v \in \mathcal{F}_v$ if and only if for an arbitrary combination of functions for the already processed clusters, $C_v = \{f_w : B^{|I_w|} \rightarrow B^{|O_w|} \mid f_w \in \tilde{\mathcal{F}}_w \text{ and } \sigma(w) < \sigma(v)\}$, the functionality of Γ meets the specification by replacing the function of v with f_v and the function of w with $f_w \in C_v$ for each w such that $\sigma(w) < \sigma(v)$. Now, observe that when larger subsets $\tilde{\mathcal{F}}_w$ are used for the preceding clusters, then a smaller set $\mathcal{F}_v$ is obtained at the cluster v[2]. This is because the procedure computes $\mathcal{F}_v$ so that the functionality of Γ meets the specification for all possible combinations of the functions of $\tilde{\mathcal{F}}_w$'s[3]. Therefore, the maximum degree of freedom can be associated with a cluster v when the functionality allowed at the rest of the clusters are maximally restricted, e.g. to the original functionality, while the flexibility for the cluster v tends to be restricted as we allow additional degree of freedom for the other clusters. Hence, we can use the proposed procedure to compute various degree of freedom for each cluster by adjusting the size of the Boolean relations for the preceding clusters.

Also note that in computing the maximum set of permissible functions at a cluster v using the proposed procedure, we don't need to restrict the functionality of the preceding clusters to the original one. Instead, we can choose any one of the permissible functions computed for a preceding cluster w. Specifically, when a cluster w is processed, we first compute a set of functions $\tilde{\mathcal{F}}_w$ as described above, and then choose one of them, possibly an optimal function $\tilde{f}_w \in \tilde{\mathcal{F}}_w$. We then update the relation H using the selected function $\tilde{f}_w$. When the cluster v is visited, the procedure computes the maximum set of permissible functions, F_v, with respect to the set of functions $\{\tilde{f}_w \mid \sigma(w) < \sigma(v)\}$. We then choose a function f_v from the relation F_v, replace the function originally associated with v by f_v, and then proceed to the next cluster. When all the clusters have been processed, a function of the entire network meets the specification. In this sense, we see that the only distinction between the maximum set of permissible functions and the maximally compatible sets of permissible functions is that whether one chooses to use the flexibility immediately when a cluster is visited, to replace its function by a simpler one, or to delay the optimization for later.

[2]To be precise, the cardinality of $\mathcal{F}_v$ never increases if strictly larger subsets are used for the preceding clusters.

[3]Note that although the sets of functions $\{\tilde{\mathcal{F}}_w\}$ with $\mathcal{F}_v$ are not maximally compatible, the resulting $\mathcal{F}_v$ is made maximal in the sense that there is no function newly added to $\mathcal{F}_v$ without destroying the compatibility with the sets $\{\tilde{\mathcal{F}}_w\}$.

2.5 A Clustered Boolean Network

In this section, we present a method of composing a clustered Boolean network from a given Boolean network. The goal in clustering is to group nodes of the original Boolean network into clusters so that the savings of the minimized logic implementations, compared to the ones originally realized in the clusters, are maximized. It is important to develop effective clustering techniques that maximize the potential for minimization.

We have developed a heuristic for partitioning Boolean network nodes to compose clusters based on *sharedness*. The sharedness of a node of a Boolean network with an existing cluster is a measure of how much a set of nodes share common logic. The nodes are grouped in such a way that the resulting clusters consist of functions with mutually high sharedness.

2.5.1 Sharedness

For a given Boolean network $\eta = (N, E)$, the sharedness, W, between two nodes s and t is the number of common minterms of the functions originally associated with the two nodes, which are dependent only on the common variables of the nodes. Specifically, let f and g be the functions associated in the original Boolean network with the nodes s and t respectively. Suppose f and g have common input variables $\{c_1, \ldots, c_k\}$. In general, we denote the set of input variables of the function f by $\{c_1, \ldots, c_k, s_1, \ldots, s_n\}$. Similarly, let $\{c_1, \ldots, c_k, t_1, \ldots, t_m\}$ be the input variables of the function g. Consider a Boolean function $h : B^k \rightarrow B$ defined by the common variables such that $h(\mathbf{c}) = 1$ if and only if for all pairs of minterms of the uncommon variables, $(\mathbf{s}, \mathbf{t}) \in B^n \times B^m$, $f(\mathbf{cs}) = g(\mathbf{ct}) = 1$. This function gives the set of common minterms dependent on the common fanin variables of the two nodes. We then define the *sharedness* between the two nodes s and t, $W(s,t)$, as the number of minterms $\mathbf{c} \in B^k$ such that $h(\mathbf{c}) = 1$. The idea is that a minterm $\mathbf{c}$ such that $h(\mathbf{c}) = 1$ is always sitting in the ON-sets of the both functions f and g for all possible input patterns of the Boolean space spanned by the uncommon variables s and t, and thus we employ a heuristic that the more sharedness two nodes have, the more common logic might be used to represent both functions. In the implementation, the function h is represented by a BDD, and the number of minterms can be counted in time linear in the number of nodes of the BDD.

In a more general case, sharedness is defined between a node s and an existing partially composed cluster C, where s is not a member of C. Since we want to see if there is common logic between s and some of the nodes in C, we define the sharedness $W(s, C)$ between a node s and a

cluster C as $W(s,C) = \max_{t \in C} W(s,t)$.

2.5.2 A Procedure for Composing a Clustered Boolean Network

The proposed procedure for clustering takes as input a Boolean network $\eta = (N, E)$ and returns a set of clusters. A pseudo-code for the clustering procedure is presented in Figure 2.5.

The clusters are formed one by one until all the nodes have been assigned to clusters. Each cluster is formed by adding one node at a time, starting from a seed node. The seed node chosen for each new cluster is the one with the maximum number of fanins among the nodes that have not been included in any cluster, since the node is likely to have high sharedness with other remaining nodes. This selection is done in Max_Fanin.

To avoid expensive calculations of sharedness for all candidate nodes t, filtering is employed. We use a first approximation to the sharedness for this purpose which is called the *usability*, U. The filtering essentially orders the node list by filtering all *bad* nodes to the end of the list.

The idea behind usability is that groups of nodes with a high degree of common fanins are likely to have high sharedness. Although good usability does not necessarily imply good sharedness, we see that bad usability does imply bad sharedness. Given a node t, the usability is defined against the seed node s of the cluster and is formally computed as:

$$U(s,t) = \frac{\sum_{r \in fanin(s) \cup fanin(t)} \frac{m(r,s,t)}{2}}{|fanin(s) \cup fanin(t)|}, \tag{2.3}$$

where $fanin(s)$ designates the set of fanin nodes of s, and $m(r,s,t)$ is 2 if r is a fanin of both s and t. Otherwise, $m(r,s,t)$ is 1.

Once the node list is ordered by usability, we then start filling the clusters in decreasing order of usability. A node t is included in cluster C if the sharedness $W(t,C)$ is greater than a user specified threshold. This process continues until all nodes have been accepted into a cluster. Note that it is also possible to have clusters with only one node in them.

Although the optimal size for a cluster is undetermined, we place an upper bound on the size of each cluster. This is a parameter set by the user.

Note that even if a node has high sharedness with a cluster, we cannot accept it if it creates a cycle in the clustered network. The acyclic check is done by Is_Legal. For this purpose, we keep track of the transitive fanin nodes and the transitive fanout nodes of each cluster composed so far. The subprocedure TFO_TFI updates the information whenever a new node is added to a cluster.

```
function Clustering(η = (N, E))
    Node_List ← N;
    i ← 0;
    while(Node_List ≠ φ){
        s ← Max_Fanin(Node_List);
        Cluster[i] ← s;
        Node_List ← Node_List - {s};
        TFO_TFI(i, s);
        Node_List ← Sort(Node_List, Usability(s));
        foreach(t ∈ Node_List){
            if(Cluster[i] is full) break;
            if(Sharedness(t, Cluster[i]) ≥ Threshold and Is_Legal(t, Cluster[i])){
                Cluster[i] ← Cluster[i] ∪{t};
                Node_List ← Node_List - {t};
                TFO_TFI(i, t);
            }
        }
        i ← i + 1;
    }
    return Cluster[0, ..., i − 1];
```

Figure 2.5: Procedure for Composing Clusters

Specifically, a node t is defined as a transitive fanin of a cluster C if t belongs to a cluster $\tilde{C}$ that is a transitive fanin cluster of C or there is a fanout node of t that is a transitive fanin node of C, where $\tilde{C}$ may be equal to C. Transitive fanout nodes of a cluster are similarly defined. Then for a given cluster C and a node s such that $s \notin C$, the inclusion of s in C creates a cycle if and only if there exists a fanout (or fanin, respectively) of s, say t, such that t is not in C and is a transitive fanin (respectively, transitive fanout) of C.

2.6 Experimental Results

The proposed procedure has been implemented in SIS [51], a logic synthesis system for sequential circuits. The procedure takes as input a Boolean network, composes a clustered Boolean network, and computes a set of permissible functions for each cluster. The set of functions is minimized for each cluster using a heuristic minimizer for Boolean relations, described in Chapter 3. The minimizer finds a sum-of-products expression with a minimal number of product terms among the functions in the set. The expression is transformed into a multi-level form, where the set of operations used for this transformation can be externally specified. The resulting multi-level representation is then compared with the original implementation of the cluster, and if the number of the literals in the factored form for the new representation is less than for the original, the original is replaced.

Experiments were performed on a number of benchmark examples. Since the proposed procedure is unlikely to be effective for the circuits with little flexibility, we mainly applied the procedure to the combinational logic parts of sequential circuits, where the set of unreachable states of the corresponding finite state machines were extracted, and were used as don't cares.

We first compared the results between the method of computing compatible sets of permissible functions and the one for computing maximum sets. As described in Section 2.4.4, the proposed procedure can be used to compute both types of permissible functions. For the case of compatible sets, we apply the procedure given in Section 2.4.2 over all the clusters first, so that maximally compatible sets of permissible functions are obtained. We then minimize each set of functions for one cluster at a time. This type of computation is referred to as type C hereafter. For the case of maximum sets, on the other hand, we apply the same procedure for each cluster, but immediately perform the minimization before processing the next cluster. When the next cluster is processed, the functionality of each of the preceding clusters is fixed to the result of the minimization for the cluster, so that the maximum set of permissible functions is obtained for the cluster being

Name	Initial			C3	M3	C2	M2	C1	M1
	In	Out	Lit.						
s27	4	1	12	12	12	12	12	12	12
s298	3	6	244	98	93	111	108	100	100
s344	9	11	269	141	141	147	147	146	146
s349	9	11	273	141	141	147	147	146	146
s382	3	6	306	159	159	160	160	156	156
s526	3	6	445	217	219	214	213	194	191
s820	18	19	757	462	468	462	467	447	423
s832	18	19	769	472	469	468	461	448	427

Table 2.3: Comparison between Compatible Sets and Maximum Sets

processed. We refer to this type of computation as type M. As mentioned in Section 2.4.4, the type M computation has more flexibility in a local sense, since the relation computed for a cluster has only to agree with the current minimized implementation of the preceding clusters, whereas the relation given by the type C computation needs to agree with all possible functions that may result in the final implementation of the preceding clusters. However, local flexibility does not imply global effectiveness, and thus it is not clear which method leads to better results in total.

Table 2.3 shows results for iscas-89 benchmarks. For each example, we performed both types of computations and compared the results, where constant nodes and nodes with single fanouts were removed from the initial Boolean network in advance. In either type of computation, the threshold in sharedness was set to 1, i.e. two nodes are considered to have good sharedness if there exists at least one minterm in common in the Boolean space spanned by the common fanins of the nodes. In Table 2.3, the column **Initial** shows the size of the initial Boolean network, where the columns In, Out, and Lit. designate the number of primary inputs, primary outputs, and the number of literal counts in factored form, respectively. The columns **C3**, **C2**, and **C1** show the number of literal counts of the resulting network obtained by the type C computation, where the attached number k, $k = 1, 2, 3$, is the upper bound on the number of nodes in a single cluster used when a clustered Boolean network was composed. Similarly, the columns **M3**, **M2**, and **M1** represent the results of the type M computation. According to the results, the difference was not significant between these two types of computations, although the type M was slightly better for most of the examples; the least literal counts were achieved by the type M computation for all the examples tried. The computational time is not given in the table, but the type M was usually faster by 10 to

20 percent than the type C computation with the same upper bound on the number of nodes in a single cluster.

We also conducted another type of experiment, where the effectiveness of the proposed procedure was examined in a context common in practice for optimizing multi-level combinational logic circuits, i.e. local optimization techniques, such as the one discussed in this chapter, are applied in conjunction with global optimization techniques such as a factorization or a decomposition [9]. The objective of the experiment is to see the effectiveness of optimizing a set of nodes at a time in a clustered Boolean network. Therefore, we compared the results with a method implemented in SIS for computing sets of permissible functions for Boolean networks, where each node has exactly one output and an optimization is made for one node at a time. The method was proposed in [50], and is referred to as `full_simplify`, its command name in SIS. The script of the optimization procedures applied for each example is shown in Figure 2.6, where `resub -a` is an algebraic resubstitution, `fx` is a procedure [58] for extracting kernels, while `sweep` and `eliminate -1` remove constant nodes and nodes with single fanouts. The command `br_simplify` is the proposed procedure, where the option `-M` specifies the upper bound k on the number of nodes in a single cluster. The threshold in the sharedness was set to 1, and the type M computation was used for computing a set of permissible functions. For the first `br_simplify` in the script, the minimized sum-of-products expression returned by the relation minimizer for each cluster was transformed into a multi-level form by applying `fx`, while it was collapsed into a one level form in the second `br_simplify`. We tried the same script by varying the upper bound on the number of nodes in a cluster from 1 to 3 in the first `br_simplify`. These results were compared with `full_simplify` by replacing both `br_simplify` commands in the script of Figure 2.6 with `full_simplify`.

Table 2.4 shows the results. The column **Initial** shows the size of the initial circuit, and the column **full_simplify** shows the results using `full_simplify` in the script. The columns **BR 3**, **BR 2**, and **BR 1** show the results of `br_simplify`, where the attached number indicates the value of k in the script. The CPU time (seconds) for all the procedures of the script was measured on a DECstation 5000/240, which includes the time for computing unreachable states of the corresponding finite state machines.

The results show that marginally better results are obtained than with a procedure of minimizing one node at a time. During the experiments for **BR 3** and **BR 2**, we frequently encountered the case where the Boolean relation computed for a cluster cannot be reduced to an incompletely specified function with don't cares, especially for the clusters close to the primary inputs. Thus a Boolean relation minimizer is indeed useful.

```
sweep; eliminate -1
br_simplify -M k
resub -a
eliminate -1; sweep

fx; resub -a; sweep
br_simplify -M 1
sweep; eliminate -1
```

Figure 2.6: Script used for Table 2.4

	Initial			full_simplify		BR 3		BR 2		BR 1	
Name	In	Out	Lit.	Lit.	Time	Lit.	Time	Lit.	Time	Lit.	Time
s27	4	1	12	12	0.2	12	0.2	12	0.2	12	0.2
s298	3	6	244	91	2.1	78	3.0	90	3.6	89	2.6
s344	9	11	269	142	9.8	140	19.0	142	18.5	142	15.9
s349	9	11	273	142	9.8	140	18.8	142	18.5	142	16.5
s382	3	6	306	127	7.1	126	12.3	133	8.9	135	7.5
s526	3	6	445	144	7.0	133	68.7	142	20.2	124	12.8
s820	18	19	757	335	36.1	309	83.1	289	38.2	319	29.1
s832	18	19	769	340	14.5	314	93.3	309	50.2	300	38.2

Table 2.4: Comparison with `full_simplify`

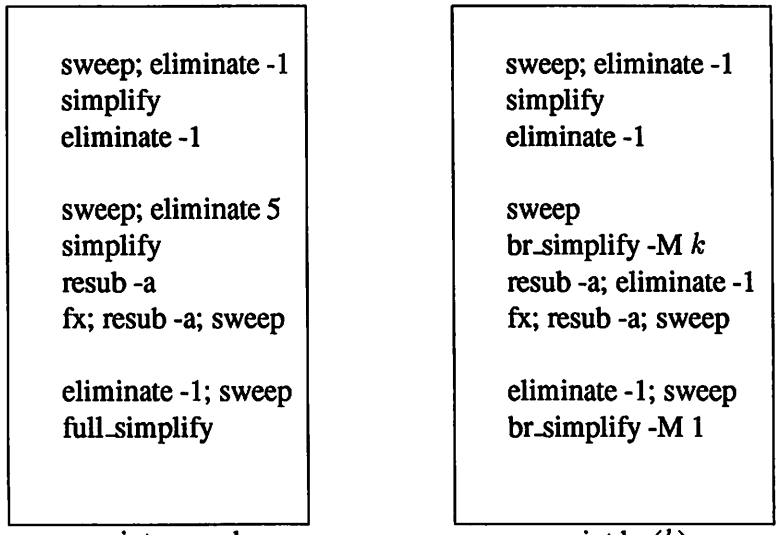

Figure 2.7: Scripts used for Table 2.5

We also made a comparison against *script.rugged*, a standard script available in SIS using the state-of-the-art optimization techniques. We used a script similar to *script.rugged*, but `br_simplify` was used. The *script.rugged* and the one we used are shown in Figure 2.7. The script we used does not invoke `eliminate 5` after the second `sweep`. The command `eliminate 5` clusters nodes of a network but uses a different criterion from the one described in Section 2.5. It also restricts itself so that the resulting cluster has a single output. Thus instead of using this clustering technique, we directly applied our clustering procedure, given in Section 2.5, which is implemented in `br_simplify`.

Table 2.5 summarizes the results. The column **SIS 1.1** shows the results obtained by *script.rugged*. The rest of the columns show those obtained by *script.br (k)*, where the number indicates the upper bound on the number of nodes in a cluster. The results derive the same observation as the previous experiments that *script.br* is slightly better than *script.rugged* for most of the examples. In fact, s832 was the only one among all the examples tried where *script.rugged* led to a better result.

	SIS 1.1 (rugged)		script.br (3)		script.br (2)		script.br (1)	
Name	Lit.	Time	Lit.	Time	Lit.	Time	Lit.	Time
s27	12	0.1	12	0.2	12	0.2	12	0.2
s298	99	1.8	83	3.9	85	3.3	91	2.9
s344	143	6.2	142	25.2	141	23.7	142	21.6
s349	143	6.2	139	24.0	141	24.4	142	21.6
s382	154	6.9	133	9.1	134	10.4	134	8.2
s526	147	7.1	147	20.4	140	11.8	140	9.7
s820	297	58.7	291	65.6	304	75.8	274	39.5
s832	286	26.4	298	83.1	295	41.6	291	41.6

Table 2.5: Comparison with *script.rugged*

2.7 Concluding Remarks

In this chapter, we consider the problem of finding a set of permissible functions for a system of interacting components implementing a combinational logic behavior. We considered two types of permissible functions; the maximum set of permissible functions defined for a single component, and maximally compatible sets of permissible functions defined for a set of components.

We first described how a set of multi-output functions can be represented by using a relation, and presented a procedure for computing maximally compatible sets of permissible functions, where a Boolean relation is computed for each cluster of a given clustered Boolean network. We also showed the relationship between the two types of permissible functions, by demonstrating how the proposed procedure can be modified so that the maximum set of permissible functions is computed instead.

The proposed procedure has an application in logic optimization of combinational logic circuits. It is an extension of a technique called node optimization, in which a set of permissible functions is computed for each node of a Boolean network, where a node implements a Boolean function with a single output. Conventionally, it has been considered as a limitation that each node must implement a function with only one output, since by taking into account multiple outputs simultaneously, one may be able to use a common logic to represent multiple functions. Therefore, we implemented our procedure and compared with state-of-the-art techniques developed for single-output case. For this purpose, we also developed and implemented a heuristic for composing a clustered Boolean network from a given Boolean network. The experimental results demonstrate

that the concurrent minimization over multiple outputs can lead marginally better results than those achieved by conventional node optimization techniques.

There is room for improvements in the clustering algorithms. The algorithm we used is based on sharedness, a measure of the common logic shared among a set of nodes. However, the objective in the optimization phase is to find as many functions as possible for a cluster that can replace the original function associated with it. We observed that sharedness is not always a sufficiently good criterion for forming clusters with many functions. In fact, since we force an upper bound on the number of nodes in a cluster, clusters formed by the algorithm sometimes have a serially cascaded set of nodes, with only one output. Since our objective is to see the effectiveness of concurrently minimizing more than one output, it is not meaningful to group such a set of nodes together. This problem must be addressed in order to gain the practical effectiveness of node optimization for multiple outputs.

Chapter 3

Minimization of Multiple-Valued Relations

3.1 Introduction

In Chapter 2, we described how to find a set of functions that can be realized in a component of a system implementing a combinational logic behavior, and showed that such a set can be represented by a Boolean relation. In this chapter, we consider the problem of optimizing relations. Namely, for a given set of functions represented by a relation, we find a least-cost representation over the functions in the set. This problem is generally referred to as *logic minimization.*

Research in logic minimization has been active over the past 40 years. Initial research was directed towards developing techniques to produce an optimum sum-of-products expression of a Boolean function under don't care conditions[34], and has evolved toward heuristic approaches for designing programmable logic arrays (PLA's) [8, 25]. A Boolean function with don't care conditions is called an incompletely specified Boolean function, and the problem above is sometimes referred to as the two-level minimization of incompletely specified Boolean functions. More recently, as seen in Chapter 2, it was shown that don't cares of the traditional kind are inadequate to capture the complete freedom for optimizing multiple output functions, and a theory of Boolean relations was introduced [11].

In parallel with this activity has been the minimization of multiple-valued functions [31, 47], in which variables can assume more than two discrete values. The significance of this problem is in its applications in areas such as PLA optimization [45] and state assignment for finite

state machines [17].

In this chapter, we assume that a given relation is in general a multiple-valued relation, in the sense that the input variables can take multiple (more than two) values. Although we assume that the output variables are still binary, this is not a restriction since we can encode a multiple-valued output using binary output variables. This aspect will be detailed later in Section 3.3. Also, a Boolean relation is a special case of the type of relations considered here, and thus our problem subsumes the problem of minimizing Boolean relations.

Multiple-valued relations arise in many contexts [6, 11, 32], besides the local optimizations of multi-level combinational logic circuits considered in Chapter 2. For example, the behavior of a completely specified deterministic finite state machine is given by a function $F : I \times S \times S \times O \rightarrow B$ such that $F(i, p, n, o) = 1$ if and only if the input i and the present state p causes the machine to evolve to the next state n and produce the output o. F is a multiple-valued relation with the input set $I \times S$ and the output set $S \times O$. For a given initial state, a set of equivalent states can be computed as a function $E : S \times S \rightarrow B$ such that $E(n, \tilde{n}) = 1$ if and only if n and $\tilde{n}$ are equivalent [27, 38]. Since a state can be mapped to any of the equivalent states of the next state, we have the possibility of implementing a more compact machine using the equivalent states. Namely, our objective is to find a least cost machine given by the function $\tilde{F} : I \times S \times S \times O \rightarrow B$ such that $\tilde{F}(i, p, n, o) = 1$ if and only if either $F(i, p, n, o) = 1$ or there exists a state $\tilde{n}$ for which $F(i, p, \tilde{n}, o) = 1$ and $E(n, \tilde{n}) = 1$. $\tilde{F}$ provides the complete family of finite state machines equivalent to the original machine under the equivalent states.

For the cost function, we use the number of product terms required in a sum-of-products expression representing a function in the set given by the relation. Conventionally, a function in the set represented by a given relation is called a *compatible function* [11]. Then the problem is that for a given multiple-valued relation, find a sum-of-products expression with the minimum number of product terms for a function compatible with the relation. Note that in terms of the minimization problem, a relation can be deemed as a generalization of an incompletely specified function, and thus if the set of functions represented by a given relation can be represented by using a function with don't care conditions, then the minimization problem is reduced to the conventional two-level minimization of incompletely specified functions.

Somenzi and Brayton proposed and implemented an exact minimization procedure for Boolean relations, with which an optimum sum-of-products representation is obtained[53]. However, since the method is exact, it is expensive both in CPU time and memory space, so that only small examples can be handled.

Our focus in this chapter is on heuristic minimizations. Ghosh *et al.* [22] proposed and implemented an approach for Boolean relations which makes use of test pattern generation techniques and heuristically finds a sum-of-products expression. The method is similar to two-level minimizers for Boolean functions (e.g. [8]) in the sense that procedures analogous to expand, irredundant, and reduce are repeatedly applied as long as the cost decreases. However, unlike the most effective two-level minimizers that consider multiple variables to be expanded or reduced simultaneously, the method is greedy and only one variable is examined at a time. Thus the minimizer of [22] is more likely to get stuck at a bad solution. This drawback is fatal for extending the method since the simultaneous expansion of multiple variables implies the use of multiple faults, which could be very expensive to detect with ATPG techniques. Furthermore, the method is a direct application of ATPG methodology to this problem and little new theoretical analysis is provided. For example, the contrast between the properties of relations and those of ordinary functions may be useful in the minimization process.

We propose a heuristic procedure for the minimization problem of multiple-valued relations, based on a paradigm of the more advanced two-level minimization techniques for Boolean functions. We present some special properties associated with relations not found in functions. These properties must be carefully accounted for while implementing a procedure that is effective in achieving high quality results. These algorithms are implemented in a program called GYOCRO[1], and provide experimental evidence of their effectiveness.

This chapter is organized as follows. In Section 3.2, terminology is defined and a brief review of multiple-valued relations is provided. Section 3.3 addresses some of the questions posed for multiple-valued relations, such as how multi-valued outputs can be handled using binary outputs. Section 3.4 describes how to identify whether a given relation is an incompletely specified function, as well as a procedure that extracts the care and don't care sets from the relation if it is a function. Section 3.5 presents the minimization procedure employed in GYOCRO in which technical details are described for each sub-procedure along with supporting theoretical analysis. Experimental results of the proposed method are presented in Section 3.6, where some potential modifications of the algorithms are also discussed. Section 3.7 summarizes the chapter with some concluding remarks.

[1]GYOCRO is a Japanese tea and although it is not strong like ESPRESSO, it has good taste.

3.2 Preliminaries

We describe the relationship among functions and relations, and see when a procedure is needed for minimizing relations directly.

3.2.1 Terminology

We follow references [8, 11, 45] for most of the terminology used throughout this chapter.

Definition: Multiple-Valued Relation

A **multiple-valued relation** R is a subset of $D \times B^m$. D is called the **input set** of R and is the Cartesian product of n sets $D_1 \times \cdots \times D_n$, where $D_i = \{0, \ldots, P_i - 1\}$ and P_i is a positive integer. D_i provides the set of values that the i-th variable of D can assume. B^m designates a Boolean space spanned by m variables, each of which can assume either 0 or 1. B^m is called the **output set** of R. If P_i is 2 for all i's, then R is called a **Boolean relation**. The variables of the input set and the output set are called the **input variables** and the **output variables** respectively. R is **well-defined** if for every $\mathbf{x} \in D$, there exists $\mathbf{y} \in B^m$ such that $(\mathbf{x}, \mathbf{y}) \in R$.

We represent a relation R by its characteristic function $R : D \times B^m \rightarrow B$ such that $R(\mathbf{x}, \mathbf{y}) = 1$ if and only if $(\mathbf{x}, \mathbf{y}) \in R$. In the implementation, we represent a characteristic function by using an MDD (Multi-valued Decision Diagram) [54]. An MDD is a data structure to represent a function with multiple-valued input variables and a single binary output, which employs a BDD [12] as the internal data structure. In the sequel, we make no distinction between a relation and its characteristic function.

Definition: Incompletely Specified Function

A single-output function $f : D \rightarrow B$ is said to be **incompletely specified**, if there exists a non-null subset $\delta \subseteq D$ for which the output value of f is not specified. An element of δ is said to be an unspecified input minterm, or a **don't care** minterm. A multiple-output function $f : D \rightarrow B^m$ is said to be **incompletely specified** if there exists at least one output for which the corresponding function is incompletely specified.

A function that is not incompletely specified is said to be *completely specified.* Clearly, a completely specified function is a special case of an incompletely specified function. Throughout this paper, whenever we define a function, we assume that it is completely specified, unless otherwise mentioned.

It is interpreted that the output value of an incompletely specified function $f : D \rightarrow B$ for a don't care minterm $\mathbf{x}$ may be either 0 or 1. In this sense, we can regard that an incompletely specified function represents a set of completely specified functions. The set $\delta \subseteq D$ of don't care minterms is called the *don't care set* for f. The set of minterms of D that are not don't care minterms is called the *care set*. Among the care set, the set of minterms $\mathbf{x}$ for which $f(\mathbf{x}) = 1$ is called the *on-set* for f, while those with $f(\mathbf{x}) = 0$ is called the *off-set* for f.

An incompletely specified function is a special case of a relation, in the sense that for a given incompletely specified function $f : D \rightarrow B^m$, a relation $F \subseteq D \times B^m$ can be defined so that $(\mathbf{x}, \mathbf{y}) \in F$ if and only if for each output j, the value of the j-th output in $\mathbf{y}$ is equal to $f^{(j)}(\mathbf{x})$, unless $\mathbf{x}$ is a don't care minterm for the output, where $f^{(j)}$ designates the j-th output function of f. We may refer to the relation F as the *characteristic function* of f.

Definition: The Image of a Multiple-Valued Relation

For a given relation R and a subset $A \subseteq D$, the **image** of A by R is a set of minterms $\mathbf{y} \in B^m$ for which there exists a minterm $\mathbf{x} \in A$ such that $(\mathbf{x}, \mathbf{y}) \in R$, i.e. $\{\mathbf{y} \mid \exists \mathbf{x} \in A : (\mathbf{x}, \mathbf{y}) \in R\}$. The image is denoted by $r(A)$. $r(A)$ may be empty.

Definition: Compatibility of a Multiple-Valued Function

For a given relation $R \subseteq D \times B^m$, a multiple-valued function $f : D \rightarrow B^m$ is **compatible** with R, denoted by $f \prec R$, if for every minterm $\mathbf{x} \in D$, $f(\mathbf{x}) \in r(\mathbf{x})$. Otherwise f is incompatible with R. Clearly, $f \prec R$ exists if and only if R is well-defined.

Definition: Literal and Product Term

For the i-th variable x_i of D, a **literal** of x_i is the characteristic function of a subset S_i of D_i, and is denoted by $x_i^{S_i}$. S_i may be empty. A **product term** p defined in D is a Boolean product of literals of all the variables of D. Thus p is the characteristic function of a subset of D.

For the j-th output function $f^{(j)}$ of $f : D \rightarrow B^m$, a *sum-of-products expression* (or simply an *expression*) of $f^{(j)}$ is a union of product terms such that the resulting characteristic function is equivalent to $f^{(j)}$. A sum-of-products expression of f is a set of sum-of-products expressions for all the output functions.

Definition: Cube and Representation

For a sum-of-products expression of a function $f : D \rightarrow B^m$, a **cube** is a product

term p of the expression specified as a row vector with two parts, $c = [I(c)|O(c)]$, where $I(c) = [I(c)_1, \ldots, I(c)_n]$ and $O(c) = [O(c)_1, \ldots, O(c)_m]$. $I(c)$ and $O(c)$ are called the input part and the output part of c respectively. The i-th component of $I(c)$ represents a set of values contained in the i-th literal of p, and consists of P_i binary bits. Each bit is called a **part**. The k-th part of $I(c)_i$, $k \in \{0, \ldots, P_i - 1\}$, is 1 if the i-th literal of p contains the value k. It is 0 otherwise. For the output part, $O(c)_j = 0$ if p is not present in the expression of $f^{(j)}$. Otherwise, $O(c)_j = 1$. We denote by $M(c)$ the set of minterms of D contained in c. A set of cubes is called a **representation**.

For two cubes c and d, c *contains* d, or c *covers* d, if c has 1 for every part that d has 1. In addition, c *strictly contains* d if they are not equal. For a given minterm $\mathbf{x} \in D$, we say $\mathbf{x}$ is *covered* by a representation, if $\mathbf{x}$ is contained in some of the cubes of the representation.

Throughout the chapter, we show examples of representations, in which all the inputs are binary variables. For the sake of simplicity, we represent the input part of a cube c as an n-tuple $[I(c)_1, \ldots, I(c)_n]$ such that $I(c)_i$ takes 0 if the i-th literal of p takes a value 0, 1 if the i-th literal of p takes 1, and 2 if the literal takes both 0 and 1.

For a given representation $\mathcal{F}$, a function $f : D \rightarrow B^m$ is uniquely defined, where an expression of f is given by $\mathcal{F}$. Thus we say that a representation $\mathcal{F}$ is *compatible* with a relation R if the corresponding function f is compatible with R. Similarly, the image of $A \subseteq D$ by the representation $\mathcal{F}$ is the image of A by a relation F given by $F = \{(\mathbf{x}, \mathbf{y}) \in D \times B^m \mid \mathbf{y} = f(\mathbf{x})\}$.

Definition: Candidate Prime (c-prime)

For a given relation R, a cube c is a **candidate prime** (or a c-prime) if there exists a function compatible with R in which c is a prime implicant [8].

Definition: Relatively Prime Cube

For a given relation R and a compatible representation $\mathcal{F}$, a cube $c \in \mathcal{F}$ is prime relative to $\mathcal{F}$ (or **relatively prime** in $\mathcal{F}$) if for any cube $\tilde{c}$ which strictly contains c, a replacement of c with $\tilde{c}$ in $\mathcal{F}$ results in an incompatible representation with R. A representation $\mathcal{F}$ is **relatively prime** if $\mathcal{F}$ is compatible with R and every cube of $\mathcal{F}$ is relatively prime in $\mathcal{F}$.

Note that if $c \in \mathcal{F}$ is relatively prime in $\mathcal{F}$ then c is a c-prime, but not the other way around. We distinguish the notion of primality between relations and ordinary functions, since in relations, the primality of a cube depends upon the other cubes of the representation in which the cube is present. This is not the case for functions.

Definition: Redundant Cube

For a given relation R and a compatible representation $\mathcal{F}$, a cube $c \in \mathcal{F}$ is **redundant** in $\mathcal{F}$ if removal of c from $\mathcal{F}$ maintains the compatibility of the representation $\mathcal{F} - \{c\}$ with R. Otherwise c is irredundant.

A representation $\mathcal{F}$ is said to be *irredundant* if $\mathcal{F}$ is compatible with R and there is no proper subset of $\mathcal{F}$ which is also compatible. Otherwise, $\mathcal{F}$ is *redundant*. Note that the irredundancy of a representation $\mathcal{F}$ implies the irredundancy of every cube of $\mathcal{F}$, but not the other way around. The following example illustrates the situation.

Example 3.2.1 *Consider a Boolean relation shown in Table 3.1. The relation has two input variables and two output variables, and the table means that a pair of input and output minterms* $(10, 11)$ *a member of the relation,* $(10, 00)$ *is another member of the relation, and so on. The representation* $\mathcal{F}$ *shown on the left hand side of the table is compatible with this relation, since it maps an input minterm* 10 *to an output minterm* 11, 00 *to* 11, *and the rest of the input minterms to* 00. *Every cube of* $\mathcal{F}$ *is irredundant, since the removal results in an incompatible representation. However, a proper subset* $\tilde{\mathcal{F}}$ *which consists only of* c_3 *is also compatible, and thus* $\mathcal{F}$ *is redundant.*

Representation $\mathcal{F}$			Relation R	
cube	Input	Output	$\mathbf{x} \in B^2$	$\mathbf{y} \in B^2$
c_1	10	01	10	11, 00
c_2	10	10	01	00
c_3	00	11	00	11
			11	00

Table 3.1: Example of Redundant Representation with Irredundant Cubes

3.2.2 Functions, Mappings, and Relations

Given a multiple-valued relation $R \subseteq D \times B^m$, a compatible function exists if and only if R is well-defined. By definition, R is well-defined if and only if for all $\mathbf{x} \in D$, there exists $\mathbf{y}$ such that $R(\mathbf{x}, \mathbf{y}) = 1$. The well-definedness can be easily checked by using MDD operations. Now, with this condition satisfied, R can be represented as a multiple-valued mapping $r : D \rightarrow B^m$ given by $r(\mathbf{x}) = \{\mathbf{y} \in B^m \mid (\mathbf{x}, \mathbf{y}) \in R\}$, where we define a mapping as one which defines at least one minterm of B^m for each minterm of D. In general, the mapping r is a one-to-many

mapping, and provides the complete family of functions compatible with R. If, in addition, $r(\mathbf{x})$ can be represented as a single cube for every $\mathbf{x} \in D$, then r can be expressed as an incompletely specified function[11]. Thus for each output of r, the set D can be divided into the On-set, the Off-set, and the Don't-care set. Once these three sets are obtained, the problem is reduced to the conventional minimization for incompletely specified functions . As we will see later, the minimization of functions is a simpler problem than the minimization of relations. Furthermore, a number of methods for function minimization exist (e.g. [10, 16, 25, 34, 45, 47]). Therefore, if this is the case, we invoke one of the conventional procedures for minimizing functions to obtain a minimal implementation of the given relation R. If, on the other hand, the mapping r cannot be expressed as an incompletely specified function, a procedure capable of handling the relation R directly is needed.

In Section 3.4, a procedure is presented that identifies whether r is an incompletely specified function, and if so extracts the care and don't care sets. In this way, logic minimization is handled in a uniform fashion; minimization of functions and minimization of true relations are viewed in parallel as equal sub-procedures in the minimization of relations. The entire minimization procedure of relations is illustrated in Figure 3.1.

3.2.3 Applications of Multiple-Valued Relations

One can easily imagine applications for minimizing relations, by considering those for minimizing functions. One such example is state assignment. Suppose a completely specified deterministic finite state machine is given by the characteristic function $F : I \times S \times S \times O \rightarrow B$ such that $F(i, p, n, o) = 1$ if and only if the next state n and the output o are asserted by the input i and the current state p. F is the characteristic function of a multiple-valued relation with the input set $S \times I$ and the output set $S \times O$. The symbolic variable for S in the output set are encoded in a way described in Section 3.3. In fact, we know that F can be expressed in a form of an ordinary function since the given machine is deterministic and is completely specified. However, suppose we are given a set of equivalent states by a function $E : S \times S \rightarrow B$ such that $E(n, \tilde{n}) = 1$ if and only if states n and $\tilde{n}$ are equivalent. Also given is a set of invalid states by a function $T : S \rightarrow B$ such that $T(p) = 1$ if and only if p is a state not reachable from some initial set of states. Then a set of machines equivalent to F under the equivalent states and the invalid states is given by a function $\tilde{F} : I \times S \times S \times O \rightarrow B$ such that $\tilde{F}(i, p, n, o) = 1$ if and only if $F(i, p, n, o) = 1$ or $T(p) = 1$, or there exists a state $\tilde{n}$ for which $F(i, p, \tilde{n}, o) = 1$ and $E(n, \tilde{n}) = 1$.

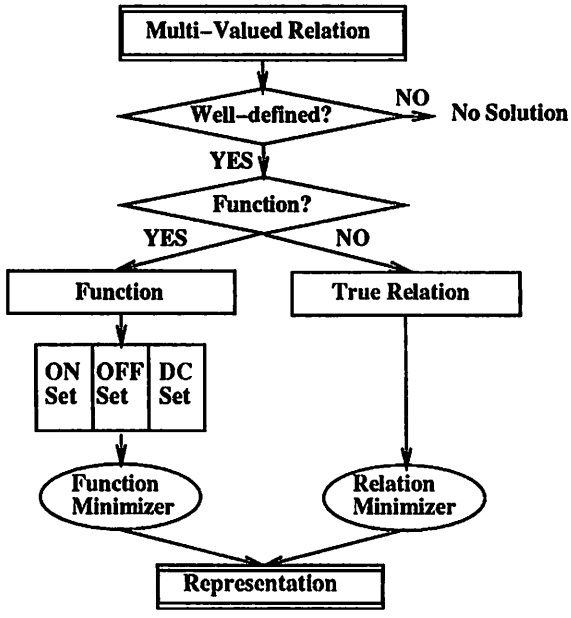

Figure 3.1: The Minimization of Relations

In order to obtain a least cost implementation of a machine equivalent to the original one, we first find a least cost machine at the symbolic level. Namely, our objective is to minimize the multiple-valued relation $\tilde{F}$. If the number of symbolic product terms is the cost function used, then the problem is reduced to the relation minimization focused in this chapter. Other applications of multiple-valued as well as Boolean relations can be found in [11, 32].

Example 3.2.2 *The case where the use of equivalent states results in a representation with less cost is illustrated in the following example. Suppose that a completely specified deterministic finite state machine is given as shown in Figure 3.2. Each circle designates a state of the machine and the label i/o associated with each arc implies that the input i and the state associated with the tail of the arc causes the machine to produce the state shown on the head of the arc and the output o. The minimized representation for this machine is shown in Table 3.2-(a). If, in addition, we know that the state s_3 and s_4 are equivalent, then we can further minimize the machine and obtain a better representation shown in Table 3.2-(b).*

Other applications, where relations arise, can be illustrated as follows. For the sake of

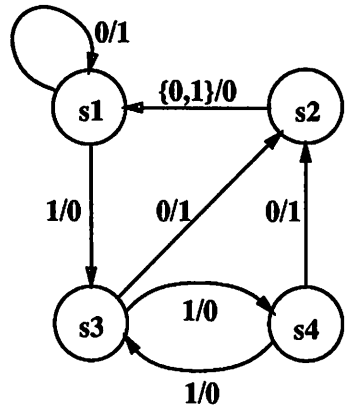

Figure 3.2: Completely Specified Finite State Machine

I	S	S	O
1	s_1, s_4	s_3	0
–	s_2	s_1	0
1	s_3	s_4	0
0	s_1	s_1	1
0	s_3, s_4	s_2	1

(a)

I	S	S	O
1	s_1, s_3, s_4	s_3	0
–	s_2	s_1	0
0	s_1	s_1	1
0	s_3, s_4	s_2	1

(b)

Table 3.2: Minimized Representations of the Finite State Machine

simplicity, we consider only the binary-valued case. Consider the situation illustrated in Figure 3.3. Let $g : B^p \rightarrow B^q$ be a Boolean function and $h : B^n \rightarrow B^q$ be a Boolean mapping, where the image of a minterm $\mathbf{x} \in B^n$ by h, $h(\mathbf{x})$, may consist of multiple minterms. Let B^r be the subspace spanned by the variables common to both B^n and B^p. B^r may be empty. Denote the orthocomplement of B^r in B^p by B^m, where $m = p - r$. Consider the problem of finding a function $f : B^n \rightarrow B^m$ such that

$$\forall \mathbf{x} \in B^n : g(\mathbf{v}f(\mathbf{x})) \in h(\mathbf{x}), \tag{3.1}$$

where $\mathbf{v}$ is the projection of $\mathbf{x}$ from B^n to B^r. Our objective is to find an implementation f of least cost with the property (3.1), if such a function exists. Namely, considering a relation R given by $R = \{(\mathbf{x}, \mathbf{y}) \in B^n \times B^m \mid g(\mathbf{vy}) \in h(\mathbf{x})\}$, our problem is to find a least cost implementation

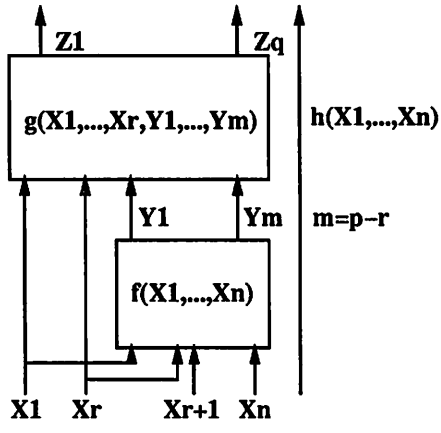

Figure 3.3: Structure where Boolean Relation Arises

compatible with R. The relation R is computed as follows. Let $H : B^n \times B^q \rightarrow B$ be a relation given by $\{(\mathbf{x}, \mathbf{z}) \mid \mathbf{z} \in h(\mathbf{x})\}$. Similarly, let $G : B^p \times B^q \rightarrow B$ be a relation such that $G(\mathbf{vy}, \mathbf{z}) = 1$ if and only if $g(\mathbf{vy}) = \mathbf{z}$. Then $R(\mathbf{x}, \mathbf{y}) = 1$ if and only if there exists $\mathbf{z} \in B^q$ such that $H(\mathbf{x}, \mathbf{z}) = 1$ and $G(\mathbf{vy}, \mathbf{z}) = 1$. The actual computation of R is done on MDD's fairly efficiently for many cases.

This problem naturally arises in several applications. One context is the combinational logic optimization of multiple-output components described in Chapter 2. Another is the rectification problem concerned with "engineering changes", where one has implemented a function g with a highly optimized layout, only to encounter a specification change such that the correct functionality must be h. One possibility for rectifying this situation is to build a block of prelogic which sits between the inputs and the circuit already built. The equation (3.1) gives the condition for the function f of the attached block so that the resulting circuit is functionally equivalent to h. A detailed discussion of the rectification problem can be found in [60].

3.3 Questions on Multiple-Valued Relations

A relation considered in this chapter is one with binary output variables. In this section, we consider the following two questions;

1. whether a relation with multiple-valued outputs can be handled with binary-output relations,

2. whether the minimization problem for a relation can be transformed into the problem of minimizing an incompletely specified function with multiple-valued inputs and a single binary output.

The posing of the second question is motivated by the fact that the answer is yes if the given relation is in fact an incompletely specified function.

3.3.1 Representations of Multiple-Valued Outputs

There are several contexts where the problem is formulated as the minimization of a relation with multiple-valued outputs. We discuss how such a relation is handled with a multiple-valued relation defined in this chapter, i.e. one with binary-valued outputs. Specifically, we describe how to represent a multiple-valued output in terms of binary outputs using three encoding schemes known as 1-*hot encoding*, 0-*hot encoding*, and *log-based encoding* respectively.

Consider a relation $\Gamma \subseteq D \times E$, where E is the Cartesian product of t sets $E_1 \times \cdots \times E_t$ where E_j consists of L_j integers, i.e. $E_j = \{1, \ldots, L_j\}$. An encoding is the process of assigning a set of binary variables for each multiple-valued variable σ_j such that (1) each value of E_j is associated with a subset of the Boolean space spanned by the binary variables and (2) for each value the subset defined by the encoding is disjoint with that for any other value of E_j.

The 1-hot encoding is an encoding scheme in which each multiple-valued variable σ_j is represented by L_j binary variables $\{y_1^{(j)}, \ldots, y_{L_j}^{(j)}\}$ such that $\sigma_j = k \in E_j$ if and only if $y_k^{(j)} = 1$ and $y_i^{(j)} = 0$ for $i \neq k$. This scheme requires $\sum_{j=1}^{t} L_j$ variables to represent all the variables of E. The 0-hot encoding is the same as the 1-hot encoding in which the meaning of 1 and 0 for each encoded binary variable is switched.

Example 3.3.1 *Suppose that Table 3.3 is a specification of a relation $\Gamma \subseteq B^2 \times E$ with two binary inputs x_1, x_2 and two multiple-valued outputs σ_1 and σ_2, where both σ_1 and σ_2 can take three values. Then the transformed relation $R \subseteq B^2 \times B^6$ with the 1-hot encoding is shown in Table 3.3.*

The log-based encoding represents a multiple-valued variable with L_j values using p_j binary variables, where p_j is the smallest integer no less than $\log_2 L_j$. Each value of E_j is represented as a product term of the encoding variables such that two product terms corresponding to two different values are disjoint. The product term for each value may correspond to a set of

$\Gamma \subseteq B^2 \times E$		$R \subseteq B^2 \times B^6$	
x_1x_2	(σ_1, σ_2)	x_1x_2	$(y_1^{(1)}, y_2^{(1)}, y_3^{(1)}, y_1^{(2)}y_2^{(2)}, y_3^{(2)})$
11	{(3, 2), (2, 1)}	11	{(0, 0, 1, 0, 1, 0), (0, 1, 0, 1, 0, 0)}
10	{(2, 3)}	10	{(0, 1, 0, 0, 0, 1)}
01	{(1, 2), (1, 3)}	01	{(1, 0, 0, 0, 1, 0), (1, 0, 0, 0, 0, 1)}
00	{(2, 1)}	00	{(0, 1, 0, 1, 0, 0)}

Table 3.3: The 1-Hot Encoding of a Relation with Multiple-Valued Outputs

minterms of the encoding variables, rather than a single minterm. For some applications, it is useful to define the encodings so that the Boolean union of the product terms over all the values of E_j is the universe of the Boolean space spanned by the encoding variables [54]. A minterm σ of E corresponds to the Boolean product of the product terms for the values of the variables of E in σ.

Example 3.3.2 *Consider the same relation Γ used in Example 3.3.1. Since both σ_1 and σ_2 can take three values, each variable is represented by two binary variables. Let $y_1^{(j)}$ and $y_2^{(j)}$ be the encoding variables for σ_j. Suppose that 1, 2, and 3 are represented as $\overline{y}_1^{(j)}\overline{y}_2^{(j)}$, $\overline{y}_1^{(j)}y_2^{(j)}$, and $y_1^{(j)}$ respectively, then the transformed relation $R \subseteq B^2 \times B^4$ is obtained as shown in Table 3.4.*

$\Gamma \subseteq B^2 \times E$		$R \subseteq B^2 \times B^4$	
x_1x_2	(σ_1, σ_2)	x_1x_2	$(y_1^{(1)}, y_2^{(1)}, y_1^{(2)}, y_2^{(2)})$
11	{(3, 2), (2, 1)}	11	{(1, 1, 0, 1), (1, 0, 0, 1), (0, 1, 0, 0)}
10	{(2, 3)}	10	{(0, 1, 1, 0), (0, 1, 1, 1)}
01	{(1, 2), (1, 3)}	01	{(0, 0, 0, 1), (0, 0, 1, 0), (0, 0, 1, 1)}
00	{(2, 1)}	00	{(0, 1, 0, 0)}

Table 3.4: The Log-Based Encoding of a Relation with Multiple-Valued Outputs

3.3.2 Transformation of Multiple Outputs to a Multiple-Valued Input

It is shown in [46] that the two-level minimization problem for an incompletely specified function with multiple-valued inputs and binary-valued outputs can be equivalently handled as the problem of minimizing an incompletely specified function with a single binary output by treating the output part of the original function as 1-hot encoded variables for a single multiple-valued variable. The newly defined incompletely specified function is conceptually the characteristic function of the

set of pairs of minterms of the inputs and the multiple-valued output $(\mathbf{x}, \sigma)$ such that $\mathbf{x}$ is mapped to σ by the original function, but the characteristic function explicitly uses don't cares.

We are interested in whether this equivalency holds for multiple-valued relations with binary outputs. Specifically, the question is whether there exists an encoding scheme with which the output part of the relation can be treated as a single multiple-valued variable such that there exists an incompletely specified function with a single binary output with the inputs consisting of the original inputs and the newly defined multiple-valued variable for which the two-level minimization problem is equivalent to the problem of minimizing the original relation. As with the minimization of incompletely specified functions, we consider whether the characteristic function can be well formulated for some encoding schemes. We show that the encoding schemes described in the previous section, i.e. the 1-hot encoding, the 0-hot encoding, and the log-based encoding, do not fall into this category.

Given a relation $R \subseteq D \times B^m$ and an encoding scheme, our objective is to prove or disprove that there is a single output function that has an additional multiple-valued input. The new variable is defined by regarding that the encoding of the variable with the given encoding scheme results in the output part of R. Our requirement is, as with ordinary functions, that the two level minimization of the resulting single function leads to a result that when interpreted correctly is a minimum of the original minimization problem.

Suppose that we employ the 1-hot encoding. Let Y be the multiple-valued variable for the outputs. Y can take m values, $E = \{1, \ldots, m\}$, and we denote a value of E simply by j, where $1 \leq j \leq m$. A minterm of B^m corresponds to a set of values of E, i.e. a literal of Y. Thus the relation R maps a minterm $\mathbf{x} \in D$ to a set of literals of Y. Denote the set of literals by $r(\mathbf{x})$. Recall that in minimizing R, we can choose, for each $\mathbf{x} \in D$, any minterm $\mathbf{y} \in B^m$ such that $(\mathbf{x}, \mathbf{y}) \in R$. Therefore, in terms of Y, we can choose any literal of Y from the set $r(\mathbf{x})$. Now, if Y is treated as an input variable, then we need an incompletely specified function $f : D \times E \rightarrow B$, such that all and only the implementations of f define compatible functions with R, where an implementation of f is interpreted that it defines a function from D to B^m which maps an input minterm $\mathbf{x} \in D$ to $\mathbf{y} \in B^m$ with the property that $y_j = 1$ if and only if $(\mathbf{x}, j)$ is mapped to 1 by the implementation. Since R defines a relation between each input minterm $\mathbf{x} \in D$ and a set of literals of Y, $r(\mathbf{x})$, the function f must reflect this relation as well. On the other hand, the output of f must be defined for a pair of *minterms* of $D \times E$, rather than a pair of minterm of D and a *literal* of Y. Therefore, the output value $f(\mathbf{x}, j)$ may not be defined for some pair of minterms $(\mathbf{x}, j) \in D \times E$. For example, if there exists a pair $(\mathbf{x}, j) \in D \times E$ for which $r(\mathbf{x})$ consists of two literals, y and $\tilde{y}$, such that $j \in y$, $j \notin \tilde{y}$

and there exists another value i of E with $i \notin y$ and $i \in \tilde{y}$, then the output $f(\mathbf{x}, j)$ cannot be defined, since it depends on which literal of $r(\mathbf{x})$ will be chosen as the output of a function compatible with R. More specifically, we cannot set $f(\mathbf{x}, j) = 1$ since it implies that *every* implementation of f must map $(\mathbf{x}, j)$ to 1, although there exists a function compatible with R whose output for $\mathbf{x}$ corresponds to $\tilde{y}$, and $j \notin \tilde{y}$ means that there may exist an implementation of f that maps $(\mathbf{x}, j)$ to 0. Similarly, we cannot set $f(\mathbf{x}, j) = 0$ since it implies that *no* implementation of f maps $(\mathbf{x}, j)$ to 1, although there exists a function compatible with R whose output for $\mathbf{x}$ corresponds to y, and $j \in y$ means that there may exist an implementation of f that maps $(\mathbf{x}, j)$ to 1. Furthermore, $(\mathbf{x}, j)$ cannot be set as a don't care minterm since it implies that for any implementation of f, the output for $(\mathbf{x}, j)$ is don't care, i.e. may be either 0 or 1, regardless of the output for $(\mathbf{x}, i)$, although an implementation that maps $(\mathbf{x}, i)$ to 0 defines a function compatible with R whose output for $\mathbf{x}$ corresponds to a literal y, and thus $(\mathbf{x}, j)$ must be mapped to 1. In order to understand this situation, divide the values of E into three sets; let $\varphi(\mathbf{x})$ be the set of values of E contained in all the literals of $r(\mathbf{x})$, $\rho(\mathbf{x})$ be the set of values of E not contained in any of the literals of $r(\mathbf{x})$, and $\delta(\mathbf{x})$ the rest. Then the problem above does not arise, i.e. the output of $f(\mathbf{x}, j)$ can be defined for all $(\mathbf{x}, j)$, if for an arbitrary subset $S \subseteq \delta(\mathbf{x})$, there exists a literal $\hat{y} \in r(\mathbf{x})$ which contains all and only the values of $\varphi(\mathbf{x})$ and S. It is because in this case, we can set the output value of f for $(\mathbf{x}, j)$ to 1 if $j \in \varphi(\mathbf{x})$, 0 if $j \in \rho(\mathbf{x})$, and unspecified or don't care otherwise. If this condition holds for every $(\mathbf{x}, j) \in D \times E$, then the function f is well formulated and the minimized result has a correspondence between each $\mathbf{x} \in D$ and exactly one literal of $r(\mathbf{x})$. In this case, we know that by definition the original relation $R \subseteq D \times B^m$ is reduced to an incompletely specified function. However, for a relation in general, the function f cannot be formulated. Therefore we cannot convert R to a single output incompletely specified function with the 1-hot encoding scheme. The same statement is claimed for the 0-hot encoding scheme.

Now consider the log-based encoding scheme. Namely, treat each minterm of B^m as a single value of a multiple-valued variable Y. Thus Y can take 2^m values, denoted as E. In this case, R maps each minterm $\mathbf{x} \in D$ to a set of values of E. Denote this set of values by $r(\mathbf{x})$. If Y is treated as an input variable, then we need an incompletely specified function $f : D \times E \rightarrow B$ such that for every $\mathbf{x} \in D$, there must exist exactly one value $\sigma \in r(\mathbf{x})$ for which $(\mathbf{x}, \sigma)$ is mapped to 1 by the implementation obtained by minimizing f. Thus we need to decide, for each $\mathbf{x} \in D$, which value of $r(\mathbf{x})$ must be chosen and once one of the values has been selected, we must not choose any other value of $r(\mathbf{x})$ with which $\mathbf{x}$ is mapped to 1. In other words, the value of $f(\mathbf{x}, \sigma)$ depends on the value of $f(\mathbf{x}, \tilde{\sigma})$ for another $\tilde{\sigma} \in r(\mathbf{x})$, and thus cannot be defined individually.

This constraint cannot be handled in the conventional minimization problem. Therefore, we cannot transform the minimization problem for a relation R to the problem of minimizing an incompletely specified function with a single output and multiple-valued inputs.

Thus, any of the encoding schemes considered above cannot be used for transforming multiple binary outputs to a single multiple-valued input. We conjecture that no such encoding scheme exists.

3.4 Function Minimization and Relation Minimization

As seen in Section 3.2.2, a multiple-valued relation may happen to be an incompletely specified function. In this case, a procedure developed for minimizing relations should be invoked since the minimization of incompletely specified functions is a simpler problem for which highly optimized software exists, and thus is completed more efficiently with a procedure designed exclusively for functions. Therefore, we need a method that identifies whether a given relation is an incompletely specified function. We present one such method which, as a byproduct, can provide the on-set, the off-set, and the don't care set if the relation is an incompletely specified function.

An overview of the procedure is as follows. The input is a well-defined multiple-valued relation $R \subseteq D \times B^m$. Hence there exists an associated mapping $r(\mathbf{x})$. For an output variable y_j, let $\Phi^{(j)}$ be a set of minterms $\mathbf{x} \in D$ such that $y_j = 1$ for every minterm $\mathbf{y} \in r(\mathbf{x})$. Similarly, let $P^{(j)}$ be a set of minterms of $\mathbf{x} \in D$ such that $y_j = 0$ for every minterm $\mathbf{y} \in r(\mathbf{x})$. Let $\Delta^{(j)}$ be the remaining minterms. These sets are computed as follows. Let $s_1^{(j)}$ be a set of minterms $\mathbf{x} \in D$ such that there exists a minterm $\mathbf{y} \in r(\mathbf{x})$ in which $y_j = 0$. Let $s_2^{(j)}$ be a set of minterms $\mathbf{x} \in D$ such that there exists a minterm $\mathbf{y} \in r(\mathbf{x})$ in which $y_j = 1$. Then $\Phi^{(j)} = \neg s_1^{(j)}$, $P^{(j)} = \neg s_2^{(j)}$, and $\Delta^{(j)} = s_1^{(j)} s_2^{(j)}$. Note that the formulas for $\Phi^{(j)}$ and $P^{(j)}$ are valid if and only if R is well-defined. The proposed procedure computes the three sets $\Phi^{(j)}$, $P^{(j)}$, and $\Delta^{(j)}$ for every output y_j in parallel, and then computes $T(\mathbf{x}, \mathbf{y}) = \prod_{j=1}^{m} (\Phi^{(j)} y_j + P^{(j)} \overline{y}_j + \Delta^{(j)})$. The following theorem identifies if R is in fact an incompletely specified function by comparing R and T.

Theorem 3.4.1 *A well-defined relation $R(\mathbf{x}, \mathbf{y})$ is the characteristic function of an incompletely specified function if and only if $R(\mathbf{x}, \mathbf{y}) \equiv T(\mathbf{x}, \mathbf{y})$.*

Proof: Suppose $R(\mathbf{x}, \mathbf{y})$ is the characteristic function of an incompletely specified function $\zeta : D \rightarrow B^m$. Then ζ is represented in terms of its on-set, off-set, and don't care set functions, $\varphi : D \rightarrow B^m$,

$\rho : D \to B^m$, and $\delta : D \to B^m$ respectively, such that for each output j,

$$(a) \quad \forall \mathbf{x} \in D : (\varphi^{(j)}(\mathbf{x}) \oplus \rho^{(j)}(\mathbf{x}))\overline{\delta}^{(j)}(\mathbf{x}) + (\rho^{(j)}(\mathbf{x}) \oplus \delta^{(j)}(\mathbf{x}))\overline{\varphi}^{(j)}(\mathbf{x}) = 1,$$

$$(b) \quad R(\mathbf{x},\mathbf{y}) = 1 \Leftrightarrow y_j = \begin{cases} 1 & \text{if } \varphi^{(j)}(\mathbf{x}) + \delta^{(j)}(\mathbf{x}) = 1, \\ 0 & \text{if } \rho^{(j)}(\mathbf{x}) + \delta^{(j)}(\mathbf{x}) = 1, \end{cases} \tag{3.2}$$

where $\varphi^{(j)}$, $\rho^{(j)}$, and $\delta^{(j)}$ are the j-th function of φ, ρ, and δ respectively. Note that the condition (3.2)-(a) ensures the disjointness of $\varphi^{(j)}$, $\rho^{(j)}$, and $\delta^{(j)}$. Thus $R(\mathbf{x},\mathbf{y})$ is represented as $R(\mathbf{x},\mathbf{y}) = \prod_{j=1}^{m}(\varphi^{(j)}y_j + \rho^{(j)}\overline{y}_j + \delta^{(j)})$. Now for an arbitrary minterm $\mathbf{x} \in D$, if $y_j = 1$ in every minterm $\mathbf{y} \in r(\mathbf{x})$, then $\varphi^{(j)}(\mathbf{x}) = 1$ and $\rho^{(j)}(\mathbf{x}) = \delta^{(j)}(\mathbf{x}) = 0$. Similarly, if $y_j = 0$ in every minterm $\mathbf{y} \in r(\mathbf{x})$, then $\rho^{(j)}(\mathbf{x}) = 1$ and $\varphi^{(j)}(\mathbf{x}) = \delta^{(j)}(\mathbf{x}) = 0$. If $y_j = 1$ in some minterm of $r(\mathbf{x})$ and $y_j = 0$ in another minterm of $r(\mathbf{x})$, then $\delta^{(j)}(\mathbf{x}) = 1$ and $\varphi^{(j)}(\mathbf{x}) = \rho^{(j)}(\mathbf{x}) = 0$, since otherwise the property (3.2)-(a) is violated. Therefore, $\varphi^{(j)}$, $\rho^{(j)}$, and $\delta^{(j)}$ are the characteristic functions of $\Phi^{(j)}$, $P^{(j)}$, and $\Delta^{(j)}$ respectively. Thus $T(\mathbf{x},\mathbf{y})$ is equivalent to $R(\mathbf{x},\mathbf{y})$.

Conversely, suppose that $R(\mathbf{x},\mathbf{y})$ is equivalent to $T(\mathbf{x},\mathbf{y})$. For an arbitrary minterm $\mathbf{w} \in D$, let $J_0(\mathbf{w})$ be the set of indices j such that $\mathbf{w} \in P^{(j)}$. Similarly, let $J_1(\mathbf{w})$ and $J_2(\mathbf{w})$ be the set of indices such that $\mathbf{w} \in \Phi^{(j)}$ and $\mathbf{w} \in \Delta^{(j)}$ respectively. Let $\mathbf{t}$ be a minterm of B^m such that $t_j = 1$ if $j \in J_1(\mathbf{w})$, $t_j = 0$ if $j \in J_0(\mathbf{w})$, and t_j may be either 1 or 0 if $j \in J_2(\mathbf{w})$, where t_j is the value of the j-th variable of $\mathbf{t}$. Since $R(\mathbf{x},\mathbf{y}) = \prod_{j=1}^{m}(\Phi^{(j)}y_j + P^{(j)}\overline{y}_j + \Delta^{(j)})$, we see that $R(\mathbf{w},\mathbf{t}) = 1$. Thus the image of $\mathbf{w}$ by the relation R, $r(\mathbf{w})$, can be represented as a single product term of the output variables such that the j-th literal contains only 1 (respectively 0) if $j \in J_1(\mathbf{w})$ (respectively $j \in J_0(\mathbf{w})$) and it contains both 1 and 0 otherwise. Since $\mathbf{w}$ is arbitrary, the relation R is the characteristic function of an incompletely specified function. ■

Intuitively, $T(\mathbf{x},\mathbf{y})$ is the characteristic function of the relation such that for every $\mathbf{x} \in D$, the image of $\mathbf{x}$ by T is the set of minterms of the smallest product term defined in B^m that contains $r(\mathbf{x})$. Hence, the relation R is in fact an incompletely specified function if and only if R is identical with T.

Therefore, we can identify whether a given relation R is an incompletely specified function by checking the equivalency between $R(\mathbf{x},\mathbf{y})$ and $T(\mathbf{x},\mathbf{y})$. Note that the equivalency is trivially checked using MDD's.

Furthermore, by the proof of Theorem 3.4.1, we see that if R is an incompletely specified function, then its on-set, off-set and the don't care set are given by the $\Phi^{(j)}$'s, $P^{(j)}$'s, and $\Delta^{(j)}$'s, respectively. Hence, having checked the well-definedness of a given relation R as described in

Section 3.2.2, we first employ the procedure described above. If the relation is an incompletely specified function, we convert the $\Phi^{(j)}$'s, $P^{(j)}$'s, and $\Delta^{(j)}$'s to some required form, e.g. sum-of-products form, and invoke a conventional minimizer for ordinary logic functions, e.g. ESPRESSO-MV[45]. If it turns out that the relation is not reduced to a function, then we minimize the relation directly using a relation minimizer, GYOCRO, presented in the following section.

3.5 Heuristic Minimization of Multiple-Valued Relations

3.5.1 Problem Formulation and Overview

We consider the following problem : *given a well-defined multiple-valued relation* $R \subseteq D \times B^m$, *find a representation* $\mathcal{F}$ *with the minimum number of product terms that is compatible with* R. We propose a heuristic procedure for this problem, where the input is given as the characteristic function of R represented by an MDD.

The procedure starts with computing an initial representation compatible with the relation. Then three basic procedures, REDUCE, EXPAND, and IRREDUNDANT, are iteratively applied as long as the cost decreases, where the cost is the number of the product terms of the representation. Every procedure takes as input a compatible representation $\mathcal{F}$ and the characteristic function R of the relation.

In REDUCE, each cube $c \in \mathcal{F}$ is reduced to a smallest cube $\hat{c} \subseteq c$ such that $\mathcal{F} - \{c\} \cup \{\hat{c}\}$ is compatible with R, where $\mathcal{F} - \mathcal{S} \cup \mathcal{T}$ designates the replacement of $\mathcal{S} \subseteq \mathcal{F}$ by a set of cubes $\mathcal{T}$. It is guaranteed at the end of REDUCE that every cube c in the resulting representation is minimal, i.e. any cube $\hat{c}$ with the condition above is equal to c. EXPAND, in turn, takes each cube $c \in \mathcal{F}$ and replaces it with a relatively prime cube containing c so that a maximal number of cubes in $\mathcal{F}$ can be removed. EXPAND guarantees that every cube of the resulting representation is relatively prime. IRREDUNDANT makes the representation $\mathcal{F}$ so that it consists only of irredundant cubes. Specifically, one cube $c \in \mathcal{F}$ is processed at a time, and is removed if and only if $\mathcal{F} - \{c\}$ is compatible. Unlike the irredundant procedure of Espresso [8], which computes a minimum subset of the current representation for a Boolean function based on the concept of partially redundant cubes, our procedure is dependent on the order that the cubes are processed. In fact, the procedure is a special case of REDUCE since a redundant cube is reduced to nothing. However, experimental results show that use of IRREDUNDANT improves the computational time.

3.5.2 Initial Representation

This section describes how an initial compatible representation for a given well-defined relation $R(\mathbf{x}, \mathbf{y})$ can be computed. Our procedure takes a representation $\mathcal{F}$, which is initially empty, and processes each output adding a set of new cubes to $\mathcal{F}$. $\mathcal{F}$ becomes compatible with R once all outputs have been processed. The overall procedure is outlined in Figure 3.4.

The procedure first duplicates the relation R, and sets it to $\tilde{R}$. $\tilde{R}$ is modified during the procedure. For an output variable y_j, let $\Phi^{(j)}$ be a set of minterms $\mathbf{x} \in D$ such that $y_j = 1$ for every minterm $\mathbf{y}$ of the image of $\mathbf{x}$ by $\tilde{R}$. Similarly, let $P^{(j)}$ be a set of minterms of $\mathbf{x} \in D$ such that $y_j = 0$ for every minterm $\mathbf{y}$ with $(\mathbf{x}, \mathbf{y}) \in \tilde{R}$. Let $\Delta^{(j)}$ be the the remaining minterms of D. These are the same sets used for identifying if a given relation is an incompletely specified function in Section 3.4. Once these three sets are obtained, a two level minimizer for multiple-valued functions is invoked with $\Phi^{(j)}$, $P^{(j)}$, and $\Delta^{(j)}$ as the on-set, the off-set, and the don't care set respectively, so that a minimal sum-of-products representation $\mathcal{F}_j$ with n inputs and a single output is computed. Then $\mathcal{F}_j$ is converted to a representation with n inputs and m outputs such that the input part is identical with $\mathcal{F}_j$ and the output part has 1 only in the j-th component. The converted representation is added to $\mathcal{F}$. Once $\mathcal{F}_j$ is computed, the relation $\tilde{R}$ is modified so that for every pair of minterms $(\mathbf{x}, \mathbf{y}) \in \tilde{R}$, y_j appears complemented in $\mathbf{y}$ if and only if $\mathbf{x}$ is not covered by $\mathcal{F}_j$. Equivalently, $\tilde{R}(\mathbf{x}, \mathbf{y})$ is replaced by $\tilde{R}(\mathbf{x}, \mathbf{y})(F_j \equiv y_j)$, where $(f \equiv g)$ designates the XNOR operation between f and g. F_j is the set of minterms of D covered by $\mathcal{F}_j$, which is computed by converting $\mathcal{F}_j$ to an MDD. Once all output variables have been processed, a compatible representation is obtained in "unwrapped" form, i.e. each cube in the representation has exactly one 1 in its output part.

Theorem 3.5.1 *A representation $\mathcal{F}$ obtained by the procedure given in Figure 3.4 is compatible with the relation R.*

Proof: We first assume, without loss of generality, that the procedure processes output variables in increasing order on j, i.e. from $j = 1$ to $j = m$. Let $f : D \rightarrow B^m$ be the function defined by $\mathcal{F}$. For a given arbitrary minterm $\mathbf{x} \in D$, let $\mathbf{y}(j)$ be the minterm $f(\mathbf{x})$ restricted to the variables $\{y_1, \ldots, y_j\}$. Thus, $\mathbf{y}(m)$ is equal to $f(\mathbf{x})$, and $\mathbf{y}(1)$ is the value of the first output variable y_1 in the minterm $f(\mathbf{x})$. We also define $\mathbf{y}(0)$ as null. For a given $\tilde{\mathbf{y}} \in B^m$, we say that $\tilde{\mathbf{y}}$ satisfies the partial identity up to j if for all $i \leq j$, the value of y_i in $\tilde{\mathbf{y}}$ is equal to that in $\mathbf{y}(j)$.

We prove, by induction on j, that there exists $\tilde{\mathbf{y}} \in B^m$ such that $(\mathbf{x}, \tilde{\mathbf{y}}) \in R$ and $\tilde{\mathbf{y}}$ satisfies the partial identity up to j. Then since $\mathbf{y}(m) = \tilde{\mathbf{y}}$ and since $\mathbf{y}(m) = f(\mathbf{x})$, the proof is done.

```
function INITIAL(R(x, y))
    F ← φ;
    R̃ ← R;
    for(each output y_j){
        Φ^(j) ← {x ∈ D | ∀y ∈ B^m : R̃(x, y) = 1 ⇒ y_j = 1};
        P^(j) ← {x ∈ D | ∀y ∈ B^m : R̃(x, y) = 1 ⇒ y_j = 0};
        Δ^(j) ← complement(Φ^(j) ∪ P^(j));
        F_j ← minimize(Φ^(j), Δ^(j), P^(j));
        R̃(x, y) ← R̃(x, y)(F_j ≡ y_j);
        F ← F ∪ convert(F_j);
    }
    return F;
```

Figure 3.4: Procedure for Computing an Initial Representation

Furthermore, denoting by $\tilde{R}(j)$ the relation $\tilde{R}$ obtained just after processing y_j in the procedure, we also show that $(\mathbf{x}, \mathbf{y}) \in \tilde{R}(j)$ if and only if $\mathbf{y}$ satisfies the partial identity up to j. For a special case, we define $\tilde{R}(0) = R$.

For the base case, where $j = 0$, $\mathbf{y}(0)$ is null. Since R is well-defined, there exists $\tilde{\mathbf{y}}$ such that $(\mathbf{x}, \tilde{\mathbf{y}}) \in R$, and thus the condition holds. Also, the second condition of $\tilde{R}(j)$ is trivially true for this case.

Suppose that the conditions hold up to $j - 1$, where $j \geq 1$. We first note that the minterm given as the image of $\mathbf{x}$ by the representation obtained just after processing y_j satisfies the partial identity up to j. This is because when y_j is processed, the procedure adds a set of cubes whose output part has 1 only at the j-th component, and thus the pattern of the image for the variables $\{y_1, \ldots, y_j\}$ is identical with that of the image obtained by the final representation, i.e. $f(\mathbf{x})$. Thus y_j is equal to 1 in $\mathbf{y}(j)$ if and only if $\mathbf{x}$ is covered by $\mathcal{F}_j$, where $\mathcal{F}_j$ is the single-output representation obtained in the procedure when y_j is processed. Since $\mathcal{F}_j$ is the result of minimizing a function with the on-set and off-set as defined in the procedure, there exists a minterm $\tilde{\mathbf{y}} \in B^m$ such that

$(\mathbf{x}, \tilde{\mathbf{y}}) \in \tilde{R}(j-1)$ and the value of y_j in $\tilde{\mathbf{y}}$ is equal to that in $\mathbf{y}(j)$. Therefore, with the induction hypothesis, $\tilde{\mathbf{y}}$ satisfies the partial identity up to j. Also, since $\tilde{R}(j-1)$ is a subset of R, $(\mathbf{x}, \tilde{\mathbf{y}}) \in R$.

Now consider the relation $\tilde{R}(j)$. The induction hypothesis implies that $(\mathbf{x}, \mathbf{y}) \in \tilde{R}(j-1)$ if and only if $\mathbf{y}$ satisfies the partial identity up to $j-1$. Furthermore, $(\mathbf{x}, \mathbf{y}) \in \tilde{R}(j)$ if and only if $(\mathbf{x}, \mathbf{y}) \in \tilde{R}(j-1)$ and the value of y_j in $\mathbf{y}$ is the same as that in $\mathbf{y}(j)$. It follows that $(\mathbf{x}, \mathbf{y}) \in \tilde{R}(j)$ if and only if $\mathbf{y}$ satisfies the partial identity up to j. Therefore, the two conditions hold for j.

Hence, by induction, $(\mathbf{x}, f(\mathbf{x})) \in R$ for all $\mathbf{x} \in D$, and thus the representation $\mathcal{F}$ is compatible with R. ■

3.5.3 Computing the Characteristic Function of a Set of Cubes

REDUCE, EXPAND, and IRREDUNDANT procedures process one cube $c \in \mathcal{F}$ at a time. In each operation, we need to compute the characteristic function corresponding to $\mathcal{F} - \{c\}$, denoted by $F_c(\mathbf{x}, \mathbf{y})$. The characteristic function of a function $f : D \rightarrow B^m$ is the characteristic function of the relation $F = \{(\mathbf{x}, \mathbf{y}) \in D \times B^m \mid \mathbf{y} = f(\mathbf{x})\}$. Throughout the rest of this chapter, we call F_c the characteristic function of $\mathcal{F} - \{c\}$. In general, the characteristic function of a representation $\mathcal{F}$, denoted by $F(\mathbf{x}, \mathbf{y})$, can be computed by scanning all the cubes of $\mathcal{F}$ to obtain the function $f^{(j)} : D \rightarrow B$ for each output y_j, followed by setting $F(\mathbf{x}, \mathbf{y}) = \prod_{j=1}^{m} (f^{(j)}(\mathbf{x}) \equiv y_j)$. This will be referred to later as the "first" method. However, it is time consuming, especially when the size of $\mathcal{F}$ is large. In fact, $F_c(\mathbf{x}, \mathbf{y})$ can be computed much more easily by the following method if F is already available.

Let $\mathcal{S}$ be the subset of $\mathcal{F}$ defined as $\mathcal{S} = \{p \in \mathcal{F} \mid p \neq c \text{ and } M(p) \cap M(c) \neq \phi\}$, where we recall that $M(p)$ is a set of minterms of D covered by the product term corresponding to the input part of p. Let $F_{\mathcal{S}} : D \times B^m \rightarrow B$ be a function such that $F_{\mathcal{S}}(\mathbf{x}, \mathbf{y}) = 1$ if and only if

$$\forall j \in \{1, \ldots, m\} \quad : \quad y_j = \begin{cases} 1 & \text{if } O(c)_j = 1 \text{ and } \exists p \in \mathcal{S} \text{ such that } \mathbf{x} \in M(p) \text{ and } O(p)_j = 1, \\ 0 & \text{otherwise,} \end{cases}$$

where y_j is the value of the j-th variable in $\mathbf{y}$. Now observe that for a minterm $\mathbf{x} \notin M(c)$, the image of $\mathbf{x}$ by F_c is identical with the image of $\mathbf{x}$ by F. For a minterm $\mathbf{x} \in M(c)$, the value of y_j of the image of $\mathbf{x}$ by F_c is identical with y_j of the image by F if $O(c)_j = 0$. If $O(c)_j = 1$ and there exists no $p \in \mathcal{F} - \{c\}$ such that $\mathbf{x} \in M(p)$ and $O(p)_j = 1$, then $y_j = 0$. Otherwise $y_j = 1$. Thus

$F_c(\mathbf{x},\mathbf{y}) = 1$ if and only if

$$\forall j \in \{1,\ldots,m\} \ : \ y_j = \begin{cases} \tilde{y}_j & \text{if } \mathbf{x} \in M(c) \text{ and } O(c)_j = 1, \\ \hat{y}_j & \text{otherwise,} \end{cases}$$

where $\tilde{y}_j$ is the value of the j-th variable in the minterm $\tilde{\mathbf{y}}$ such that $F_S(\mathbf{x},\tilde{\mathbf{y}}) = 1$ and $\hat{y}_j$ is the value of the j-th variable in the minterm $\hat{\mathbf{y}}$ such that $F(\mathbf{x},\hat{\mathbf{y}}) = 1$. In order to accomplish this computation, we first introduce additional variables $T = \{t_1,\ldots,t_m\}$ and $Z = \{z_1,\ldots,z_m\}$ to represent F and F_S in terms of $(\mathbf{x},\mathbf{t})$ and $(\mathbf{x},\mathbf{z})$ respectively, and compute a characteristic function $Y_1(\mathbf{y},\mathbf{t},\mathbf{z})$ such that $Y_1(\mathbf{y},\mathbf{t},\mathbf{z}) = 1$ if and only if for each $j \in \{1,\ldots,m\}$, $y_j = t_j$ if $O(c)_j = 0$ and $y_j = z_j$ if $O(c)_j = 1$. Specifically, $Y_1(\mathbf{y},\mathbf{t},\mathbf{z})$ is given by $\prod_{j\in Y_l(c)} (y_j \equiv t_j) \prod_{j\notin Y_l(c)} (y_j \equiv z_j)$, where $Y_l(c) = \{j \in \{1,\ldots,m\} \mid O(c)_j = 0\}$. Then, $F_c(\mathbf{x},\mathbf{y}) = 1$ if and only if either $\mathbf{x} \notin M(c)$ and $F(\mathbf{x},\mathbf{y}) = 1$, or $\mathbf{x} \in M(c)$ and there exists $(\mathbf{t},\mathbf{z})$ such that $F(\mathbf{x},\mathbf{t}) = F_S(\mathbf{x},\mathbf{z}) = Y_1(\mathbf{y},\mathbf{t},\mathbf{z}) = 1$. These computations are performed on MDD's. In this way, F_c is computed from F by scanning a subset $\mathcal{S}$ of $\mathcal{F}$.

According to experiments, the first method is slightly faster for small examples such as 10 variables and 20 cubes. However, the CPU time for the second method is almost invariant in the size of the representations and is much faster for moderate sized examples. For one with 33 binary variables and 553 cubes, the second method was 20 times faster.

The second method uses the characteristic function F of $\mathcal{F}$. The computation of F is done once at the beginning of the entire procedure using the first method. However, it must be updated whenever a cube of $\mathcal{F}$ is replaced by another cube either in the REDUCE or the EXPAND procedure. As a converse of the second method, given $\mathcal{F}$, $c \in \mathcal{F}$, and the characteristic function F_c of $\mathcal{F} - \{c\}$, $F(\mathbf{x},\mathbf{y})$ can be computed as follows. For a minterm $\mathbf{x} \in D$, the value of y_j of the image of $\mathbf{x}$ by F is identical with y_j of the image by F_c if $\mathbf{x} \notin M(c)$ or $\mathbf{x} \in M(c)$ but $O(c)_j = 0$. Otherwise $y_j = 1$. Thus using new variables $T = \{t_1,\ldots,t_m\}$ to represent F_c in terms of $(\mathbf{x},\mathbf{t})$, we see that $F(\mathbf{x},\mathbf{y}) = 1$ if and only if either $\mathbf{x} \notin M(c)$ and $F_c(\mathbf{x},\mathbf{y}) = 1$, or there exists $\mathbf{t}$ such that $F_c(\mathbf{x},\mathbf{t}) = 1$ and $Y_2(\mathbf{y},\mathbf{t}) = 1$, where $Y_2(\mathbf{y},\mathbf{t}) = 1$ if and only if $y_j = t_j$ for all $j \in Y_l(c)$ and $y_j = 1$ for all $j \notin Y_l(c)$.

In this way, each of the REDUCE, EXPAND, and IRREDUNDANT procedures takes as an additional input the characteristic function F of $\mathcal{F}$, updates it whenever $\mathcal{F}$ changes, and hands it to the succeeding procedure. Note that it is not necessary to update F in the IRREDUNDANT procedure since if c is removed, then F_c becomes the characteristic function of the new representation.

3.5.4 REDUCE

A reduction is an operation which takes a cube c and returns a cube $\hat{c} \subseteq c$, where $\hat{c}$ may be empty. A reduction is *valid* if the replacement of c with the reduced cube results in a representation compatible with R.

Definition: Maximally Reduced Cube

For a given representation $\mathcal{F}$ compatible with R and a cube $c \in \mathcal{F}$, a cube $c^- \subseteq c$ is a **maximally reduced cube** for c in $\mathcal{F}$ if $\mathcal{F} - \{c\} \cup \{c^-\}$ is compatible with R and there exists no cube $\hat{c}^- \subset c^-$ such that $\mathcal{F} - \{c\} \cup \{\hat{c}^-\}$ is compatible.

If a maximally reduced cube for c in $\mathcal{F}$ is c itself, we say c is maximally reduced. The goal of the REDUCE procedure is to compute a compatible representation which consists of only maximally reduced cubes. It is known that if R is in fact an incompletely specified function, i.e. $r(\mathbf{x})$ can be expressed by a single cube for every $\mathbf{x} \in D$, then the maximally reduced cube c^- for c in $\mathcal{F}$ is unique. Also, for any cube $\tilde{c}$ such that $c^- \subseteq \tilde{c} \subseteq c$, the replacement of c by $\tilde{c}$ preserves compatibility. Thus the maximally reduced cube is easily obtained by lowering, in any order, one part at a time from 1 to 0 for c, followed by checking the compatibility of the resulting representation, until no valid reduction is possible. Note that the compatibility check of a representation $\mathcal{F}$ is equivalent to the containment check between $F(\mathbf{x},\mathbf{y})$ and $R(\mathbf{x},\mathbf{y})$, i.e. $F(\mathbf{x},\mathbf{y}) \subseteq R(\mathbf{x},\mathbf{y})$, which can be done efficiently with MDD's. Therefore, a two level minimizer for ordinary logic functions achieves the goal mentioned above by taking each cube and replacing it with the unique maximally reduced cube.

The REDUCE procedure of the proposed method also processes one cube at a time and replaces it by a maximally reduced cube. However, only a weak form of uniqueness holds.

Theorem 3.5.2 *Let $\mathcal{F}$ be a representation compatible with a relation R. The input part of a maximally reduced cube for $c \in \mathcal{F}$ is unique.*

Proof: Suppose that there are two maximally reduced cubes $c^{(1)}$ and $c^{(2)}$. The objective is to show that $M(c^{(1)}) \equiv M(c^{(2)})$. Suppose first that either of them, say $M(c^{(1)})$, is empty. Then $M(c^{(2)})$ must be empty and thus the proof is done since otherwise $c^{(2)}$ is not a maximally reduced cube by definition.

Now we assume that neither $M(c^{(1)})$ nor $M(c^{(2)})$ is empty. We first show that $M(c^{(1)}) \cap M(c^{(2)}) \neq \phi$. Suppose the contrary. Let $\hat{c}^{(1)}$ be any cube strictly contained in $c^{(1)}$ obtained by

reducing only the input part of $c^{(1)}$. $\hat{c}^{(1)}$ may be an empty cube. Let ω be any minterm included in $M(c^{(1)}) \cap \neg M(\hat{c}^{(1)})$. Since $c^{(1)}$ and $c^{(2)}$ are assumed disjoint, $\omega \notin M(c^{(2)})$. However, since $\mathcal{F} - \{c\} \cup \{c^{(2)}\}$ is compatible, the image of ω by $\mathcal{F} - \{c\}$ must be a member of $r(\omega)$. Thus a reduction from $c^{(1)}$ to $\hat{c}^{(1)}$ is valid, which is a contradiction. Therefore $M(c^{(1)})$ and $M(c^{(2)})$ are not disjoint.

Suppose that $M(c^{(1)}) \not\equiv M(c^{(2)})$. Let $\mathbf{m}$ be any minterm included in $M(c^{(1)}) \cap \neg M(c^{(2)})$. Let m_j be the value of the j-th input variable in $\mathbf{m}$. Let $\hat{c}^{(1)}$ be the cube which is identical with $c^{(1)}$ except that $I(\hat{c}^{(1)})_j = \neg m_j \cap I(c^{(1)})_j$, where j is an index such that $m_j \notin I(c^{(2)})_j$. Note that $\hat{c}^{(1)}$ is not an empty cube since otherwise $I(c^{(1)})_j \cap I(c^{(2)})_j = \phi$, which implies $M(c^{(1)}) \cap M(c^{(2)}) = \phi$. Since $c^{(1)}$ is maximally reduced, $\mathcal{F} - \{c\} \cup \{\hat{c}^{(1)}\}$ is not compatible. This implies that there exists a minterm $\pi \in M(c^{(1)}) \cap \neg M(\hat{c}^{(1)})$ such that the image of π by $\mathcal{F} - \{c\}$ is not a member of $r(\pi)$. However, we know that $\pi_j = m_j$, and thus $\pi \notin M(c^{(2)})$. Since $\mathcal{F} - \{c\} \cup \{c^{(2)}\}$ is compatible and since $\pi \notin M(c^{(2)})$, the image of π by $\mathcal{F} - \{c\}$ must be a member of $r(\pi)$, which is a contradiction. Thus the assumption that $M(c^{(1)}) \not\equiv M(c^{(2)})$ is incorrect. ■

Theorem 3.5.3 *Given a relation R and a cube $c \in \mathcal{F}$, where $\mathcal{F}$ is a representation compatible with R, let $\tilde{c} \subseteq c$ be a cube such that $O(\tilde{c}) = O(c)$ and $M(\tilde{c}) \supseteq M(c^{-})$, where c^{-} is a maximally reduced cube for c in $\mathcal{F}$. Then the reduction from c to $\tilde{c}$ is valid.*

Proof: For a minterm $\mathbf{m} \in M(\tilde{c})$, we have $\mathbf{m} \in M(c)$. Since $O(\tilde{c}) = O(c)$, the image of $\mathbf{m}$ by $\mathcal{F} - \{c\} \cup \{\tilde{c}\}$ is equal to the image by $\mathcal{F}$. For a minterm $\mathbf{m} \in M(c) \cap \neg M(\tilde{c})$, $\mathbf{m} \notin M(c^{-})$. Thus the image of $\mathbf{m}$ by $\mathcal{F} - \{c\} \cup \{\tilde{c}\}$ is equal to the image by $\mathcal{F} - \{c\} \cup \{c^{-}\}$, which is a member of $r(\mathbf{m})$. Therefore, $\mathcal{F} - \{c\} \cup \{\tilde{c}\}$ is a compatible representation. ■

Thus by fixing the output part of c, we can lower each part of $I(c)$ with the value 1 to 0, one at a time, and accept the reduction if the new representation is compatible with R. The unique input part of a maximally reduced cube is obtained once all the input components have been processed. Note that c is redundant in $\mathcal{F}$ if there is a component $I(c)_j$ in which every part is 0.

We can check the compatibility of the new representation as follows, without computing its characteristic function. Let $\tilde{c}$ be a cube which is identical with c except that a single part of $I(c)$ has been lowered from 1 to 0. Let $Q = M(c) \cap \neg M(\tilde{c})$. We see that $\mathcal{F} - \{c\} \cup \{\tilde{c}\}$ is compatible if and only if for all minterms $\mathbf{m} \in Q$, the image of $\mathbf{m}$ by $\mathcal{F} - \{c\}$ is a member of $r(\mathbf{m})$. Equivalently, $\mathcal{F} - \{c\} \cup \{\tilde{c}\}$ is compatible with R if and only if for all $\mathbf{x} \in Q$, there exists $\mathbf{y}$ such that $R(\mathbf{x}, \mathbf{y}) = 1$ and $F_c(\mathbf{x}, \mathbf{y}) = 1$. This check is easily done on MDD's.

Thus the reduction of the input part is performed in a straightforward way. However, unlike the input part, the following example shows that the output part of the maximally reduced cube is not unique.

Example 3.5.1 *Suppose that a representation $\mathcal{F}$ shown in Table 3.5 is compatible with R, where part of its relations are shown in Table 3.5. Then $c_1^{(1)} = [1021 \mid 1000]$ and $c_1^{(2)} = [1021 \mid 0110]$ are both maximally reduced cubes for c_1 in $\mathcal{F}$.*

Representation $\mathcal{F}$			Relation R	
cube	Input	Output	$\mathbf{x} \in B^4$	$\mathbf{y} \in B^4$
c_1	1021	1110	1011	1111, 0111, 1001
c_2	1201	0100	1001	1111, 0111, 1101, 1001, 0001
c_3	2021	0001	1101	0100
c_4	2211	0001	1111	0001

Table 3.5: Example where a Maximally Reduced Cube is not Unique

Since we want to reduce c as much as possible, we want to find the smallest maximally reduced cube c^-, i.e. the maximally reduced cube with the minimum number of 1's in the output part. One difficulty is that compatibility does not necessarily hold for a cube $\tilde{c}$ such that $c^- \subset \tilde{c} \subset c$. For Example 3.5.1, although $\mathcal{F}$ and $\mathcal{F} - \{c_1\} \cup \{c_1^{(1)}\}$ are both compatible with R, for $p = [1021 \mid 1100]$ or $q = [1021 \mid 1010]$, the replacement of c by either p or q results in an incompatible representation. Thus the smallest maximally reduced cube may not be found by greedily reducing the output part of c.

Now consider a cube c such that its input part is maximally reduced. We define a *feasible cube* for c as follows: c^* is feasible if $\mathcal{F} - \{c\} \cup \{c^*\}$ is compatible with R, the input part of c^* is equal to that of c, and $c^* \subseteq c$. A feasible cube with the minimum number of 1's in the output part is a maximally reduced cube we seek. Our objective is to compute the characteristic function h of the set of all the feasible cubes for c, and choose one with the minimum number of 1's in the output part. For a set of components $Y_h = \{j \in \{1, \ldots, m\} \mid O(c)_j = 1\}$, consider a Boolean space $B^{|Y_h|}$ defined by the output variables of Y_h. For a minterm $\mathbf{y} \in B^{|Y_h|}$, we define a reduced cube c^* for c with respect to $\mathbf{y}$ as follows. The input part is equal to that of c, and $O(c^*)_j = 1$ if and only if $j \in Y_h$ and $y_j = 1$. Let $h : B^{|Y_h|} \rightarrow B$ be a function such that $h(\mathbf{y}) = 1$ if and only if c^* is feasible, where c^* is a reduced cube for c with respect to $\mathbf{y}$. By definition, $h(\mathbf{y}) = 1$ if and

only if the following property holds for every minterm $\mathbf{x} \in M(c)$:

$$\exists \tilde{\mathbf{y}} \in B^m : \quad R(\mathbf{x}, \tilde{\mathbf{y}}) = 1, \quad \forall j \in \{1, \ldots, m\} : \quad \tilde{y}_j = \begin{cases} \hat{y}_j + y_j & \text{if } j \in Y_h, \\ \hat{y}_j & \text{if } j \notin Y_h, \end{cases} \tag{3.3}$$

where $\hat{y}_j$ is the value of the j-th variable of the minterm $\hat{\mathbf{y}} \in B^m$ such that $F_c(\mathbf{x}, \hat{\mathbf{y}}) = 1$.

Let $H : M(c) \times B^{|Y_h|} \rightarrow B$ be a function such that $H(\mathbf{x}, \mathbf{y}) = 1$ if and only if (3.3) holds for $(\mathbf{x}, \mathbf{y})$. Then by definition, $h(\mathbf{y}) = 1$ if and only if $H(\mathbf{x}, \mathbf{y}) = 1$ for all $\mathbf{x} \in M(c)$. In order to compute H on MDD's, we first introduce additional variables $T = \{t_1, \ldots, t_m\}$ and $Z = \{z_1, \ldots, z_m\}$ to represent R and F_c in terms of $(\mathbf{x}, \mathbf{z})$ and $(\mathbf{x}, \mathbf{t})$, and compute a characteristic function $Y_3(\mathbf{y}, \mathbf{t}, \mathbf{z})$ such that $Y_3(\mathbf{y}, \mathbf{t}, \mathbf{z}) = 1$ if and only if for each $j \in \{1, \ldots, m\}$, $z_j = t_j + y_j$ if $j \in Y_h$ and $z_j = t_j$ if $j \notin Y_h$. Namely, $Y_3(\mathbf{y}, \mathbf{t}, \mathbf{z}) = \prod_{j \in Y_h} (z_j \equiv (t_j + y_j)) \prod_{j \notin Y_h} (z_j \equiv t_j)$. Then $H(\mathbf{x}, \mathbf{y}) = 1$ if and only if $\mathbf{x} \in M(c)$ and there exists $(\mathbf{t}, \mathbf{z})$ such that $R(\mathbf{x}, \mathbf{z}) = F_c(\mathbf{x}, \mathbf{t}) = Y_3(\mathbf{y}, \mathbf{t}, \mathbf{z}) = 1$.

Note that $h(\mathbf{y})$ is represented by a BDD, since the output variables are all binary. Once $h(\mathbf{y})$ is computed, a maximally reduced cube with the minimum number of 1's is obtained efficiently. In fact, the following theorem is analogous to a result of a BDD based approach for a covering problem [32].

Theorem 3.5.4 *A maximally reduced cube for c in $\mathcal{F}$ with the minimum number of 1's is given by a shortest path connecting a 1 leaf to the root of a BDD for the function $h(\mathbf{y})$ defined above, where the length of an edge of the BDD is 1 if the edge is a 1 edge and 0 if the edge is a 0 edge.*

Proof: For a reduced cube c^* with respect to $\mathbf{y}$, $O(c^*)_j = 1$ if and only if $y_j = 1$. Thus a maximally reduced cube with the minimum number of 1's is a reduced cube for $\mathbf{y}$ with the minimum number of 1's such that $h(\mathbf{y}) = 1$. For any path from the root of the BDD for h which ends in the 1 leaf, one can obtain a minterm $\mathbf{y}$ such that $h(\mathbf{y}) = 1$, by setting $y_j = 1$ only if the path contains a 1 edge incident with a node for y_j. The number of 1's of the minterm is minimum among all the minterms represented by the path. Let $\mathbf{y}$ be the minterm with the minimum number of 1's associated with a shortest path P that ends in the 1 leaf. The proof is done if we show that $\mathbf{y}$ has the minimum number of 1's among all the minterms in the on-set of h. Suppose there is another minterm $\mathbf{y}'$ such that the number of the 1's of $\mathbf{y}'$ is less than that of $\mathbf{y}$. Then there exists at least one path in the BDD corresponding to $\mathbf{y}'$. Let P' be the shortest such path. Since the number of 1's of $\mathbf{y}'$ is at least the length of P', P' must be shorter than P, which is a contradiction. ■

Therefore, once the input part of a cube $c \in \mathcal{F}$ has been maximally reduced, the smallest maximally reduced cube for c is obtained by computing a function $h(\mathrm{y})$, followed by performing a shortest path algorithm, which runs in time linear in the number of the nodes of BDD's.

We have shown how to compute a maximally reduced cube for each cube of $\mathcal{F}$. However, we want to compute a representation in which every cube is maximally reduced. Due to the nature of sum-of-products representations for relations, a property of a cube like maximal reduction, which may hold at some time in the reduction process, may cease to hold when other cubes are subsequently reduced. We must deal with this difficulty which does not arise for functions. This is illustrated by the following example.

Example 3.5.2 *In Example 3.5.1, suppose* c_1 *has been replaced by a maximally reduced cube* $c_1^{(1)}$. *If we have replaced* c_2 *by its maximally reduced cube* $c_2^{(1)} = [1101 \mid 0100]$, *then* $c_1^{(1)}$ *can be further reduced to* $[1011 \mid 1000]$ *while still keeping the compatibility of the resulting representation.*

One solution is to iterate the entire procedure, until no cube is replaced by a smaller cube. Then at the end, it is guaranteed that every cube of the final representation is maximally reduced.

As with ordinary logic functions, the result of this procedure depends on the order of the cubes to be processed. Experiments were performed with several ordering strategies. Although the results varied slightly with different orderings, none was always better than another. As in the case of the minimization of functions [45], the final results seemed to be independent of the ordering strategies in general. Thus we order the cubes with the same strategy employed in Espresso [8], in which the largest cube is processed first and the rest are sorted in increasing order of the number of mismatches (distance) of each cube against the largest one.

3.5.5 EXPAND

The objective of EXPAND is to remove as many cubes as possible from a given compatible representation. Furthermore, we want the final representation of the procedure to consist of relatively prime cubes. The proposed procedure achieves this goal by processing one cube at a time, in which the cube is expanded so that a maximal number of cubes is removed. An expansion is an operation which takes a cube c and returns a cube $\hat{c} \supseteq c$. The expansion is *valid* if the replacement of c with $\hat{c}$ results in a compatible representation. The result of the procedure is order dependent and we sort the cubes, as in ESPRESSO-MV [45], in increasing order of the weights, where the weight of a cube is defined as the number of parts at which the cube has 1.

The expand procedure for each cube $c \in \mathcal{F}$, EXPAND1, is designed as an extension of the expand procedure of Espresso, and is illustrated in Figure 3.5. EXPAND1 employs a covering matrix C which was introduced in Espresso. C has $(t+m)$ columns and as many rows as cubes of $\mathcal{F} - \{c\}$, where t is the number of the parts of $I(c)$, which is given by the sum of the number of values that each input variable can assume. Each element of C, C_{ij}, is defined as follows:

$$\forall j \in \{1,\ldots,t\}: \quad C_{ij} = \begin{cases} 1 & \text{if the } j\text{-th part of } I(c) \text{ is 0 and the } j\text{-th part of } I(\mathcal{F}^{(i)}) \text{ is 1,} \\ 0 & \text{otherwise,} \end{cases}$$

$$\forall j \in \{1,\ldots,m\}: \quad C_{i(t+j)} = \begin{cases} 1 & \text{if } O(c)_j = 0 \text{ and } O(\mathcal{F}^{(i)})_j = 1, \\ 0 & \text{otherwise,} \end{cases}$$

where $\mathcal{F}^{(i)}$ is the i-th cube of $\mathcal{F} - \{c\}$. The covering matrix allows each part of c to be handled in a uniform way, without making any distinction among the input or the output variables.

Throughout EXPAND1, we maintain two sets of columns of C, $\mathcal{R}$ and $\mathcal{L}$, where $\mathcal{R}$ is initially a set of columns at which c has 1, and $\mathcal{L}$ is empty. $\mathcal{R}$ is called a *raising set* and $\mathcal{L}$ is called a *lowering set.* $\mathcal{R}$ is used to store the columns that have been raised, while $\mathcal{L}$ maintains the columns that have been determined not to be raised.

Among all the operations of EXPAND1, the maximal-feasible-covering (MFC) operation is key, in which directions of expansion for c are determined. Other operations are used to complete the procedure efficiently. The objective of MFC is to expand c so that the maximum number of cubes of $\mathcal{F} - \{c\}$ can be removed. Overall flow of the operation is as follows. First, for each cube $p \in \mathcal{F} - \{c\}$ corresponding to each row of the covering matrix C, compute the smallest cube $c^*(p)$ such that $\mathcal{F} - \{c,p\} \cup \{c^*(p)\}$ is compatible. $c^*(p)$ may not exist. We choose the smallest such cube since we want to leave freedom of expansion for eliminating other cubes. Note that $c^*(p)$ may not cover p.[2] If $c^*(p)$ exists, then we compute the maximum subset $S(p)$ of $\mathcal{F} - \{c,p\}$ such that $\mathcal{F} - \{c,p\} - S(p) \cup \{c^*(p)\}$ is compatible. Once all the cubes corresponding to the rows of C have been processed, and if no $c^*(p)$ exists, then we exit the operation. Otherwise, a cube $c^*(q)$ with the maximum cardinality of $S(q)$ is chosen. Then the rows of C corresponding to the cubes of $q \cup S(q)$ are removed and the characteristic function of the new representation except c is computed as F_c. Finally, the operation is exited with the set of newly raised columns.

Practically, it is expensive to accomplish these procedures completely and thus we introduce some restrictions. First, $S(p)$ is restricted to those covered by $c^*(p)$, and $c^*(q)$ is chosen as the one that covers the largest number of $c^*(p)$'s over all the p's. Thus we do not compute $S(p)$ at all and

[2] In fact $c^*(p)$ may not even cover c, but we exclude this case in our procedure.

```
procedure EXPAND1(c, F, R, F)
begin
    F_c ← CFc(F, F);   /* compute the characteristic function of F − {c} */
    C ← COVERING_MATRIX(c, F);
    R ← {j | The j-th part of c is 1.};
    L ← φ;
    while(|R| + |L| < (t + m) and C ≠ φ){
        X_E ← ESSENTIAL(c, F_c, R, R, L);
        L ← L ∪ X_E;
        C ← ELM1(C, X_E);
        J ← MFC(C, c, F_c, R, R);
        if(|J| = 0)   J ← EG(C);
        R ← R ∪ J;
        C ← ELM2(C, J);
    }
    if(|R| + |L| < (t + m))   {R, L} ← GREEDY(c, F_c, R, R, L);
    c ← RAISE(c, R);
    if(c has been expanded)   F ← CF(F_c, c);
end
```

Figure 3.5: EXPAND1

the rows of C covered by $c^*(q)$ are eliminated by another operation. Now, we still need to update F_c once a cube $c^*(q)$ is chosen. Furthermore, in order to compute $c^*(p)$ for each $p \in \mathcal{F} - \{c\}$, we need to compute the characteristic function of $\mathcal{F} - \{c, p\}$. These characteristic functions are necessary so that we can evaluate the functionality of $\mathcal{F} - \{c, q\} - S(q) \cup \{c^*(q)\}$ (and $\mathcal{F} - \{c, p\} \cup \{c^*(p)\}$) without computing the real characteristic functions for the representations. However, we can observe that both of these tasks are necessary because $c^*(p)$ may not cover p. This is because if $c^*(q)$ covers q as well as all the cubes of $S(q)$, then the functionality of $\mathcal{F} - \{c, q\} - S(q) \cup c^*(q)$ is identical with the functionality of $\mathcal{F} - \{c\} \cup c^*(q)$. Thus as long as the characteristic function for $\mathcal{F} - \{c, q\} - S(q)$ is used to evaluate the functionality of the entire representation, the characteristic function for $\mathcal{F} - \{c\}$, F_c, can be used as an alternative. Note that F_c does not reflect the functionality of $\mathcal{F} - \{c, q\} - S(q)$, but still the functionality of $\mathcal{F} - \{c, q\} - S(q) \cup c^*(q)$ can be correctly evaluated with the help of $c^*(q)$. The same statement holds for the characteristic function of $\mathcal{F} - \{c, p\}$. Since F_c is already available, with the additional restriction that $c^*(p) \supseteq p$, the entire procedure of MFC can be completed without computing characteristic functions. Due to the enormous improvement in computational time, we employ this restriction.

What remains in MFC is a computation of $c^*(p)$, the smallest cube containing both c and p such that $\mathcal{F} - \{c, p\} \cup \{c^*(p)\}$ is compatible with R. The smallest cube containing both c and p, denoted as $\hat{c}$, is obtained by taking the part-wise union between c and p [45]. In other words, if p corresponds to the i-th row of C, then for $j \in \{1, \ldots, t\}$, the j-th part of $I(\hat{c})$ is 0 if and only if $C_{ij} = 0$ and $j \notin \mathcal{R}$, while $O(\hat{c})_j = 0$ if and only if $C_{i(t+j)} = 0$ and $(t + j) \notin \mathcal{R}$ for $j \in \{1, \ldots, m\}$. If R is an incompletely specified function, then $c^*(p) = \hat{c}$ if $\mathcal{F} - \{c, p\} \cup \{\hat{c}\}$ is compatible, otherwise $c^*(p)$ does not exist. For relations, however, it is claimed, as with REDUCE, that there exist cases where a compatible representation is obtained by raising the output part of $\hat{c}$ even though $\mathcal{F} - \{c, p\} \cup \{\hat{c}\}$ is originally incompatible. One sees in this case that raising the input part of $\hat{c}$ does not help since the image of a minterm for which the incompatibility occurs is not changed unless either the output part of $\hat{c}$ or other cubes of $\mathcal{F} - \{c, p\}$ are modified. Considering that the functionality of $\mathcal{F} - \{c, p\} \cup \{\hat{c}\}$ is identical with that of $\mathcal{F} - \{c\} \cup \{\hat{c}\}$, we define a *feasible cube* for $\hat{c}$ as a cube c^* such that $\mathcal{F} - \{c\} \cup \{c^*\}$ is compatible with R, the input part of c^* is equal to that of $\hat{c}$, and $c^* \supseteq \hat{c}$. Then our objective is to find the smallest feasible cube for $\hat{c}$.

This is done with a similar technique used in REDUCE, i.e. we compute the characteristic function l of the set of all the feasible cubes for $\hat{c}$, and choose the smallest one if it exists. Let $Y_l = \{j \in \{1, \ldots, m\} \mid O(\hat{c})_j = 0\}$, and consider a cube corresponding to a minterm $\mathbf{y} \in B^{|Y_l|}$ such that the input part is identical with that of $\hat{c}$ and the j-th output part is 0 if and only if $j \in Y_l$

and $y_j = 0$. Let $l : B^{|Y_l|} \rightarrow B$ be a function such that $l(\mathbf{y}) = 1$ if and only if $\mathcal{F} - \{c\} \cup \{c^*\}$ is compatible with R, where c^* is the cube corresponding to $\mathbf{y}$. There is a one-to-one correspondence between a feasible cube and a minterm $\mathbf{y}$ such that $l(\mathbf{y}) = 1$, and thus our objective is to find the minterm $\mathbf{y}$ with the minimum number of 1's in the output part for which $l(\mathbf{y}) = 1$. We see that $l(\mathbf{y}) = 1$ if and only if the following property holds for every minterm $\mathbf{x} \in M(\hat{c})$:

$$\exists \tilde{\mathbf{y}} \in B^m : \quad R(\mathbf{x}, \tilde{\mathbf{y}}) = 1, \quad \forall j \in \{1, \ldots m\} : \quad \tilde{y}_j = \begin{cases} \hat{y}_j + y_j & \text{if } j \in Y_l, \\ 1 & \text{if } j \notin Y_l, \end{cases} \tag{3.4}$$

where $\hat{y}_j$ is the value of the j-th variable of the minterm $\hat{\mathbf{y}}$ such that $F_c(\mathbf{x}, \hat{\mathbf{y}}) = 1$. In order to compute l, we use a function $L : M(\hat{c}) \times B^{|Y_l|} \rightarrow B$ such that $L(\mathbf{x}, \mathbf{y}) = 1$ if and only if (3.4) holds for $(\mathbf{x}, \mathbf{y})$, as with the REDUCE procedure. Specifically, introducing additional variables $T = \{t_1, \ldots, t_m\}$ and $Z = \{z_1, \ldots, z_m\}$ to represent R and F_c in terms of $(\mathbf{x}, \mathbf{z})$ and $(\mathbf{x}, \mathbf{t})$, we first compute a function $Y_4(\mathbf{y}, \mathbf{t}, \mathbf{z})$ such that $Y_4(\mathbf{y}, \mathbf{t}, \mathbf{z}) = 1$ if and only if for each $j \in \{1, \ldots, m\}$, $z_j = 1$ for all $j \notin Y_l$ and $z_j = t_j + y_j$ for all $j \in Y_l$. Y_4 is given by $Y_4(\mathbf{y}, \mathbf{t}, \mathbf{z}) = \prod_{j \in Y_l} (z_j \equiv (t_j + y_j)) \prod_{j \notin Y_l} z_j$. Then $L(\mathbf{x}, \mathbf{y}) = 1$ if and only if $\mathbf{x} \in M(\hat{c})$ and there exists $(\mathbf{t}, \mathbf{z})$ such that $R(\mathbf{x}, \mathbf{z}) = F_c(\mathbf{x}, \mathbf{t}) = Y_4(\mathbf{y}, \mathbf{t}, \mathbf{z}) = 1$. Then by definition, $l(\mathbf{y}) = 1$ if and only if $L(\mathbf{x}, \mathbf{y})$ for all $\mathbf{x} \in M(\hat{c})$. Note that, unlike REDUCE, $l(\mathbf{y})$ may be identically zero. This is the case where there is no feasible cube for $\hat{c}$.

Now, as a variation of Theorem 3.5.4, it is claimed that if $l(\mathbf{y})$ is not identically zero, then a minterm $\mathbf{y}$ with the minimum number of 1's such that $l(\mathbf{y}) = 1$ is given by a shortest path connecting a 1 leaf to the root of a BDD of $l(\mathbf{y})$. Hence, the smallest cube $c^*(p)$ containing both c and p such that $\mathcal{F} - \{c, p\} \cup \{c^*(p)\}$ is obtained by computing $l(\mathbf{y})$ for a cube $\hat{c}$, followed by performing a shortest path algorithm.

The restriction that $c^*(p)$ must cover p brings another efficiency; once there is no such $c^*(p)$ for p at some time during the expansion of c, then there is no hope that $c^*(p)$ exists in the future. Thus we mark a row of C corresponding to p if $c^*(p)$ does not exist, so that a function $l(\mathbf{y})$ for p is never computed for the rest of the procedure. We note if $c^*(p)$ is not restricted to cover p, then this does not hold for relations in general. The following example illustrates the situation.

Example 3.5.3 *A relation R and a representation $\mathcal{F}$, compatible with the relation, are given in Table 3.6. Suppose c is being expanded. There is no cube $c^*(p) \supseteq c$ such that $\mathcal{F} - \{c, p\} \cup \{c^*(p)\}$ is compatible, and thus p cannot be eliminated. On the other hand, q is eliminated from $\mathcal{F}$ if c is expanded to $c^{(1)} = [110 \mid 101]$. Then if $c^{(1)}$ is further expanded to $c^{(2)} = [112 \mid 101]$, $\{c^{(2)}\}$ is a*

compatible representation and thus p is eliminated. Note that the expansion from c to $c^{(2)}$ *is not valid if q is present in* $\mathcal{F}$.

Representation $\mathcal{F}$			Relation R	
cube	Input	Output	$\mathbf{x} \in B^3$	$\mathbf{y} \in B^3$
c	110	001	110	011, 101
p	121	101	101	000, 101
q	112	010	111	111, 001, 101

Table 3.6: Example of Expansion for a Boolean Relation

We have seen how to determine the direction of expansion for c in MFC. However, $c^*(p)$ might not exist for any $p \in \mathcal{F} - \{c\}$. In this case, an operation EG is invoked. If, in addition, C is empty, then another operation GREEDY is called. Otherwise, as Espresso does, the operation chooses a single column not in $\mathcal{R}$ or $\mathcal{L}$ with the maximum column count in C. Then a set of columns obtained by either MFC or EG are included to $\mathcal{R}$. Now C might have the rows that have a zero in every column not in $\mathcal{R}$ or $\mathcal{L}$. This means that the cubes of $\mathcal{F} - \{c\}$ corresponding to these rows are covered by c if all the columns in $\mathcal{R}$ are raised. Thus these rows are eliminated by ELM2.

The operation ESSENTIAL finds essential columns. A column $j \notin \{\mathcal{R} \cup \mathcal{L}\}$ is essential if there is no feasible cube for a cube $\hat{c}$ in which all the columns of $\mathcal{R} \cup \{j\}$ are raised. This is checked by computing $l(\mathbf{y})$ for $\hat{c}$, and seeing if l is identically zero. Once a set of essential columns are obtained, these are included in $\mathcal{L}$. If there is a row in C that has a 1 in one of the columns which have been included in $\mathcal{L}$, then there is no hope that the cube corresponding to the row is covered by expanding c. Therefore ELM1 eliminates such rows of C.

In case C becomes empty but some columns are not in either $\mathcal{R}$ or $\mathcal{L}$, GREEDY is invoked, where each such column of the input part is examined if the column is essential. If the column is not essential then it is put in $\mathcal{R}$, otherwise included in $\mathcal{L}$. Then for a cube $\hat{c}$ which has all the columns of $\mathcal{R}$ raised, we compute the *largest* feasible cube for $\hat{c}$, i.e. the feasible cube with the maximum number of 1's in the output part. Such a cube is obtained by computing the longest path to the 1 leaf of a BDD for $l(\mathbf{y})$ of $\hat{c}$. Namely, the j-th column of the output part such that $j \notin \mathcal{L}$ is included in $\mathcal{L}$ if there is a 0-edge incident with a node of the BDD for y_j in the longest path. The rest of the columns are put in $\mathcal{R}$.

Finally all the columns in $\mathcal{R}$ are raised in c by RAISE and the characteristic function of the

new representation is computed. The following theorem guarantees that at the end of EXPAND1, c is prime relative to $\mathcal{F}$.

Theorem 3.5.5 *Suppose EXPAND1 expands c to c^+. Then c^+ is prime relative to $\mathcal{F} - \{c\} \cup \{c^+\}$.*

Proof: Suppose the contrary that there exists a cube $c^{++} \supset c^+$ for which $\mathcal{F} - \{c\} \cup \{c^{++}\}$ is compatible. Let j be any part at which c^+ is lowered and c^{++} is raised. Then j was added to $\mathcal{L}$ either by ESSENTIAL or GREEDY. If j is in the output part, then it was not included by GREEDY since GREEDY chooses the largest output part among all the possible feasible cubes. Therefore, j must be an essential column at the time it was included to $\mathcal{L}$. Let $\hat{c}$ be the cube for which the existence of feasible cubes was examined for j. Note that $M(c^{++}) \supseteq M(\hat{c})$. For a cube $p = [I(\hat{c}) \mid O(c^{++})]$, the image of any minterm $\mathbf{x} \in M(\hat{c})$ by $\mathcal{F} - \{c\} \cup \{p\}$ is equal to that by $\mathcal{F} - \{c\} \cup \{c^{++}\}$. Since there was no feasible cube for $\hat{c}$ and $p \supseteq \hat{c}$, there exists a minterm $\mathbf{x} \in M(p)$ such that the image of $\mathbf{x}$ by $\mathcal{F} - \{c\} \cup \{p\}$ is not a member of $r(\mathbf{x})$. Thus $\mathcal{F} - \{c\} \cup \{c^{++}\}$ is incompatible, which is a contradiction. ■

As with REDUCE, it is claimed that there is a case where a relatively prime cube in $\mathcal{F}$ may become non-prime by expanding another cube of $\mathcal{F}$. Therefore, we iterate the entire procedure, until no cube is further expanded. At the end of EXPAND, we obtain a compatible representation in which every cube is prime relative to the representation.

3.5.6 IRREDUNDANT

In IRREDUNDANT, we produce a compatible representation in which every cube is irredundant. Specifically, the procedure takes one cube c at a time and checks if the representation $\mathcal{F} - \{c\}$ is compatible. If this is the case, c is removed.

The result of the procedure depends on the order of the cubes; the procedure processes them in decreasing order of size. However, since IRREDUNDANT is always applied as a successor of EXPAND which also sorts the cubes in the order of decreasing size and since EXPAND is iterated until no cube changes, the cubes are already sorted when given as input to IRREDUNDANT. Thus we do not sort them in this routine. IRREDUNDANT is a special case of REDUCE. Thus, as with Example 3.5.2, an irredundant cube in $\mathcal{F}$ may become redundant once another cube is modified. Thus we iterate this procedure, until no cube is removed. Note that the decreasing order of the cubes is preserved even if some cubes are removed. Note that the representation still might have a proper subset that is also compatible, as seen in Example 3.2.1, and thus the procedure does not guarantee the irredundancy of the resulting representation.

3.6 Experimental Results

The proposed procedure has been implemented in the program called GYOCRO. The system computes an initial representation if not given externally. Once the initial representation is verified to be compatible, the proposed procedure is applied. We use the same data structure as ESPRESSO-MV [45] to represent sum-of-products expressions. In the EXPAND procedure, employing the techniques introduced in [45], we have implemented the proposed algorithm by directly using a representation $\mathcal{F}$ instead of a covering matrix.

For comparison with existing techniques, several examples of Boolean relations were tried. The results are compared with the other two approaches [22, 53] developed for Boolean relations, where the programs for both procedures were provided to the authors. Table 3.7 shows the number of the product terms and CPU time (seconds) measured on a DECstation 5000/240 for all three methods. **Exact** is the exact procedure [53], **Herb** is a heuristic approach proposed in [22]. **GYOCRO** is the proposed approach. GYOCRO performs quite well both in CPU time and results. In fact, among the 18 examples for which the exact minimizer worked, the proposed procedure achieved optimum solutions for 13 examples.

In order to speed up the proposed method, we tried a modification. In the original procedure, each of the REDUCE, EXPAND, and IRREDUNDANT procedures is iterated until no cube of the representation is changed. This is because we want to guarantee that every cube produced by these procedures is maximally reduced, relatively prime, and irredundant respectively. We tried another way where each procedure is exited after a single sweep. In this case, some cubes of the final representation may not be relatively prime or irredundant. However, every cube is guaranteed to be a c-prime since the cube was made prime relative to some representation compatible with the relation.

The results of the modified procedure are shown in the last column (**GYOCRO-I**) of Table 3.7. The CPU time was improved roughly by a factor of two and the results were precisely the same. For medium size problems, where exact results are not known, GYOCRO appears to be quite effective, overcoming the problem of getting stuck too early that is common with greedy procedures such as Herb; for example, compare the results for gr or b9. The total time taken for the largest example completed, int15, has 24 inputs and 14 outputs, taking about 10 minutes. This is not an unacceptable time for this size of problem. On the other hand, for ib with 48 inputs and 17 outputs, GYOCRO could not complete due to shortage of memory. These two problems give a good indication of the limitations of GYOCRO in terms of size and speed.

Name	In	Out	Exact		Herb		GYOCRO		GYOCRO-I	
			Terms	Time	Terms	Time	Terms	Time	Terms	Time
int1	4	3	5	15.9	8	0.2	5	0.1	5	0.1
int5	4	3	7	0.2	8	0.6	7	0.2	7	0.1
int10	6	4	25	42446.3	32	3.6	25	1.9	25	1.2
c17b	5	3	7	18.2	7	0.4	7	0.2	7	0.1
c17i	5	3	13	0.9	14	1.7	15	0.6	15	0.4
she1	7	3	6	70.3	9	96.5	6	0.7	6	0.4
she2	5	5	time out		14	18.0	10	2.6	10	1.6
she3	7	4	time out		10	358.5	9	2.1	9	1.4
she4	5	6	time out		27	22.4	20	4.6	20	3.2
gr	15	11	*	*	126	400.4	53	182.0	53	78.3
b9	16	5	*	*	452	1439.3	270	207.1	270	116.1
int15	24	14	*	*	145	421.8	131	526.6	131	273.1
ib	48	17	*	*	out of memory		out of memory		out of memory	

* : the method has not been applied for the example.

Table 3.7: Experimental Results

3.7 Concluding Remarks

The minimization problem of multiple-valued relations naturally arises in many contexts. We have proposed a heuristic procedure which achieves the minimizations, where a relation is represented by its characteristic function using an MDD. The procedure is based on a paradigm for two level minimization of ordinary functions, in which a solution is obtained through an iteration of expand and reduce procedures. We have described some special properties of relations that do not hold for functions. These properties are easily handled through MDD manipulations. Unlike greedily expanding or reducing a cube, reduction and expansion are achieved by computing the set of cubes that satisfy a property that we want to obtain and by directly choosing the best among the set. The proposed procedure has been implemented in the program GYOCRO. The results are encouraging in the sense that, for those examples where we know the minimum solution, our heuristic minimizer reproduces the result most of the time or comes very close. On large examples where the exact minimizer can not complete, our method outperforms the other heuristic Boolean relation minimizer. Computing times and the size of problems that can be completed are reasonable.

Chapter 4

Permissible Behaviors for Finite State Machines

4.1 Introduction

4.1.1 Overview

In Chapter 2, we considered the problem of computing a set of permissible combinational logic behaviors. The model employed there is a system of interacting components, where the connections of the components are acyclic and each component is combinational; it implements a Boolean function. The specification for the behavior of the system is provided as a set of Boolean functions, which is represented by a Boolean relation. It was shown that a set of permissible functions can be computed and represented by a Boolean relation. We showed how a minimal-cost representation can be found for a given relation in Chapter 3.

In this and the following chapters, an analogous investigation is made for *sequential logic behaviors*. We consider a system where each component is a completely specified deterministic finite state machine. In other words, each component implements a single sequential logic behavior, in the sense that for a given finite sequence of signals (or *events*) defined over the inputs of the component, the finite state machine associated with the component provides exactly one sequence with the same length defined over the outputs of the component. Note that a component with a single state is allowed, and thus purely combinational components may be included. We consider synchronous systems only. Specifically, we assume an existence of global time steps, where each time step is an instant moment with no interval, and is generated by a global timing controller

called a *clock*. A component of a system *operates* at all and only the time steps. Thus, operations take place simultaneously over all the components at each time step. An operation of a component consists of two actions; one is that of taking an input event, i.e. a set of values defined for the input variables of the component, while the other action is a generation of an output event. Each output event is generated as a function of a given input event and internal state of the finite state machine associated with the component. A component takes these two actions at the same time for each operation. Therefore, at each time step, every component simultaneously takes an input event, and immediately generates the corresponding output event. The output events are communicated either to the global outputs of the system or to other connected components.

For a synchronous system, the behavior of the entire system can be also modeled by a completely specified deterministic finite state machine. Namely, for each finite sequence defined for the global inputs of the system, the system generates exactly one sequence of the global outputs. As an analogy to the combinational case, we assume that more than one output sequence may be allowed for a given sequence of the global inputs. In other words, there are in general a set of sequential behaviors allowed to be implemented for the entire system, and we say the system *meets* the specification if it implements one of those behaviors. Thus, we assume that the specification for the behavior of the entire system is given by a non-deterministic finite state machine. Non-determinism allows one to represent a set of behaviors in a single machine. Specifically, for a given present state and input, a non-deterministic finite state machine may go to different next states and/or have different outputs. Therefore, for a given input sequence, the system may allow more than one output sequence, and we provide a specification so that each of the output sequences is regarded as a valid one. Of course, the final implementation generates only one output sequence, but the specification is given as a set of behaviors, which provides more flexibility for optimization.

Under these assumptions, we consider the problem of finding the set of permissible sequential behaviors at a given component of a system. Namely, for a given synchronous system of interacting finite state machines and for a given specification, we find the complete set of permissible behaviors at a particular component of the system. A behavior is said to be permissible if the resulting behavior of the entire system meets the specification, where we assume that the behaviors of the rest of the components are fixed. This is an analogous to the problem of finding the maximum set of permissible Boolean functions for combinational logic behaviors. The resulting set of permissible behaviors is then used in some optimal search procedure for a best choice. The optimization problem, where the optimality is defined in terms of the number of states, will be studied in Chapter 5.

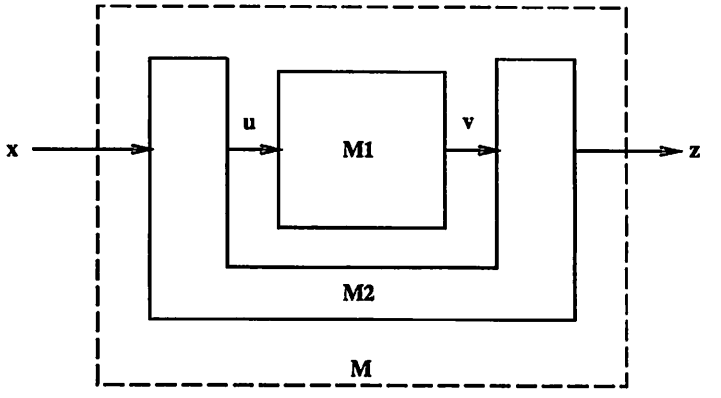

Figure 4.1: Interaction between Two Machines

The problem of finding permissible behaviors can be viewed as an interaction between two finite state machines, shown in Figure 4.1, where M_1 is the machine associated with the component being optimized, M_2 represents the behavior of the rest of the system, and M gives the specification. Our problem is to find the set of behaviors that can be realized at M_1 so that the resulting behavior made of M_1 and M_2 meets the specification M.

4.1.2 Related Problems

There are several problems/applications that can be viewed as a variation of this problem. One such problem is to find the set of permissible behaviors of the outside component M_2 when the internal component M_1 and the specification M are given. This problem can be solved in exactly the same way as the original problem, since the interaction shown in Figure 4.1 can be redrawn as shown in Figure 4.2-(a), which yields to the same picture of Figure 4.1 by modifying M_1 so that the global input X and the global output Z pass through M_1. This is illustrated in Figure 4.2-(b). Such a problem arises in rectification problems[20, 60], where the designer wants to change the functionality of a design, perhaps because of an engineering change, by attaching a small block of logic (M_2) external to the original circuitry.

Another related problem is a supervisory control problem for discrete event processes[42]. The problem is that for a given generator of discrete events (M_1) and a specification on the generated events (M), we observe the events provided by the generator and control them (M_2) by feeding

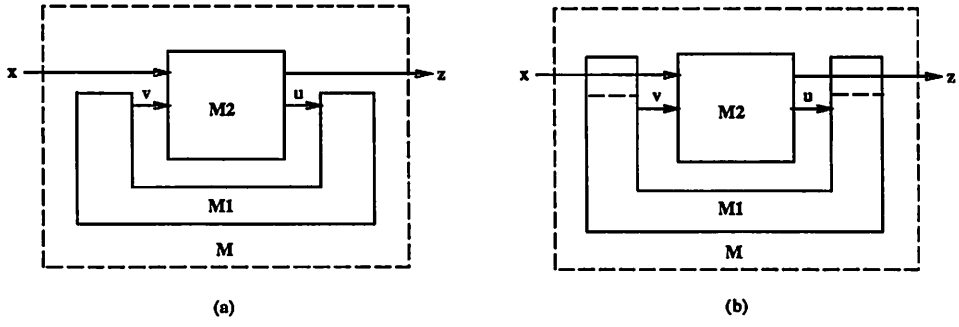

Figure 4.2: Rectification Problem

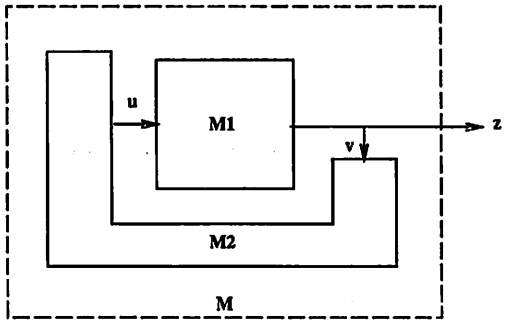

Figure 4.3: Supervisory Control Problem

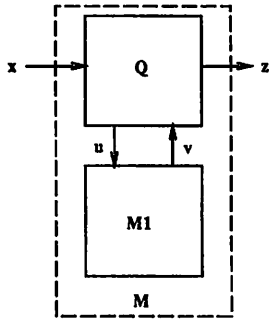

Figure 4.4: FSM Boolean Division

control events to the generator so that the resulting output events meet the specification. If the generator and the supervisor are synchronizing, this problem can be deemed as a variation of our problem, as shown in Figure 4.3, where M_1 is the generator, M_2 is the supervisor, and V and Z are identical while M_1 may be a non-deterministic machine.

Finally, our core problem includes the "division" problem for finite state machines ; given an initial machine M and a "divisor" M_1, find the quotient Q of the two as given in Figure 4.4. This is the central problem of factorization and decomposition of finite state machines, and is similar to the rectification problem above.

4.1.3 Related Work

The problem of finding a set of permissible behaviors for interacting finite state machines has been studied previously [18, 29, 43, 56, 59]. Most results, following analogies from the combinational logic case, are based on *don't care sequences*. For example, input don't care sequences are sequences of inputs of M_1 which never occur. Output don't care sequences are sequences of outputs of M_1, defined for given sequences of the global inputs, so that the resulting global outputs meet the specification M. Approaches based on don't care sequences have several limitations. First, since the inputs of M_1 (outputs of M_2) may depend on the outputs of M_1, the sequences that appear at the inputs of M_1 can be controlled by changing the functionality of M_1, which may then define a different set of don't care sequences. Thus the previous work either makes an assumption on the topology of M_1 and M_2, such as cascaded machines where M_2 is independent of M_1, restricts to computing only a subset of don't care sequences, or computes input don't care sequences and output don't care sequences separately[18, 29, 43, 59]. Furthermore, due to complexity, often the sequences are only partially considered, up to a certain, typically small, length. As a result, even though one finds the *best* implementation among the set of permissible behaviors computed, there is no guarantee that it is best among all permissible behaviors.

In this chapter, we ask if it is possible to compute and represent easily the complete set of permissible behaviors at M_1. The answer is yes and it can be represented by a single non-deterministic finite state machine, which we call the *E-machine*. The result is obtained by a simple fixed point computation which provides the transition relation of the E-machine. The procedure has been implemented and initial results for the E-machines for a small set of artificial examples of moderate size have been derived.

4.2 Terminology

In this section, we define basic terminology used throughout this and the following chapters.

Definition: Finite State Machine

A **finite state machine** is a 5-tuple (I, O, S, T, r), where I is the set of input variables, O is the set of output variables, S is the set of states, $T : S \times B^{|I|} \times B^{|O|} \times S \rightarrow B$ is a characteristic function with $B = \{0, 1\}$, and r is an element of S. The machine stays in exactly one state, say s_p, of S at any given time. The machine takes as input a minterm $\mathbf{i} \in B^{|I|}$, by which a **transition** is enabled. A transition at a state s_p consists of a pair made of a state and an output minterm, $(s_n, \mathbf{o}) \in S \times B^{|O|}$, which indicates that the machine moves from state s_p to state s_n and outputs $\mathbf{o}$. It is assumed that a transition takes no time. The function T defines the valid transitions of the machine, i.e. $T(s_p, \mathbf{i}, \mathbf{o}, s_n) = 1$ if and only if the transition $(s_n, \mathbf{o})$ can be enabled at the state s_p by the input $\mathbf{i}$. There exists at least one (possibly more) valid transition for each state s_p and input $\mathbf{i}$. The state r, defined in the 5-tuple, designates the state at which the machine stays initially, and is called the **reset state** or the **initial state**.

The function $T : S \times B^{|I|} \times B^{|O|} \times S \rightarrow B$ is called the transition relation of the finite state machine. For a given state $s_0 \in S$ and a sequence $\sigma_i = (\mathbf{i}_0, \ldots, \mathbf{i}_{t-1})$ of the input minterms, where $\mathbf{i}_j \in B^{|I|}$ for each $j = 0, \ldots, t-1$ and $t \geq 0$,[1] there always exists at least one sequence of output minterms $\sigma_o = (\mathbf{o}_0, \ldots, \mathbf{o}_{t-1})$ and a sequence of states $\sigma_s = (s_0, \ldots, s_t)$ with the property that $T(s_j, \mathbf{i}_j, \mathbf{o}_j, s_{j+1}) = 1$ for all $j = 0, \ldots, t-1$. Such a sequence of output minterms (a sequence of states, respectively) is called an output sequence (state transition) defined at s_0 by σ_i. We say that a pair of sequences (σ_i, σ_o) is *realized* at the state s_0 in the machine. In particular, if s_0 is set to the reset state r, we say that the sequence σ_i *leads* the machine to the state s_t with the output sequence σ_o. In this case, we also say that the pair (σ_i, σ_o) is *realized* by the machine. The integer t of the sequence σ is called the *length* of σ and is denoted by $|\sigma|$.

Definition: Deterministic Finite State Machine

A finite state machine (I, O, S, T, r) is said to be **deterministic** if there exists a pair of functions, $\lambda : S \times B^{|I|} \rightarrow \{0, 1, *\}^{|O|}$ and $\delta : S \times B^{|I|} \rightarrow S \cup \{*\}$, referred to as the **output function** and the **transition function** respectively, such that for all $(s_p, \mathbf{i}, \mathbf{o}, s_n) \in S \times B^{|I|} \times B^{|O|} \times S$,

[1] Such a sequence σ_i is called an input sequence. If $t = 0$, the sequence is null.

$T(s_p, \mathbf{i}, \mathbf{o}, s_n) = 1$ if and only if the following two conditions hold:

$$(1) \quad \mathbf{o} \subseteq \lambda(s_p, \mathbf{i}) \text{ and}$$
$$(2) \quad \delta(s_p, \mathbf{i}) = * \text{ or } \delta(s_p, \mathbf{i}) = s_n,$$

where for $\tilde{\mathbf{o}} = \lambda(s_p, \mathbf{i})$, $\mathbf{o} \subseteq \tilde{\mathbf{o}}$ designates that for each output variable j, $\tilde{o}_j \neq *$ implies that $o_j = \tilde{o}_j$.

A deterministic finite state machine is said to be *completely specified* if for all $(s_p, \mathbf{i}) \in S \times B^{|I|}$, $\lambda(s_p, \mathbf{i}) \in B^{|O|}$ and $\delta(s_p, \mathbf{i}) \neq *$. Otherwise, the machine is said to be *incompletely specified.*

Intuitively, the function $\lambda(s_p, \mathbf{i})$ represents the output value of the machine obtained when it takes $\mathbf{i}$ as input at state s_p, while $\delta(s_p, \mathbf{i})$ designates the corresponding next state. Furthermore, if the value of $\lambda(s_p, \mathbf{i})$ for an output variable j is $*$, which denotes a *don't care*, then it means that the machine can output any value of B, i.e. 0 or 1, at output j. Similarly, if $\delta(s_p, \mathbf{i}) = *$, then the machine can move to any state of S when it takes the input $\mathbf{i}$ at the state s_p.

By definition, the valid transitions of a deterministic finite state machine can be represented using the functions λ and δ given above. Therefore, we may represent the machine by a 6-tuple $(I, O, S, \lambda, \delta, r)$. Note that at each state, the output sequence and the state transition defined at the state by a given input sequence are unique for a completely specified deterministic machine.

A deterministic finite state machine $(I, O, S, \lambda, \delta, r)$ is called a *Moore machine*[38] if for each state $s_p \in S$, there exists a unique $\mathbf{o} \in \{0, 1, *\}^{|O|}$ such that for all $\mathbf{i} \in B^{|I|}$, $\lambda(s_p, \mathbf{i}) = \mathbf{o}$. Otherwise, it is called a *Mealy machine*[35]. Note that the function λ of a Moore machine depends only on the states S and not on the inputs $B^{|I|}$.

A finite state machine that is not deterministic is said to be *non-deterministic* .

Definition: Reachable States

Given a completely specified deterministic finite state machine $(I, O, S, \lambda, \delta, r)$, a state $s \in S$ is said to be **reachable** if there exists an input sequence which leads the machine to s.

A state that is not reachable is said to be *unreachable.*

Definition: Equivalent States

Given a completely specified deterministic finite state machine $(I, O, S, \lambda, \delta, r)$, a pair of states $(s, \tilde{s}) \in S \times S$ is said to be **equivalent** if for all input sequences, say σ, the output sequence defined by σ at s is identical with that defined at $\tilde{s}$.

A set of states of S is said to be **equivalent** if every pair of states in the set is equivalent.

The set of states S of a completely specified deterministic finite state machine can be uniquely divided into a set of disjoint classes, where each class consists of maximal number of equivalent states. Each such class is called an *equivalence class*.

Definition: Equivalent Machines

Two completely specified deterministic machines $M = (I, O, S, \lambda, \delta, r)$ and $\tilde{M} = (I, O, \tilde{S}, \tilde{\lambda}, \tilde{\delta}, \tilde{r})$ are **equivalent** if for all input sequences, say σ, the output sequence define at r by σ in M is identical with that defined at $\tilde{r}$ by σ in $\tilde{M}$.

In this chapter, we discuss the *behaviors* of finite state machines . Intuitively, a behavior between the input variables I and the output variables O is the set of pairs of input and output sequences realized by a completely specified deterministic finite state machine with the input I and the output O. In this sense, we say that the machine *represents* the behavior. Although this intuitive definition will suffice to understand the chapter , we provide a formal definition of a behavior using the notion of finite automata.

Definition: Finite Automaton

A deterministic **finite automaton** is a 5-tuple (X, S, δ, F, r), where X is the set of input variables, S is the set of states, $\delta : S \times B^{|X|} \rightarrow S$ is the transition function, $F \subseteq S$ is the set of final states, and $r \in S$ is the reset state.

A finite automaton (X, S, δ, F, r) has a one-to-one correspondence with a completely specified deterministic finite state machine with a single output o, $(X, o, S, \lambda, \delta, r)$, which has the identical transition function δ, where the output function $\lambda(s, \mathbf{x}) = 1$ if and only if $\delta(s, \mathbf{x}) \in F$ in the original automaton. Hence, terminology, defined for finite state machines, will be used for finite automata as well. A sequence on X which leads the automaton to a state in F is said to be *accepted* by the automaton. We now define a behavior as follows.

Definition: Behavior

Given a set of input variables I and a set of output variables O, a behavior between I and O is a set of pairs of input and output sequences, $\mathcal{B} = \{(\sigma_i, \sigma_o) \mid |\sigma_i| = |\sigma_o|\}$, which satisfies the following conditions:

1. Completeness:

   For an arbitrary sequence σ_i on I, there exists a unique pair in $\mathcal{B}$ whose input sequence is

equal to σ_i.

2. Prefix closed:
   For an arbitrary pair $p = (\sigma_i, \sigma_o) \in \mathcal{B}$, where $\sigma_i = (\mathbf{i}_0, \ldots, \mathbf{i}_k)$ and $\sigma_o = (\mathbf{o}_0, \ldots, \mathbf{o}_k)$ with $k > 0$, let $\tilde{\sigma}_i = (\mathbf{i}_0, \ldots, \mathbf{i}_{k-1})$ and $\tilde{\sigma}_o = (\mathbf{o}_0, \ldots, \mathbf{o}_{k-1})$. Then $(\tilde{\sigma}_i, \tilde{\sigma}_o) \in \mathcal{B}$.

3. Regularity:
   For an arbitrary pair $p = (\sigma_i, \sigma_o) \in \mathcal{B}$, where $\sigma_i = (\mathbf{i}_0, \ldots, \mathbf{i}_k)$ and $\sigma_o = (\mathbf{o}_0, \ldots, \mathbf{o}_k)$ with $k \geq 0$, let $\sigma(p)$ be a sequence on $I \cup O$ defined as $\sigma(p) = (\mathbf{i}_0\mathbf{o}_0, \ldots, \mathbf{i}_k\mathbf{o}_k)$. Then there exists a deterministic finite automaton with inputs $I \cup O$ which accepts all and only the sequences of the set given by $\{\sigma(p) \mid p \in \mathcal{B}\}$.

For each pair (σ_i, σ_o) of a behavior, we say that (σ_i, σ_o) is *realized* by the behavior.

For a non-deterministic finite state machine, there might exist more than one valid transition for some state and an input. In this sense, we can regard that a non-deterministic machine represents a set of behaviors represented by a set of completely specified deterministic machines. We call each such behavior a *contained behavior*.

Definition: Contained Behavior

Given a finite state machine $T = (I, O, S, T, r)$, a behavior between I and O is said to be **contained** in T if every pair of input and output sequences of the behavior is realized by T.

By definition, if T is a completely specified deterministic machine, there is a unique behavior contained in it.

4.3 The Problem and Assumptions

Consider the case of two interacting finite state machines shown in Figure 4.1. M_1 takes input u and outputs v, and M_2 takes input x and v and outputs u and z. M is a finite state machine with input x and output z. It represents behaviors allowed for the entire system composed of M_1 and M_2.

Specifically, let $M = (X, Z, S, T_M, r)$ and $M_2 = (X \cup V, U \cup Z, S_2, \lambda_2, \delta_2, r_2)$ be given. Assume that M is a non-deterministic machine while M_2 is a completely specified deterministic machine. By allowing non-determinism on M, we can specify a set of behaviors, rather than a single behavior, for the entire system. In the sequel, we often discuss which outputs *can* be obtained

from M at a particular state and a particular input. For this purpose, introduce two functions $\Lambda : 2^S \times B^{|X|} \rightarrow 2^{B^{|Z|}}$ and $\Delta : 2^S \times B^{|X|} \times B^{|Z|} \rightarrow 2^S$ defined as follows:

$$\begin{aligned} \Lambda(s^*, \mathbf{x}) &= \{\mathbf{z} \in B^{|Z|} \mid \exists(\tilde{s}, s) \in s^* \times S : T_M(\tilde{s}, \mathbf{x}, \mathbf{z}, s) = 1\} \\ \Delta(s^*, \mathbf{x}, \mathbf{z}) &= \{s \in S \mid \exists \tilde{s} \in s^* : T_M(\tilde{s}, \mathbf{x}, \mathbf{z}, s) = 1\}, \end{aligned}$$

where s^* is a subset of states of M and 2^S is the power set of S. Intuitively, $\Lambda(s^*, \mathbf{x})$ defines the set of values that can be output by at least one state of s^* in M under the input $\mathbf{x}$. Similarly, $\Delta(s^*, \mathbf{x}, \mathbf{z})$ is the set of next states to which M can move from at least one state of s^* under the input $\mathbf{x}$ and output $\mathbf{z}$.

Note that the output of the output function λ_2 of M_2 is a pair of minterms $(\mathbf{u}, \mathbf{z}) \in B^{|U|} \times B^{|Z|}$. In the sequel, we also represent λ_2 using two functions $\lambda_2^{(u)} : S_2 \times B^{|X \cup V|} \rightarrow B^{|U|}$ and $\lambda_2^{(z)} : S_2 \times B^{|X \cup V|} \rightarrow B^{|Z|}$ such that $\lambda_2(s_2, \mathbf{xv}) = (\lambda_2^{(u)}(s_2, \mathbf{xv}), \lambda_2^{(z)}(s_2, \mathbf{xv}))$.

We are interested in finding a set of behaviors represented by finite state machines permissible at M_1. From these, we can derive circuit implementations, but some might have *combinational loops*. A combinational loop is a topological loop made of gates and wires, where no latch or flip-flop is included. Note that in general, if both M_1 and M_2 are Mealy machines, since the outputs depend on the inputs, there might exist a variable of V which depends on a variable of U in M_1, while the variable of U depends on the variable of V in M_2. Although a circuit with a combinational loop might be acceptable, it is known that such a circuit could cause an unexpected problem called *race-around condition* [37], and thus it is still very rare to find such a circuit in practical synchronous digital designs. Therefore, we exclude this situation and consider only the machines that can be implemented at M_1 without introducing combinational loops. Specifically, we define implementable machines as follows.

Definition: Implementable Finite State Machine

Given $M_2 = (X \cup V, U \cup Z, S_2, \lambda_2, \delta_2, r_2)$, a completely specified deterministic finite state machine $(U, V, S_1, \lambda_1, \delta_1, r_1)$ is said to be **implementable** at M_1 if there exists a pair of circuit implementations of M_1 and M_2 respectively such that no combinational loop is created by connecting them together at U and V.

Note that if M_2 is a Moore machine, then implementability imposes no restriction, since every machine defined at M_1 is implementable. We discuss implementability in more detail in Section 4.7 and provide a necessary and sufficient condition under which M_1 is implementable. We say a behavior between U and V is implementable at M_1 if there exists an implementable machine

at M_1 representing the behavior. Note that for an implementable machine M_1, and for an arbitrary sequence of $B^{|X|}$, say $\sigma = (\mathbf{x}_0, \ldots, \mathbf{x}_t)$, if we denote by $(s_1, s_2) \in S_1 \times S_2$ the pair of states of M_1 and M_2 led to by $(\mathbf{x}_0, \ldots, \mathbf{x}_{t-1})$, then $\mathbf{x}_t$ defines a unique pair $(\mathbf{u}, \mathbf{v}) \in B^{|U|} \times B^{|V|}$ such that $\mathbf{u} = \lambda_2^{(u)}(s_2, \mathbf{x}_t\mathbf{v})$ and $\mathbf{v} = \lambda_1(s_1, \mathbf{u})$.

For an implementable machine $M_1 = (U, V, S_1, \lambda_1, \delta_1, r_1)$, we define the product machine of M_1 and M_2, denoted by $M_1 \times M_2$, as a completely specified deterministic finite state machine $(X, Z, S_p, \lambda_p, \delta_p, r_p)$ such that $S_p = S_1 \times S_2$, $r_p = (r_1, r_2)$, and for a state $(s_1, s_2) \in S_p$ and a minterm $\mathbf{x} \in B^{|X|}$, $\lambda_p((s_1, s_2), \mathbf{x}) = \mathbf{z}$ if and only if there exist $\mathbf{u} \in B^{|U|}$ and $\mathbf{v} \in B^{|V|}$ such that $\lambda_1(s_1, \mathbf{u}) = \mathbf{v}$ and $\lambda_2(s_2, \mathbf{xv}) = (\mathbf{u}, \mathbf{z})$. Similarly, $\delta_p((s_1, s_2), \mathbf{x}) = (\tilde{s}_1, \tilde{s}_2)$ if and only if there exist $\mathbf{u} \in B^{|U|}$ and $\mathbf{v} \in B^{|V|}$ such that $\lambda_1(s_1, \mathbf{u}) = \mathbf{v}$, $\lambda_2^{(u)}(s_2, \mathbf{xv}) = \mathbf{u}$, and $(\delta_1(s_1, \mathbf{u}), \delta_2(s_2, \mathbf{xv})) = (\tilde{s}_1, \tilde{s}_2)$.

We now define a permissible machine as follows.

Definition: Permissible Finite State Machine

Given $M = (X, Z, S, T_M, r)$ and $M_2 = (X \cup V, U \cup Z, S_2, \lambda_2, \delta_2, r_2)$, a completely specified deterministic finite state machine $M_1 = (U, V, S_1, \lambda_1, \delta_1, r_1)$ is said to be **permissible** if M_1 is implementable and the behavior of $M_1 \times M_2$ is contained in M.

The behavior represented by a permissible machine is called a *permissible behavior*. Our objective is to find the complete set of permissible behaviors at M_1. Note that we are not interested in finding the complete set of permissible machines at M_1 since for a given behavior, there are in general an infinite number of machines representing the behavior. Thus we just need enough machines which represent the complete set of permissible behaviors. We show that the complete set of permissible behaviors can be computed and represented by a single finite state machine, which we call the E-machine.

4.4 Prime Machines

To show how the complete set of permissible behaviors can be computed and represented by a single finite state machine, the key first step is to represent the behavior of a machine implementable at M_1 using a special deterministic machine called a prime machine. In this section, we discuss the notion of prime machines.

Given $M = (X, Z, S, T, r)$ and $M_2 = (X \cup V, U \cup Z, S_2, \lambda_2, \delta_2, r_2)$, consider an implementable machine $M_1 = (U, V, S_1, \lambda_1, \delta_1, r_1)$.

Definition: $\Sigma(s_1, t)$

For a state $s_1 \in S_1$ and an integer $t \geq 0$, let $\Sigma(s_1, t)$ be a subset of $S_2 \times 2^S$ such that $(s_2, s^*) \in S_2 \times 2^S$ is in $\Sigma(s_1, t)$ if and only if there exists a sequence σ_x of $B^{|X|}$ with length t such that:

1. σ_x leads $M_1 \times M_2$ to (s_1, s_2), and
2. σ_x leads M to all and only the states of s^* with the output sequence of $B^{|Z|}$ that is realized in $M_1 \times M_2$ by applying σ_x.

As a special case, we define $\Sigma(s_1, -1) = \phi$ for all $s_1 \in S_1$.

$\Sigma(s_1, t)$ consists of all possible pairs, $(s_2, s^*) \in S_2 \times 2^S$, such that M_2 and M can be led to by some global input sequence σ_x of length t with the same global output sequence σ_z, while M_1 is led to s_1. The sequence σ_z is the one given in $M_1 \times M_2$ by applying σ_x. Note that if there is no state transition realized by (σ_x, σ_z) in M, then s^* is an empty set. Note also that if there is no global input sequence of length t which leads M_1 to s_1, then $\Sigma(s_1, t)$ is empty.

Definition: $N(\tilde{s}_1, \Sigma, \mathbf{u}, s_1)$

Given $s_1 \in S_1$, $\tilde{s}_1 \in S_1$, $\mathbf{u} \in B^{|U|}$, and $\Sigma \subseteq S_2 \times 2^S$, let $N(\tilde{s}_1, \Sigma, \mathbf{u}, s_1)$ be a subset of $S_2 \times 2^S$ given by

$$\{(s_2, s^*) \in S_2 \times 2^S \mid \exists \mathbf{x} \in B^{|X|}, (\tilde{s}_2, \tilde{s^*}) \in \Sigma : \begin{array}{ll} \mathbf{u} = \lambda_2^{(u)}(\tilde{s}_2, \mathbf{xv}), & s_1 = \delta_1(\tilde{s}_1, \mathbf{u}), \\ s_2 = \delta_2(\tilde{s}_2, \mathbf{xv}), & s^* = \Delta(\tilde{s^*}, \mathbf{x}, \lambda_2^{(z)}(\tilde{s}_2, \mathbf{xv})) \end{array} \},$$

where $\mathbf{v} = \lambda_1(\tilde{s}_1, \mathbf{u})$.

Intuitively, $N(\tilde{s}_1, \Sigma, \mathbf{u}, s_1)$ defines all possible pair of "next" states of M_2 and M, (s_2, s^*), such that $M_1 \times M_2$ can move from $(\tilde{s}_1, \tilde{s}_2)$ to (s_1, s_2) and M can move from $\tilde{s^*}$ to s^* with the output $\lambda_2^{(z)}(\tilde{s}_2, \mathbf{xv})$ in a single transition for some $(\tilde{s}_2, \tilde{s^*}) \in \Sigma$, where the transition causes M_1 to move to s_1 under input minterm $\mathbf{u}$.

Definition: Prime Machine

An implementable machine $M_1 = (U, V, S_1, \lambda_1, \delta_1, r_1)$ is **prime**, with respect to M_2 and M, if for each state $s_1 \in S_1$, there exists a subset $\Sigma(s_1) \subseteq S_2 \times 2^S$ with the following property:

(1) $\forall t \geq 0 : \Sigma(s_1, t) \neq \phi \Rightarrow \Sigma(s_1, t) = \Sigma(s_1)$

(2) $\forall \mathbf{u} \in B^{|U|}, \tilde{s}_1 \in S_1 : s_1 = \delta_1(\tilde{s}_1, \mathbf{u}) \Rightarrow N(\tilde{s}_1, \Sigma(\tilde{s}_1), \mathbf{u}, s_1) = \Sigma(s_1)$

(3) $(\forall t \geq 0 : \Sigma(s_1, t) = \phi) \Rightarrow \Sigma(s_1) = \{\phi\}$

In other words, each state of a prime machine, whenever it is reached, is identified with exactly one subset of $S_2 \times 2^S$. Note that if M is a completely specified deterministic machine, then $\Sigma(s_1)$ is a set of pairs of states of M_2 and M, i.e. a subset of $S_2 \times S$. We define prime machines only for implementable machines, in order to ensure that for an arbitrary sequence of $B^{|X|}$, say σ, with length t, the pair of states of M_1 and M_2 led to by the prefix subsequence of σ with length $t-1$ is uniquely defined and there exists a unique pair $(\mathbf{u}, \mathbf{v}) \in B^{|U|} \times B^{|V|}$ such that $\mathbf{u} = \lambda_2^{(u)}(s_2, \mathbf{x}_t\mathbf{v})$ and $\mathbf{v} = \lambda_1(s_1, \mathbf{u})$, where $\mathbf{x}_t$ is the last element of σ.

Theorem 4.4.1 *For each implementable machine M_1, there exists an equivalent prime machine.*

Proof: We prove the theorem by presenting a procedure which takes as input an implementable machine $M_1 = (U, V, S_1, \lambda_1, \delta_1, r_1)$ and returns an equivalent prime machine M_1'. The procedure is shown in Figure 4.5.

The procedure first duplicates M_1, where S_1' is set to S_1, λ' and δ' are identical with λ and δ, respectively. The transitions of M_1' are then modified during the procedure. The function $E(s_1)$, used in the procedure, is defined for a state $s_1 \in S_1$. $E(s_1)$ designates the equivalent class that s_1 initially belongs to in M_1. When M_1 is copied to M_1' at the beginning of the procedure, we assume that $E(s_1)$ is associated with each state s_1 of S_1'. Furthermore, $N'(\tilde{s}_1, \Sigma, \mathbf{u}, s_1)$ is given by

$$\{(s_2, s^*) \in S_2 \times 2^S \mid \exists \mathbf{x} \in B^{|X|}, (\tilde{s}_2, \tilde{s^*}) \in \Sigma : \begin{array}{ll} \mathbf{u} = \lambda_2^{(u)}(\tilde{s}_2, \mathbf{xv}), & s_1 = \delta_1'(\tilde{s}_1, \mathbf{u}), \\ s_2 = \delta_2(\tilde{s}_2, \mathbf{xv}), & s^* = \Delta(\tilde{s^*}, \mathbf{x}, \lambda_2^{(z)}(\tilde{s}_2, \mathbf{xv})), \end{array} \},$$

where $\mathbf{v} = \lambda_1'(\tilde{s}_1, \mathbf{u})$. When a new state $\hat{s}_1$ is created in M_1', we set $\delta_1'(\hat{s}_1, \tilde{\mathbf{u}}) \leftarrow \delta_1(s_1, \tilde{\mathbf{u}})$ for each $\tilde{\mathbf{u}} \in B^{|U|}$, where s_1 is the next state of $\tilde{s}_1$ in M_1' under the input $\mathbf{u}$. Note that it is always true that the next state s_1 is a state which originally existed in M_1' when M_1 was duplicated. Therefore the state corresponding to s_1 also exists in M_1, which we denote also by s_1. Hence, by $\delta_1'(\hat{s}_1, \tilde{\mathbf{u}}) \leftarrow \delta_1(s_1, \tilde{\mathbf{u}})$, we mean $\delta_1'(\hat{s}_1, \tilde{\mathbf{u}})$ is set to the state of M_1' which corresponds to the state of M_1 given by $\delta_1(s_1, \tilde{\mathbf{u}})$. In other words, the transitions of a newly created state $\hat{s}_1$ are made identical with those defined at s_1 in M_1.

We first claim that the procedure maintains the invariance that a state s_1 of M_1' is equivalent to every state of $E(s_1)$ of M_1. Namely, for all sequences σ of $B^{|U|}$, the output sequence defined at s_1 by σ in M_1' is identical with that defined at a state of $E(s_1)$ in M_1. The invariance is trivially true in the beginning since the function E is so defined. Suppose that the invariance holds immediately

```
function prime(M_1 = (U, V, S_1, λ_1, δ_1, r_1))
    /* let M_1' = (U, V, S_1', λ_1', δ_1', r_1) */
    M_1' ← copy(M_1);
    for(each s_1 ∈ S_1'){  Σ(s_1) ← undefined;  }
    Σ(r_1) ← {(r_2, {r})}; /* r_1 ∈ S_1' */
    mark r_1;
start:
    while(there exists s̃_1 ∈ S_1' that is marked){
        for(each u ∈ B^|U|){
            /* let s_1 = δ_1'(s̃_1, u) */
            N ← N'(s̃_1, Σ(s̃_1), u, s_1); /* since s̃_1 is marked, Σ(s̃_1) is defined. */
            if(∃ŝ_1 ∈ S_1' : Σ(ŝ_1) = N and E(ŝ_1) = E(s_1))  δ_1'(s̃_1, u) ← ŝ_1;
            else if(Σ(s_1) = undefined){  Σ(s_1) ← N;  mark s_1;  }
            else{  /* create a new state ŝ_1 */
                S_1' ← S_1' ∪ {ŝ_1};
                for(each ũ ∈ B^|U|){
                    δ_1'(ŝ_1, ũ) ← δ_1(s_1, ũ);  λ_1'(ŝ_1, ũ) ← λ_1(s_1, ũ);
                }
                δ_1'(s̃_1, u) ← ŝ_1;  Σ(ŝ_1) ← N;
                E(ŝ_1) ← E(s_1);  mark ŝ_1;
            }
        }
        remove the mark of s̃_1;
    }
    for(each s_1 ∈ S_1' such that Σ(s_1) = undefined){  Σ(s_1) ← {φ};  mark s_1;  }
    if(there is a marked state)  goto start;
    return M_1';
```

Figure 4.5: Procedure to Generate a Prime Machine

before a state $\tilde{s}_1$ is processed. Consider the case where $\tilde{s}_1$ is processed for a minterm $\mathbf{u} \in B^{|U|}$. Suppose a new state $\hat{s}_1$ is created. Since the transitions defined at $\hat{s}_1$ in M_1' are identical with those of s_1 defined in M_1, $\hat{s}_1$ of M_1' is equivalent to s_1 of M_1. Since $E(\hat{s}_1)$ is set to $E(s_1)$, the invariance holds for the state $\hat{s}_1$. Also, if the state $\delta_1'(\tilde{s}_1, \mathbf{u})$ is changed from s_1 to another already existing state $\hat{s}_1$, then $E(\hat{s}_1) = E(s_1)$ holds by construction. Since the output values of $\tilde{s}_1$ do not change during the process, $\tilde{s}_1$ obtained after the process for $\mathbf{u}$ is still equivalent to a state of $E(\tilde{s}_1)$ of M_1. Thus the invariance holds. Therefore, M_1' obtained immediately after processing $\tilde{s}_1$ is equivalent to M_1.

We next claim that the procedure terminates. This is because every state is processed exactly once and a new state is created only if there is no state $\hat{s}_1$ equivalent to s_1 with $\Sigma(\hat{s}_1) = N$. Since the number of states of M_1 and the number of subsets of $S_2 \times 2^S$ are both finite, the procedure must terminate.

It follows that M_1' obtained at the end of the procedure is equivalent to M_1.

Finally, we claim that M_1' is a prime machine. The condition (2) shown in the definition of prime machines holds for M_1' since all the states of M_1' are processed and the set N used in the procedure is as defined by $N(\tilde{s}_1, \Sigma(\tilde{s}_1), \mathbf{u}, s_1)$ in the definition, and we always set $\Sigma(s_1)$ equal to N for each next state. For the condition (3), there are two classes of states s_1 for which there is no $t \geq 0$ such that $\Sigma(s_1, t) \neq \phi$; one is those which are not reachable in M_1' and the other is those which are reachable in M_1' but not with the existence of M_2. For a state s_1 in the first class, $\Sigma(s_1)$ remains *undefined* until it is explicitly set to $\{\phi\}$ at the end of the procedure, and thus the condition holds. For a state s_1 of the second class, the condition holds if the conditions (1) and (2) hold, since in this case, the procedure sets N to $\{\phi\}$.

Hence, the proof is done if we prove condition (1), i.e. for each $s_1 \in S_1'$ and for all $t \geq 0$, if $\Sigma(s_1, t) \neq \phi$, then $\Sigma(s_1, t) = \Sigma(s_1)$. We claim it by induction on $t \geq 0$. Consider the case where $t = 0$. The only state s_1 with $\Sigma(s_1, 0) \neq \phi$ is the reset state r_1. The procedure sets $\Sigma(r_1) = \{(r_2, \{r\})\}$, which is equal to $\Sigma(r_1, 0)$.

In the induction step, let s_1 be a state such that $\Sigma(s_1, t) \neq \phi$, where $t > 0$. Consider an arbitrary $\mathbf{u} \in B^{|U|}$ and $\tilde{s}_1 \in S_1'$ such that $s_1 = \delta_1'(\tilde{s}_1, \mathbf{u})$ holds in M_1'. We claim that if $\Sigma(\tilde{s}_1, t-1)$ is not empty, then $N'(\tilde{s}_1, \Sigma(\tilde{s}_1, t-1), \mathbf{u}, s_1) = \Sigma(s_1)$. Note that the non-emptiness of $\Sigma(s_1, t)$ implies that there exists at least one such $\tilde{s}_1$. It follows that $\Sigma(s_1, t) = \Sigma(s_1)$ since $\Sigma(s_1, t)$ is given by the union of $N'(\tilde{s}_1, \Sigma(\tilde{s}_1, t-1), \mathbf{u}, s_1)$ over all $\mathbf{u} \in B^{|U|}$ and all $\tilde{s}_1 \in S_1'$ with $s_1 = \delta_1'(\tilde{s}_1, \mathbf{u})$ and since if $\Sigma(\tilde{s}_1, t-1)$ is empty, then $N'(\tilde{s}_1, \Sigma(\tilde{s}_1, t-1), \mathbf{u}, s_1)$ is also empty. By the induction hypothesis, $\Sigma(\tilde{s}_1, t-1) = \Sigma(\tilde{s}_1)$, and thus $N'(\tilde{s}_1, \Sigma(\tilde{s}_1, t-1), \mathbf{u}, s_1)$ is equal to N defined in the procedure for $\tilde{s}_1$ and $\mathbf{u}$. Since at the end of the procedure, the existence of the transition $s_1 = \delta_1'(\tilde{s}_1, \mathbf{u})$ implies

that $\Sigma(s_1) = N, N'(\tilde{s}_1, \Sigma(\tilde{s}_1, t-1), \mathbf{u}, s_1) = \Sigma(s_1)$. This completes the proof for the condition (1). Hence M_1' is a prime machine. ■

The procedure shown in Figure 4.5 is only presented for proving the theorem. It is not used for constructing the E-machine.

The theorem claims that the set of prime machines provides the complete set of implementable behaviors. Hence only prime machines need to be considered in order to be able to represent all permissible behaviors. Next we present another property that holds for permissible prime machines, which is used in constructing the E-machine.

Theorem 4.4.2 *Suppose a machine M_1, prime with respect to M_2 and M, is also permissible. Consider a state $s_1 \in S_1$ such that $\Sigma(s_1, t) \neq \phi$ for some $t \geq 0$. Then the following property holds.*

$$\forall (s_2, s^*) \in \Sigma(s_1), \forall \mathbf{x} \in B^{|X|} : \lambda_2^{(z)}(s_2, \mathbf{xv}) \in \Lambda(s^*, \mathbf{x}),$$

where $\mathbf{v} \in B^{|V|}$ is the output minterm of M_1 uniquely defined for the input $\mathbf{x}$ at the state (s_1, s_2) of $M_1 \times M_2$.

Proof: Suppose for contrary that $\lambda_2^{(z)}(s_2, \mathbf{xv}) \notin \Lambda(s^*, \mathbf{x})$. Since $\Sigma(s_1, t) \neq \phi$ for some $t \geq 0$ and since $(s_2, s^*) \in \Sigma(s_1)$, there exists a sequence σ on X which leads $M_1 \times M_2$ to (s_1, s_2) while M is led to the states of s^* with the same output sequence given by $M_1 \times M_2$ for σ. Since M_1 is permissible, s^* is not an empty set. Then at the state (s_1, s_2), the output of $M_1 \times M_2$ with the input $\mathbf{x}$ is different from any output that can be obtained by M at a state of s^* with the same input $\mathbf{x}$. It follows that the behavior of $M_1 \times M_2$ is not contained in M, which conflicts with the fact that M_1 is permissible. This completes the proof. ■

Example 4.4.1 *Consider M_2 and M shown in Figure 4.6, where each of X, V, U, and Z consists of a single variable, while a node and an edge represents a state and a transition, respectively. The label associated with an edge shows the minterms of the inputs and the outputs for the transition corresponding to the edge. The label associated with a node is the name of the corresponding state. The reset states of M_2 and M are state 1 and state A, respectively.*

Three permissible machines for these M_2 and M are shown in Figure 4.7-(a), (b), and (c). For each machine, the one shown on the right-hand side is an equivalent prime machine, where the label associated with each state s_1 is $\Sigma(s_1)$.

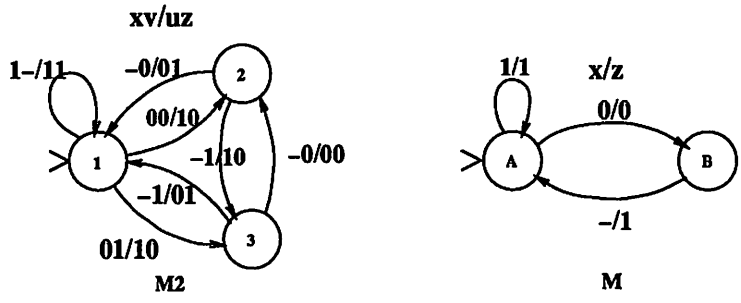

Figure 4.6: Example of M_2 and M

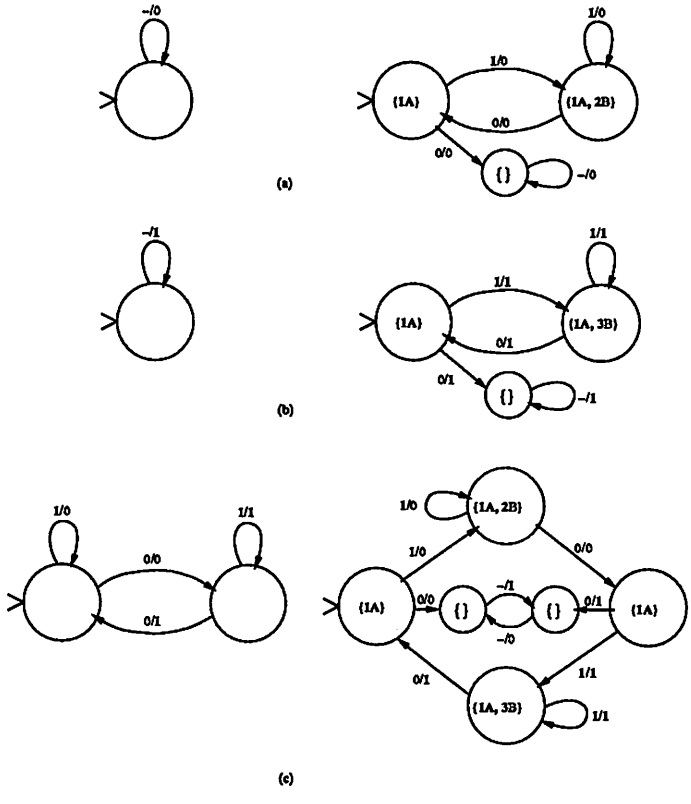

Figure 4.7: Permissible Machines M_1 (u/v)

4.5 The E-machine and its Properties

In this section, we show that the complete set of permissible behaviors for M_1 can be represented by a single non-deterministic machine, called the E-machine. The E-machine is computed by a fixed point iteration. We first present the definition of the E-machine, and then claim associated properties.

4.5.1 The E-machine

Consider the transition relation of a non-deterministic machine given by the following computation. Let $\mathcal{S}^{(0)} = \{(r_2, \{r\})\}$ and compute $T^{(t+1)}$ and $\mathcal{S}^{(t+1)}$ for a given $\mathcal{S}^{(t)} \subseteq S_2 \times 2^S$. Let Σ_p and Σ_n be subsets of $S_2 \times 2^S$, respectively, and $\mathbf{u}$ and $\mathbf{v}$ minterms of $B^{|U|}$ and $B^{|V|}$. $T^{(t+1)}(\Sigma_p, \mathbf{u}, \mathbf{v}, \Sigma_n) = 1$ if and only if the following three conditions are satisfied:

(1) $\Sigma_p \in \mathcal{S}^{(t)}$

(2) $\forall(\mathbf{x}, \tilde{s}_2, \tilde{s^*}) \in B^{|X|} \times S_2 \times 2^S$: $(\tilde{s}_2, \tilde{s^*}) \in \Sigma_p$ and $\mathbf{u} = \lambda_2^{(u)}(\tilde{s}_2, \mathbf{xv})$
$\Rightarrow$ (a) $\lambda_2^{(z)}(\tilde{s}_2, \mathbf{xv}) \in \Lambda(\tilde{s^*}, \mathbf{x})$
(b) $(\delta_2(\tilde{s}_2, \mathbf{xv}), \Delta(\tilde{s^*}, \mathbf{x}, \lambda_2^{(z)}(\tilde{s}_2, \mathbf{xv}))) \in \Sigma_n$

(3) $\forall(s_2, s^*) \in S_2 \times 2^S$: $(s_2, s^*) \in \Sigma_n$
$\Rightarrow$ $\exists(\mathbf{x}, \tilde{s}_2, \tilde{s^*}) \in B^{|X|} \times S_2 \times 2^S$:
(a) $(\tilde{s}_2, \tilde{s^*}) \in \Sigma_p$
(b) $\mathbf{u} = \lambda_2^{(u)}(\tilde{s}_2, \mathbf{xv})$
(c) $s_2 = \delta_2(\tilde{s}_2, \mathbf{xv})$
(d) $s^* = \Delta(\tilde{s^*}, \mathbf{x}, \lambda_2^{(z)}(\tilde{s}_2, \mathbf{xv}))$.

In each computation, $\mathcal{S}^{(t)}$ is a set of subsets of $S_2 \times 2^S$. Note that the empty set $\{\phi\}$ may be in $\mathcal{S}^{(t)}$. Given $T^{(t+1)}$, we compute $\mathcal{S}^{(t+1)}$ as follows. $\mathcal{S}^{(t+1)}(\Sigma_p) = 1$ if and only if $\mathcal{S}^{(t)}(\Sigma_p) = 1$ or there exist $\tilde{\Sigma}_p \in \mathcal{S}^{(t)}$, $\mathbf{u} \in B^{|U|}$, and $\mathbf{v} \in B^{|V|}$ such that $T^{(t+1)}(\tilde{\Sigma}_p, \mathbf{u}, \mathbf{v}, \Sigma_p) = 1$. Intuitively, we are computing a transition relation that on each step is being extended to a new set of states, where each state corresponds to a subset of $S_2 \times 2^S$. These states are added to the transition relation. This is continued until nothing new is seen, i.e. a fixed point is reached.

Let K be the smallest positive integer such that $\mathcal{S}^{(K)}(\Sigma_p) = \mathcal{S}^{(K-1)}(\Sigma_p)$. Such K always exists since the number of the elements of the set $\mathcal{S}^{(t)}$ is not decreasing during the computation and the number of subsets of $S_2 \times 2^S$ is finite. Let $\mathcal{S} = \mathcal{S}^{(K)} \cup \{\phi\}$.

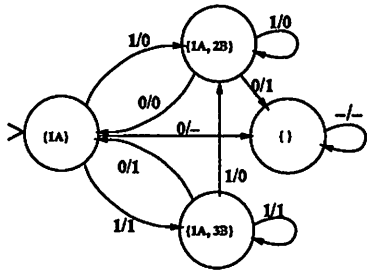

Figure 4.8: The E-machine for Example 4.4.1

Let $T : S \times B^{|U|} \times B^{|V|} \times S \rightarrow B$ be a relation such that $T(\Sigma_p, \mathbf{u}, \mathbf{v}, \Sigma_n) = 1$ if and only if

(1) $\Sigma_p = \Sigma_n = \{\phi\}$ or

(2) $T^{(K)}(\Sigma_p, \mathbf{u}, \mathbf{v}, \Sigma_n) = 1$.

Finally the E-machine is defined as a 5-tuple $T = (U, V, S, T, \Sigma_r)$, where $\Sigma_r = \{(r_2; \{r\})\}$. Recall that each state of the E-machine represents a subset of $S_2 \times 2^S$, but note that if M is a completely specified deterministic machine, a state of the E-machine is a set of pairs of states of M_2 and M, i.e. a subset of $S_2 \times S$, rather than $S_2 \times 2^S$.

Example 4.5.1 *For M_2 and M in Example 4.4.1, the transition relation of the E-machine is shown in Figure 4.8.*

4.5.2 Properties of the E-machine

The objective in this section is to show that the E-machine captures the complete set of permissible behaviors. More specifically, a behavior implementable at M_1 is permissible if and only if the behavior is contained in the E-machine.

Theorem 4.5.1 *A behavior implementable at M_1 contained in the E-machine is permissible.*

Proof: Consider an arbitrary sequence $\sigma = (\mathbf{x}_0, \ldots, \mathbf{x}_k)$ of $B^{|X|}$. Let $\sigma_x^{(t)} = (\mathbf{x}_0, \ldots, \mathbf{x}_t)$ be the prefix subsequence of σ with the length $t+1$, where $0 \leq t \leq k$ and define σ_{-1} as the null sequence. Since the behavior is implementable, $\sigma_x^{(t)}$ uniquely defines the pair of sequences of $B^{|U|}$ and $B^{|V|}$,

say $(\sigma_u^{(t)}, \sigma_v^{(t)})$, as well as the sequence $\sigma_z^{(t)}$ of $B^{|Z|}$, where $\sigma_u^{(t)} = (\mathbf{u}_0, \ldots, \mathbf{u}_t)$, $\sigma_v^{(t)} = (\mathbf{v}_0, \ldots, \mathbf{v}_t)$, and $\sigma_z^{(t)} = (\mathbf{z}_0, \ldots, \mathbf{z}_t)$, such that $(\sigma_u^{(t)}, \sigma_v^{(t)})$ is realized by the behavior and for each i, $0 \leq i \leq t$, $\mathbf{u}_i\mathbf{z}_i = \lambda_2(s_2^{(i)}, \mathbf{x}_i\mathbf{v}_i)$ and $s_2^{(i+1)} = \delta_2(s_2^{(i)}, \mathbf{x}_i\mathbf{v}_i)$, where $s_2^{(0)} = r_2$. Let us define $(\sigma_u^{(-1)}, \sigma_v^{(-1)})$ as the pair of null sequences. Note that M_2 is led to $s_2^{(t+1)}$ by applying $\sigma_x^{(t)}$. Also, for $t \geq 0$, let $s^* \subseteq S$ be the set of states to which M is led by $\sigma_x^{(t)}$ with the output sequence $\sigma_z^{(t)}$. Similarly, let $\tilde{s^*} \subseteq S$ be the set of states to which M is led by $\sigma_x^{(t-1)}$ with the output sequence $\sigma_z^{(t-1)}$. In case $t = 0$, we define $\tilde{s^*} = \{r\}$.

We show by induction on $t \geq 0$ that by applying the subsequence $\sigma_x^{(t)}$,

1. $\lambda_2^{(z)}(s_2^{(t)}, \mathbf{x}_t\mathbf{v}_t) \in \Lambda(\tilde{s^*}, \mathbf{x}_t)$,
2. there exists a unique state $\Sigma^{(t+1)} \in \mathcal{S}$ to which the E-machine is led by $(\sigma_u^{(t)}, \sigma_v^{(t)})$,
3. s^* is not an empty set, and
4. $(s_2^{(t+1)}, s^*) \in \Sigma^{(t+1)}$.

Denote $\Sigma^{(0)} = \Sigma_r$. Then $\Sigma^{(0)}$ is the unique state of the E-machine to which the E-machine can be led by $(\sigma_u^{(-1)}, \sigma_v^{(-1)})$, where we see $(r_2, \{r\}) \in \Sigma^{(0)}$. By the induction hypothesis, there exists a unique $\Sigma^{(t)}$ to which the E-machine can be led by $(\sigma_u^{(t-1)}, \sigma_v^{(t-1)})$. Furthermore, $\tilde{s^*}$ is not empty and $(s_2^{(t)}, \tilde{s^*}) \in \Sigma^{(t)}$. It follows that $\Sigma^{(t)} \neq \{\phi\}$.

Since $(s_2^{(t)}, \tilde{s^*}) \in \Sigma^{(t)}$ and $\mathbf{u}_t = \lambda_2^{(u)}(s_2^{(t)}, \mathbf{x}_t\mathbf{v}_t)$, then $T^{(K)}(\Sigma^{(t)}, \mathbf{u}, \mathbf{v}, \{\phi\}) = 0$. However, since the behavior is contained in T and since $\Sigma^{(t)}$ is the unique state to which the E-machine can be led by $(\sigma_u^{(t-1)}, \sigma_v^{(t-1)})$, there must exist a state $\Sigma^{(t+1)} \in \mathcal{S}$ such that $T(\Sigma^{(t)}, \mathbf{u}, \mathbf{v}, \Sigma^{(t+1)}) = 1$. It follows that $T^{(K)}(\Sigma^{(t)}, \mathbf{u}, \mathbf{v}, \Sigma^{(t+1)}) = 1$. Therefore, by construction, $\lambda_2^{(z)}(s_2^{(t)}, \mathbf{x}_t\mathbf{v}_t) \in \Lambda(\tilde{s^*}, \mathbf{x}_t)$. Since $s^* = \Delta(\tilde{s^*}, \mathbf{x}_t, \mathbf{z}_t)$ and $\mathbf{z}_t = \lambda_2^{(z)}(s_2^{(t)}, \mathbf{x}_t\mathbf{v}_t)$, s^* is not empty. Also, since $s_2^{(t+1)} = \delta_2(s_2^{(t)}, \mathbf{x}_t\mathbf{v}_t)$, $(s_2^{(t+1)}, s^*) \in \Sigma^{(t+1)}$ by construction. It remains to show that such $\Sigma^{(t+1)}$ is unique. Since we know the uniqueness of $\Sigma^{(t)}$, the proof is done if we show that for any $\Sigma \in \mathcal{S}$ such that $T(\Sigma^{(t)}, \mathbf{u}, \mathbf{v}, \Sigma) = 1$, $\Sigma = \Sigma^{(t+1)}$. Consider an arbitrary such Σ. By the argument above, $\Sigma \neq \{\phi\}$, and thus $T^{(K)}(\Sigma^{(t)}, \mathbf{u}, \mathbf{v}, \Sigma) = 1$. Then by the condition (3) of the construction of $T^{(K)}$ shown in Section 4.5.1, for an arbitrary pair $(s_2, s^*) \in \Sigma$, there exist $\mathbf{x} \in B^{|X|}$ and $(\tilde{s}_2, \tilde{s^*}) \in \Sigma^{(t)}$ such that $\mathbf{u} = \lambda_2^{(u)}(\tilde{s}_2, \mathbf{x}\mathbf{v})$, $s_2 = \delta_2(\tilde{s}_2, \mathbf{x}\mathbf{v})$, and $s^* = \Delta(\tilde{s^*}, \mathbf{x}, \lambda_2^{(z)}(\tilde{s}_2, \mathbf{x}\mathbf{v}))$. Then since $T^{(K)}(\Sigma^{(t)}, \mathbf{u}, \mathbf{v}, \Sigma^{(t+1)}) = 1$, by the condition (2) of the construction of $T^{(K)}$, (s_2, s^*) must be a member of $\Sigma^{(t+1)}$. Thus $\Sigma \subseteq \Sigma^{(t+1)}$. The same argument holds to claim that $\Sigma^{(t+1)} \subseteq \Sigma$, and thus $\Sigma^{(t+1)}$ with the property above is unique. This completes the proof for the induction step.

Hence, the sequence of the global output Z, realized for the input sequence σ by the behavior M_1 together with M_2, is also realized by M. Since σ is arbitrary, the behavior is permissible. ■

We now claim that all permissible behaviors can be captured by the E-machine. A machine *contained* in the E-machine is defined as follows.

Definition: Contained Machine

Given a finite state machine $T = (U, V, S, T, \Sigma_r)$, a completely specified deterministic finite state machine $M_1 = (U, V, S_1, \lambda_1, \delta_1, r_1)$ is **contained** in T if there exists a mapping $\varphi : S_1 \rightarrow S$ such that $\varphi(r_1) = \Sigma_r$ and for all $s_1 \in S_1$ and $\mathbf{u} \in B^{|U|}$, $T(\varphi(s_1), \mathbf{u}, \lambda_1(s_1, \mathbf{u}), \varphi(\delta_1(s_1, \mathbf{u}))) = 1$.

Lemma 4.5.1 *Consider a machine $M_1 = (U, V, S_1, \lambda_1, \delta_1, r_1)$ contained in a finite state machine $T = (U, V, S, T, \Sigma_r)$. Then the behavior of M_1 is contained in T.*

Proof: We show by induction on $t \geq 0$ that for an arbitrary input sequence on U with the length t, the output sequence of M_1 given by the input sequence can be realized by T. The claim is trivially true when $t = 0$. Consider the case where $t > 0$. Let σ_u be an arbitrary sequence on U with the length $t - 1$ and let $\mathbf{u} \in B^{|U|}$ be an arbitrary minterm. Let $\tilde{s}_1$ be the state of M_1 to which σ_u leads M_1. Let σ_v be the sequence of V given by M_1 for the input sequence σ_u. By the induction hypothesis, (σ_u, σ_v) is realized by T. Since M_1 is contained in T, $T(\varphi(\tilde{s}_1), \mathbf{u}, \lambda_1(\tilde{s}_1, \mathbf{u}), \varphi(\delta_1(\tilde{s}_1, \mathbf{u}))) = 1$. Thus the pair of sequences $(\sigma_u \mathbf{u}, \sigma_v \lambda(\tilde{s}_1, \mathbf{u}))$ is realized by T, which completes the proof for the induction step. Hence the behavior of M_1 is contained in T. ■

With this lemma, all we need to show is that for an arbitrary permissible machine M_1, there exists an equivalent machine contained in the E-machine.

Theorem 4.5.2 *For each permissible machine M_1, there exists an equivalent machine contained in the E-machine.*

Proof: By Theorem 4.4.1, there exists a prime machine $M_1' = (U, V, S_1, \lambda_1, \delta_1, r_1)$ which is equivalent to M_1. Let $\varphi : S_1 \rightarrow S$ be a mapping such that for each state $s_1 \in S_1$, $\varphi(s_1) = \Sigma(s_1)$. Note that $\varphi(r_1) = \Sigma_r$. We claim that M_1' is contained in the E-machine under φ.

Note first that since M_1' is prime, for a state $\tilde{s}_1$ for which there is no $t \geq 0$ such that $\Sigma(\tilde{s}_1, t) \neq \phi$, $\Sigma(\tilde{s}_1) = \{\phi\}$. Then for an arbitrary $\mathbf{u} \in B^{|U|}$, and for the next state

$s_1 = \delta_1(\tilde{s}_1, \mathbf{u})$, $N(\tilde{s}_1, \Sigma(\tilde{s}_1), \mathbf{u}, s_1) = \Sigma(s_1) = \{\phi\}$. Thus by the construction of the E-machine, $T(\varphi(\tilde{s}_1), \mathbf{u}, \lambda_1(\tilde{s}_1, \mathbf{u}), \varphi(s_1)) = 1$ for such $\tilde{s}_1$.

We now show by induction on $t \geq 0$ that for all states $\tilde{s}_1 \in S_1$ such that $\Sigma(\tilde{s}_1, t) \neq \phi$ and for all $\mathbf{u} \in B^{|U|}$, $T^{(t+1)}(\varphi(\tilde{s}_1), \mathbf{u}, \mathbf{v}, \varphi(s_1)) = 1$, where $\mathbf{v} = \lambda_1(\tilde{s}_1, \mathbf{u})$ and $s_1 = \delta_1(\tilde{s}_1, \mathbf{u})$. We also prove that if $\Sigma(s_1, t+1) \neq \phi$, then $\varphi(s_1) \in \mathcal{S}^{(t+1)}$, where $\mathcal{S}^{(t+1)}$ is defined in Section 4.5.1. We show that each condition for constructing the E-machine, given in Section 4.5.1, is satisfied, where $\Sigma_p = \varphi(\tilde{s}_1) = \Sigma(\tilde{s}_1)$.

First, since M_1' is prime, for its reset state r_1, $\varphi(r_1)$ must be equal to $\{(r_2, \{r\})\}$, and thus $\varphi(r_1) \in \mathcal{S}^{(0)}$. In the induction step for a general t, the induction hypothesis implies that $\varphi(\tilde{s}_1) \in \mathcal{S}^{(t)}$.

Consider an arbitrary $\mathbf{x} \in B^{|X|}$ and $(\tilde{s}_2, \tilde{s}^*) \in \Sigma(\tilde{s}_1)$ such that $\mathbf{u} = \lambda_2^{(u)}(\tilde{s}_2, \mathbf{xv})$. If there is no such $\mathbf{x}$ and $(\tilde{s}_2, \tilde{s}^*)$, then the conditions (2) and (3) are trivially true with $\Sigma_n = \{\phi\}$. Therefore, $T^{(t+1)}(\varphi(\tilde{s}_1), \mathbf{u}, \mathbf{v}, \{\phi\}) = 1$. Since in this case $N(\tilde{s}_1, \Sigma(\tilde{s}_1), \mathbf{u}, s_1)$ is empty, the primeness of M_1' implies that $\Sigma(s_1) = \{\phi\}$. Therefore $T^{(t+1)}(\varphi(\tilde{s}_1), \mathbf{u}, \mathbf{v}, \varphi(s_1)) = 1$, and the proof for this case is done.

Suppose such $\mathbf{x}$ and $(\tilde{s}_2, \tilde{s}^*)$ exist. Note that in this case, $\Sigma(s_1, t+1) \neq \phi$. We first consider condition (2).

Since M_1' is permissible, Theorem 4.4.2 implies that $\lambda_2^{(z)}(\tilde{s}_2, \mathbf{xv}) \in \Lambda(\tilde{s}^*, \mathbf{x})$. Also, since M_1' is prime, $\Sigma(s_1)$ is equal to $N(\tilde{s}_1, \Sigma(\tilde{s}_1), \mathbf{u}, s_1)$, which is denoted by N hereafter. The definition of N implies that $(\delta_2(\tilde{s}_2, \mathbf{xv}), \Delta(\tilde{s}^*, \mathbf{x}, \lambda_2^{(z)}(\tilde{s}_2, \mathbf{xv})))$ is a member of N. Therefore, it is also a member of $\Sigma(s_1)$. Since $\Sigma(s_1) = \varphi(s_1)$, condition (2) holds for $\Sigma_n = \varphi(s_1)$.

By the equality between $\Sigma(s_1)$ and N given above, for all $(s_2, s^*) \in \Sigma(s_1)$, $(s_2, s^*) \in N$. It follows that condition (3) is satisfied for $\Sigma_n = \varphi(s_1)$.

Therefore, $T^{(t+1)}(\varphi(\tilde{s}_1), \mathbf{u}, \mathbf{v}, \varphi(s_1)) = 1$. It follows that $\Sigma(s_1) \in \mathcal{S}^{(t+1)}$, and thus the claim above holds. Note that by this induction, we see that for each $s_1 \in S_1$, $\varphi(s_1) \in \mathcal{S}$. Hence, by the construction of T, $T(\varphi(\tilde{s}_1), \mathbf{u}, \mathbf{v}, \varphi(s_1)) = 1$ for all $\tilde{s}_1 \in S_1$ and for all $\mathbf{u} \in B^{|U|}$. ■

We have now reached the key statement of the E-machine.

Theorem 4.5.3 *Suppose a behavior at M_1 is implementable. Then the behavior is permissible if and only if it is contained in the E-machine.*

It follows that the set of implementable behaviors given in the E-machine precisely provides the complete set of permissible behaviors at M_1. By Theorem 4.5.2, we see that the E-machine contains the permissible behaviors by using their associated contained prime machines.

In this sense, we can say that the E-machine tells not only which behavior is permissible, but also how the behavior can be realized by at least one finite state machine. In other words, the following claim holds for permissible machines.

Corollary 4.5.1 *Suppose a finite state machine is implementable at M_1. Then it is permissible if and only if there exists an equivalent machine contained in the E-machine.*

Proof: If M_1 is a permissible machine, then by Theorem 4.5.2, there exists an equivalent machine contained in the E-machine. Conversely, suppose that for an implementable machine M_1, there exists an equivalent machine contained in the E-machine. Then by Lemma 4.5.1, the behavior of M_1 is contained in the E-machine. Thus, by Theorem 4.5.1, M_1 is permissible. ∎

4.5.3 A Variation of the E-machine

By definition, each state of the E-machine corresponds to a set of pairs $(s_2, s^*) \in S_2 \times 2^S$. In case M is a completely specified deterministic machine, this corresponds to a set of pairs $(s_2, s) \in S_2 \times S$. The reason why a *set* of pairs is associated with each state is that the global inputs X do not directly drive M_1. Specifically, consider the fixed point iteration introduced in Section 4.5.1 for constructing the E-machine. In condition (2), for each pair $(\tilde{s}_2, \tilde{s^*}) \in \Sigma_p$, there might exist more than one $\mathbf{x} \in B^{|X|}$ such that $\mathbf{u} = \lambda_2^{(u)}(\tilde{s}_2, \mathbf{xv})$. Then for each such $\mathbf{x} \in B^{|X|}$, we need to include the pair of next states $(\delta_2(\tilde{s}_2, \mathbf{xv}), \Delta(\tilde{s^*}, \mathbf{x}, \lambda_2^{(z)}(\tilde{s}_2, \mathbf{xv})))$ in Σ_n. Since the pair of next states may be different for different $\mathbf{x}$, we include in the next state Σ_n of the E-machine more than one pair $(\delta_2(\tilde{s}_2, \mathbf{xv}), \Delta(\tilde{s^*}, \mathbf{x}, \lambda_2^{(z)}(\tilde{s}_2, \mathbf{xv})))$ for each $(\tilde{s}_2, \tilde{s^*})$ and $(\mathbf{u}, \mathbf{v})$.

However, as shown below, if the global inputs X directly drives M_1, then each state of the E-machine corresponds to a single pair $(s_2, s^*) \in S_2 \times 2^S$. In this case, the only difference from our original problem is that M_1 takes as input U and X, as shown in Figure 4.9-(a). Namely, M_1 is given by $(X \cup U, V, S_1, \lambda_1, \delta_1, r_1)$. Note first that this can be reduced to the original problem by modifying M_2 so that the global inputs drive M_1 through M_2, as shown in Figure 4.9-(b). Alternatively, we can directly apply the fixed point computation given in Section 4.5.1 to construct the E-machine for the case of Figure 4.9-(a). The difference is that the E-machine is now defined with the input $X \cup U$, and thus the transition relation $T^{(t+1)}$ has a minterm $\mathbf{x} \in B^{|X|}$ as input.

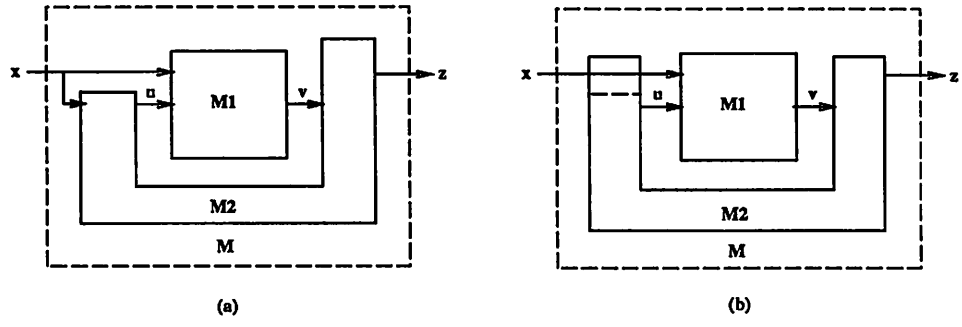

Figure 4.9: The Problem where Global Inputs Drive M_1

Specifically, $T^{(t+1)}(\Sigma_p, \mathbf{xu}, \mathbf{v}, \Sigma_n) = 1$ if and only if the following three conditions are satisfied:

(1) $\Sigma_p \in \mathcal{S}^{(t)}$

(2) $\forall(\tilde{s}_2, \tilde{s^*}) \in S_2 \times 2^S$: $(\tilde{s}_2, \tilde{s^*}) \in \Sigma_p$ and $\mathbf{u} = \lambda_2^{(u)}(\tilde{s}_2, \mathbf{xv})$

$\Rightarrow$ (a) $\lambda_2^{(z)}(\tilde{s}_2, \mathbf{xv}) \in \Lambda(\tilde{s^*}, \mathbf{x})$

(b) $(\delta_2(\tilde{s}_2, \mathbf{xv}), \Delta(\tilde{s^*}, \mathbf{x}, \lambda_2^{(z)}(\tilde{s}_2, \mathbf{xv}))) \in \Sigma_n$

(3) $\forall(s_2, s^*) \in S_2 \times 2^S$: $(s_2, s^*) \in \Sigma_n$

$\Rightarrow$ $\exists(\tilde{s}_2, \tilde{s^*}) \in \times S_2 \times 2^S$:

(a) $(\tilde{s}_2, \tilde{s^*}) \in \Sigma_p$

(b) $\mathbf{u} = \lambda_2^{(u)}(\tilde{s}_2, \mathbf{xv})$

(c) $s_2 = \delta_2(\tilde{s}_2, \mathbf{xv})$

(d) $s^* = \Delta(\tilde{s^*}, \mathbf{x}, \lambda_2^{(z)}(\tilde{s}_2, \mathbf{xv}))$.

As mentioned above, if Σ_p is a singleton, i.e. it consists of a single pair $(\tilde{s}_2, \tilde{s^*})$, then Σ_n defined by these conditions is also a singleton. Since initially $\mathcal{S}^{(0)}$ is defined as $\{(r_2, \{r\})\}$, each state of the E-machine corresponds to a single pair of elements of $S_2 \times 2^S$, rather than a set of pairs. The correctness of the E-machine, i.e. Theorem 4.5.3, can be proved in the exactly same way as the original case.

Hence for the case where the global inputs also drive M_1, the number of states of the E-machine can be significantly smaller than for the original case. Note that the problem of finding permissible behaviors for the outside component M_2, discussed in Section 4.1.2, falls into this category.

4.5.4 The E-machine in Hierarchical Optimization

As seen so far, the E-machine is in general a non-deterministic finite state machine, i.e. for a given state and input, the next state and the corresponding output may not be unique. The non-determinism makes it possible to represent a set of behaviors in a single machine. Since the specification M may also be non-deterministic, we can intuitively say that the flexibility originally given for the entire system is mapped into a particular component M_1, and the resulting flexibility is also represented by a non-deterministic machine, i.e. the E-machine.

This observation leads to the use of E-machines in the following optimization scenario in a framework of hierarchical system designs. Consider a system of interacting components, where each component corresponds to a completely specified deterministic finite state machine. Suppose that we have computed the E-machine for a particular component of the system. Suppose also that the system is described in a hierarchical way, so that each component can also be regarded as a system of interacting finite state machines at one level below. This situation is illustrated in Figure 4.10, where Figure 4.10-(b) is one level below that of Figure 4.10-(a), and the entire system in (b) corresponds to the shaded component given at the original level (a). Then at level (b), the specification of the system is given by the set of permissible behaviors identified at the corresponding component at level (a), which is given by the E-machine computed for the component at the original level. Then using the E-machine as the specification, we choose a sub-component at level (b), and compute the E-machine for that component. This procedure can be repeated to compute E-machines for levels further below. Hence, using E-machines, the flexibility given at a higher level can be transformed into a lower level in a uniform fashion.

4.6 The Structure of the E-machine and a Non-Deterministic Construction

We have stated that the non-determinism of the E-machine is the means by which a set of behaviors can be represented by a single machine. However, we note that the E-machine is a special type of non-deterministic machine. Namely, for a given state $\Sigma_p \in \mathcal{S}$ and pair of input and output minterms $(\mathbf{u}, \mathbf{v}) \in B^{|U|} \times B^{|V|}$, if there exists a next state Σ_n such that $T(\Sigma_p, \mathbf{u}, \mathbf{v}, \Sigma_n) = 1$, then such Σ_n is unique. In other words, if we introduce a set of new symbols which uniquely represent and replace each pair of input and output minterms of the E-machine, then the result is a deterministic finite automaton. This is true since in the construction of the E-machine, we uniquely

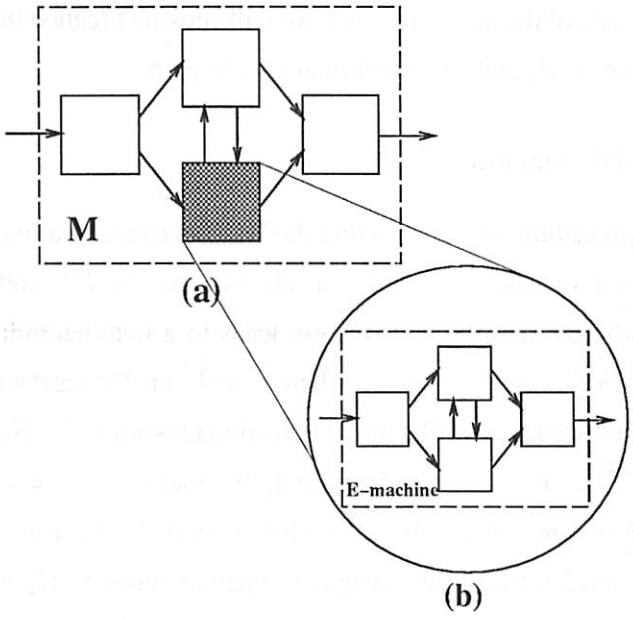

Figure 4.10: Hierarchical Optimization of Interacting Finite State Machines

define the next state, if it exists, for a given pair of input and output minterms. We call a finite state machine with this property a *pseudo non-deterministic finite state machine.*

It is informative to ask whether it is possible to construct the E-machine, so that the automaton corresponding to the machine is non-deterministic and accepts the same language as the original. In other words, if we perform the subset construction to determinize the non-deterministic automaton, where we assume each pair of input and output minterms is a single symbol, then can we obtain the deterministic automaton corresponding to the E-machine? It is interesting to construct and represent the E-machine this way, since it is known that the subset construction, or determinization, could introduce an exponentially large number of states in general. Thus we expect that the non-deterministic automaton has a smaller state space; the complete set of permissible behaviors is then represented in the more compact way.

In this section, we consider the structure given in Figure 4.1 but assume that M is a completely specified deterministic machine. We then present a procedure, suggested by Alex Saldanha, that generates a machine such that by performing an operation similar to the subset construction we obtain the E-machine as originally defined. Recall that in case M is a deterministic

machine, a state of the E-machine defined in Section 4.5.1 corresponds to a set of pairs of states of M_2 and M. A state of the machine which we will construct (called the NDE-machine) corresponds to a pair of states of M_2 and M, rather than a set of pairs.

4.6.1 The NDE-machine

The procedure for constructing the NDE-machine is a fixed point iteration. Denote the transition relation and the set of states at the t-th step by $T_N^{(t)}$ and $\mathcal{S}_N^{(t)}$ respectively, where the subscript N indicates that the construction leads to a non-deterministic E-machine in the sense above. Initially, $\mathcal{S}_N^{(0)} = \{(r_2, r), \phi, \kappa\}$, where r_2 and r are the reset states of M_2 and M while ϕ and κ are newly introduced states. The initial transition relation $T_N^{(0)} : \mathcal{S}_N^{(0)} \times B^{|U|} \times B^{|V|} \times \mathcal{S}_N^{(0)} \to B$ is defined as $T_N^{(0)}(\varsigma_p, \mathbf{u}, \mathbf{v}, \varsigma_n) = 1$ if and only if either $\varsigma_p = \varsigma_n = \phi$ or $\varsigma_p = \varsigma_n = \kappa$, i.e. we start with only self-loops on ϕ and κ. In general for the step $(t+1)$, $T_N^{(t+1)}(\varsigma_p, \mathbf{u}, \mathbf{v}, \varsigma_n)$ is defined when $\varsigma_p \in \mathcal{S}_N^{(t)} - \{\phi, \kappa\}$, i.e. the present state ς_p is a pair of states of M_2 and M, say $\varsigma_p = (\tilde{s}_2, \tilde{s})$, which has been introduced as a state in the transition relation $T_N^{(t)}$. Then $T_N^{(t+1)}(\varsigma_p, \mathbf{u}, \mathbf{v}, \varsigma_n) = 1$ if and only if one of the following three conditions hold:

$$
\begin{aligned}
&\text{(a)} \quad \forall \mathbf{x} \in B^{|X|} : \mathbf{u} \neq \lambda_2^{(u)}(\tilde{s}_2, \mathbf{xv}) \text{ and } \varsigma_n = \phi, \text{ or} \\
&\text{(b)} \quad \exists \mathbf{x} \in B^{|X|} : \mathbf{u} = \lambda_2^{(u)}(\tilde{s}_2, \mathbf{xv}) \text{ and } \lambda_2^{(z)}(\tilde{s}_2, \mathbf{xv}) \neq \lambda(\tilde{s}, \mathbf{x}) \text{ and } \varsigma_n = \kappa, \text{ or} \\
&\text{(c)} \quad \forall \mathbf{x} \in B^{|X|} : \mathbf{u} = \lambda_2^{(u)}(\tilde{s}_2, \mathbf{xv}) \Rightarrow \lambda_2^{(z)}(\tilde{s}_2, \mathbf{xv}) = \lambda(\tilde{s}, \mathbf{x}) \text{ and} \\
&\qquad \exists \mathbf{x} \in B^{|X|} : \mathbf{u} = \lambda_2^{(u)}(\tilde{s}_2, \mathbf{xv}) \text{ and } \varsigma_n = (\delta_2(\tilde{s}_2, \mathbf{xv}), \delta(\tilde{s}, \mathbf{x})),
\end{aligned}
\tag{4.1}
$$

where $\lambda : S \times B^{|X|} \to B^{|Z|}$ and $\delta : S \times B^{|X|} \to S$ are the output and the next state functions of M, which are defined since M is deterministic. Condition (a) says that if there is no $\mathbf{x}$ which causes M_2 to output $\mathbf{u}$ at the state $\tilde{s}_2$ for the input $\mathbf{v}$, then we cause a transition to ϕ. Condition (b) means that if there exists an $\mathbf{x}$ which causes M_2 to output $\mathbf{u}$ at $\tilde{s}_2$ for $\mathbf{v}$ but the z output is not allowed, then we cause a transition to κ. Finally (c), if all possible z outputs are allowed and if there is at least one $\mathbf{x}$ that makes M_2 and M transit to ς_n, then this transition is put in $T_N^{(t+1)}$.

Let T_N be the transition relation of the fixed point of the computation. Namely, for positive integer K, if $\mathcal{S}_N^{(K)} = \mathcal{S}_N^{(K-1)}$, then $T_N = T_N^{(K)}$. Similarly, let $\mathcal{S}_N = \mathcal{S}_N^{(K)}$. The NDE-machine is defined as $(U, V, \mathcal{S}_N, T_N, \varsigma_r)$, where the reset state ς_r is given by $\varsigma_r = (r_2, r)$. The transition relation of the NDE-machine for M_2 and M used in Example 4.4.1 is shown in Figure 4.11, where the states ϕ and κ are denoted respectively by $\{\}$ and k. Note that unlike the E-machine, the NDE-machine has a property that for a state ς_p and pair of input and output minterms $(\mathbf{u}, \mathbf{v})$, there might exist

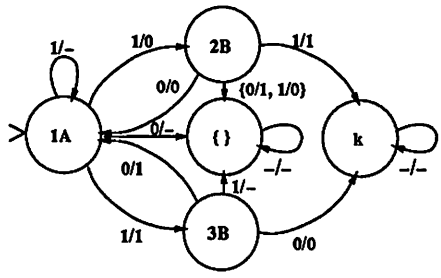

Figure 4.11: The NDE-machine for Example 4.4.1

more than one state ς_n such that $T_N(\varsigma_p, \mathbf{u}, \mathbf{v}, \varsigma_n) = 1$. In Figure 4.11, the NDE-machine can go to either 1A or 2B from state 1A under the input 1 and output 0. It is because for different global input $\mathbf{x} \in B^{|X|}$, M_2 and M may go to different next states with the same $(\mathbf{u}, \mathbf{v})$.

Now, for a given NDE-machine $(U, V, \mathcal{S}_N, T_N, \varsigma_r)$, consider a finite state machine $(U, V, \mathcal{S}_D, T_D, \Sigma_{D_r})$ defined as follows. The state space $\mathcal{S}_D$ is the set of subsets of $\mathcal{S}_N$ that contain ϕ and not contain κ. The reset state Σ_{D_r} is the subset $\{\varsigma_r, \phi\}$. The transition relation $T_D : \mathcal{S}_D \times B^{|U|} \times B^{|V|} \times \mathcal{S}_D \rightarrow B$ is defined as $T_D(\Sigma_{D_p}, \mathbf{u}, \mathbf{v}, \Sigma_{D_n}) = 1$ if and only if

$$\Sigma_{D_n} = \{\varsigma_n \in \mathcal{S}_N \mid \exists \varsigma_p \in \Sigma_{D_p} : T_N(\varsigma_p, \mathbf{u}, \mathbf{v}, \varsigma_n) = 1\} \text{ and } \kappa \notin \Sigma_{D_n} \tag{4.2}$$

This construction is the subset construction, or determinization, of a non-deterministic finite automaton [26], where the state κ is the unique non-accepting state, meaning that a string which *can* lead the automaton to κ is not accepted. Only subsets, generated in the subset construction, which do not contain κ are allowed next-state subsets. In this way, we end up with a finite state machine which contains only permissible behaviors.

Let $\mathcal{S}'_D$ be the union of the state $\{\phi\} \in \mathcal{S}_D$ and the set of states of T_D reachable from the reset state Σ_{D_r}. Let $T'_D : \mathcal{S}'_D \times B^{|U|} \times B^{|V|} \times \mathcal{S}'_D \rightarrow B$ be the transition relation of T_D restricted to the states $\mathcal{S}'_D$. We then claim that the restricted machine $T'_D = (U, V, \mathcal{S}'_D, T'_D, \Sigma_{D_r})$ is the E-machine. More specifically, T'_D and the E-machine are isomorphic, i.e. there exists a one-to-one onto mapping f from the state space of the E-machine to that of T'_D such that $T(\Sigma_p, \mathbf{u}, \mathbf{v}, \Sigma_n) = 1$ if and only if $T'_D(f(\Sigma_p), \mathbf{u}, \mathbf{v}, f(\Sigma_n)) = 1$.

Theorem 4.6.1 *The machine T'_D and the E-machine are isomorphic.*

Proof: Given a subset Σ of pairs of states of M_2 and M, let $f(\Sigma)$ be the subset given by adding the state ϕ to Σ, i.e. $f(\Sigma) = \Sigma \cup \{\phi\}$. By definition, f is a one-to-one mapping and thus the inverse of f is also defined. We claim that T'_D and the E-machine are isomorphic under the mapping f.

Suppose $T(\Sigma_p, \mathbf{u}, \mathbf{v}, \Sigma_n) = 1$ holds in the E-machine. If $\Sigma_p = \{\phi\}$, then by the definition of the E-machine, $\Sigma_n = \{\phi\}$. By construction of the NDE-machine T_N, for the present state $\phi \in \mathcal{S}_N$, ϕ is the unique state that satisfies $T_N(\phi, \mathbf{u}, \mathbf{v}, \phi) = 1$. Therefore, $T_D(\{\phi\}, \mathbf{u}, \mathbf{v}, \{\phi\}) = 1$. Since $f(\{\phi\}) = \{\phi\}$, $T'_D(f(\Sigma_p), \mathbf{u}, \mathbf{v}, f(\Sigma_n)) = 1$, and the claim holds.

Consider the case where $\Sigma_p \neq \{\phi\}$. We show $T'_D(f(\Sigma_p), \mathbf{u}, \mathbf{v}, f(\Sigma_n)) = 1$ under the assumption that $f(\Sigma_p) \in \mathcal{S}'_D$. This assumption does not affect the claim, since $f(\Sigma_r) \in \mathcal{S}'_D$ for the reset state $\Sigma_r = \{(r_2, r)\}$ and $T'_D(f(\Sigma_p), \mathbf{u}, \mathbf{v}, f(\Sigma_n)) = 1$ implies that $f(\Sigma_n) \in \mathcal{S}'_D$. By construction of the E-machine, the next state Σ_n from Σ_p under $\mathbf{u}/\mathbf{v}$ is given by

$$\Sigma_n = \{(s_2, s) \in S_2 \times S \mid \exists \mathbf{x} \in B^{|X|}, (\tilde{s}_2, \tilde{s}) \in \Sigma_p : \begin{array}{l} \mathbf{u} = \lambda_2^{(u)}(\tilde{s}_2, \mathbf{xv}), \quad s_2 = \delta_2(\tilde{s}_2, \mathbf{xv}), \\ s = \delta(\tilde{s}, \mathbf{x}) \end{array} \}.$$

Consider arbitrary $\mathbf{x} \in B^{|X|}$ and $(\tilde{s}_2, \tilde{s}) \in \Sigma_p$ such that $\mathbf{u} = \lambda_2^{(u)}(\tilde{s}_2, \mathbf{xv})$. If there are no such $\mathbf{x}$ and $(\tilde{s}_2, \tilde{s})$, then $\Sigma_n = \{\phi\}$. In this case, for an arbitrary pair of states of M_2 and M contained in $f(\Sigma_p)$, only condition (a) holds in the definition of the NDE-machine. Therefore, for all elements $\varsigma_p \in f(\Sigma_p)$, $\varsigma_n = \phi$ is the unique state which satisfies $T_N(\varsigma_p, \mathbf{u}, \mathbf{v}, \varsigma_n) = 1$. Thus we obtain $f(\Sigma_n) = \{\varsigma_n \mid \exists \varsigma_p \in f(\Sigma_p) : T_N(\varsigma_p, \mathbf{u}, \mathbf{v}, \varsigma_n) = 1\}$. Hence $T'_D(f(\Sigma_p), \mathbf{u}, \mathbf{v}, f(\Sigma_n)) = 1$.

Suppose there exist such $\mathbf{x}$ and $(\tilde{s}_2, \tilde{s}) \in \Sigma_p$ with the property that $\mathbf{u} = \lambda_2^{(u)}(\tilde{s}_2, \mathbf{xv})$. Then by the definition of the E-machine, $\lambda_2^{(z)}(\tilde{s}_2, \mathbf{xv}) = \lambda(\tilde{s}, \mathbf{x})$. Therefore, in the definition of the NDE-machine, the condition (a) and (b) do not hold and the first half of the condition (c) holds. Hence, Σ_n given above can be rewritten as

$$\Sigma_n = \{(s_2, s) \in S_2 \times S \mid \exists (\tilde{s}_2, \tilde{s}) \in \Sigma_p : T_N((\tilde{s}_2, \tilde{s}), \mathbf{u}, \mathbf{v}, (s_2, s)) = 1\}.$$

Thus by definition of T_D, $T'_D(f(\Sigma_p), \mathbf{u}, \mathbf{v}, f(\Sigma_n)) = 1$.

Conversely, suppose $T'_D(\Sigma_{D_p}, \mathbf{u}, \mathbf{v}, \Sigma_{D_n}) = 1$. We will show that the corresponding transition exists in the E-machine, i.e. $T(f^{-1}(\Sigma_{D_p}), \mathbf{u}, \mathbf{v}, f^{-1}(\Sigma_{D_n})) = 1$, where f^{-1} is the inverse of f. Note that the function $f^{-1}(\Sigma_{D_p})$ simply removes the state $\phi \in \mathcal{S}_N$ from the subset Σ_{D_p}, where in case $\Sigma_{D_p} = \{\phi\}$, $f^{-1}(\Sigma_{D_p}) = \{\phi\}$. We employ the assumption that $f^{-1}(\Sigma_{D_p}) \in \mathcal{S}$. The assumption does not affect the claim for the same reason above. If $\Sigma_{D_p} = \{\phi\}$, then $\Sigma_{D_n} = \{\phi\}$. Since $T(\{\phi\}, \mathbf{u}, \mathbf{v}, \{\phi\}) = 1$, the claim holds.

Consider the case where $\Sigma_{D_p} \neq \{\phi\}$. Since $f^{-1}(\Sigma_{D_p}) \in \mathcal{S}$, there exists some t such that $f^{-1}(\Sigma_{D_p}) \in \mathcal{S}^{(t)}$, where $\mathcal{S}^{(t)}$ is defined in Section 4.5.1. We will show that conditions (2) and (3) defined in the definition of $T^{(K)}$ in Section 4.5.1 hold, and thus $T^{(t+1)}(f^{-1}(\Sigma_{D_p}), \mathbf{u}, \mathbf{v}, f^{-1}(\Sigma_{D_n})) = 1$. Hereafter, let us denote $\Sigma_p = f^{-1}(\Sigma_{D_p})$ and $\Sigma_n = f^{-1}(\Sigma_{D_n})$. Consider arbitrary $\mathbf{x} \in B^{|X|}$ and $(\tilde{s}_2, \tilde{s}) \in S_2 \times S$ such that $(\tilde{s}_2, \tilde{s}) \in \Sigma_{D_p}$ and $\mathbf{u} = \lambda_2^{(u)}(\tilde{s}_2, \mathbf{xv})$. If there are no such $\mathbf{x}$ and $(\tilde{s}_2, \tilde{s})$, then the condition (2) trivially holds. Also in this case, for each $\varsigma_p = (\tilde{s}_2, \tilde{s}) \in \Sigma_{D_p}$, only the condition (a) holds in the construction of the NDE-machine T_N, and thus $\Sigma_{D_n} = \{\phi\}$. Since condition (3) trivially holds if $\Sigma_n = \{\phi\}$, we obtain $T^{(t+1)}(\Sigma_p, \mathbf{u}, \mathbf{v}, \Sigma_n) = 1$.

Suppose such $\mathbf{x}$ and $(\tilde{s}_2, \tilde{s})$ exist. Since $\kappa \notin \Sigma_{D_n}$, condition (b) does not hold for this pair $(\tilde{s}_2, \tilde{s})$ in the definition of the NDE-machine. Thus the first half of the condition (c) holds, and $\lambda_2(\tilde{s}_2, \mathbf{xv}) = \lambda(\tilde{s}, \mathbf{x})$. Since $(\delta_2(\tilde{s}_2, \mathbf{xv}), \delta(\tilde{s}, \mathbf{x})) \in \Sigma_{D_n}$, by definition of T_D, $(\delta_2(\tilde{s}_2, \mathbf{xv}), \delta(\tilde{s}, \mathbf{x})) \in \Sigma_n$, and the condition (2) holds.

For condition (3), consider an arbitrary $(s_2, s) \in \Sigma_n$. Since $(s_2, s) \in \Sigma_{D_n}$, there exists $(\tilde{s}_2, \tilde{s}) \in \Sigma_{D_p}$ such that $T_N((\tilde{s}_2, \tilde{s}), \mathbf{u}, \mathbf{v}, (s_2, s)) = 1$. Hence, condition (c) in the definition of the NDE-machine holds, and there exists $\mathbf{x} \in B^{|X|}$ such that $\mathbf{u} = \lambda_2^{(u)}(\tilde{s}_2, \mathbf{xv})$ and $(s_2, s) = (\delta_2(\tilde{s}_2, \mathbf{xv}), \delta(\tilde{s}, \mathbf{x}))$. Thus condition (3) holds, and we obtain $T^{(t+1)}(\Sigma_p, \mathbf{u}, \mathbf{v}, \Sigma_n) = 1$. ■

Thus, the E-machine can be obtained by applying an operation similar to the subset construction on the NDE-machine T_N. One might wonder why the operation like the subset construction is necessary, i.e. how is the set of behaviors contained in the E-machine related to those of the NDE-machine? The answer is that the NDE-machine contains *more* implementable behaviors than the E-machine. Specifically, in the NDE-machine, an implementable behavior is not permissible if there exists a pair (σ_u, σ_v) of sequences of U and V in the behavior which can lead the NDE-machine to κ, since it means that the corresponding sequence on the global output Z is inconsistent with what is required by M.[2] Therefore, we need to remove the set of pairs that have a possibility to lead the NDE-machine to κ. It is analogous to removing, or complementing the set of strings that have a possibility to lead a non-deterministic finite automaton to an accepting state, where κ is now treated as the accepting state. Hence, we employ the subset construction to remove those additional behaviors, and then guarantee that an arbitrary implementable behavior contained in the resulting machine (E-machine) is permissible. It is illustrated in the following example.

Example 4.6.1 *Consider M_2 and M shown in Figure 4.12, which are slightly different from those used in Example 4.4.1. The corresponding E-machine and the NDE-machine are shown in Fig-*

[2]Note that the pair (σ_u, σ_v) is not allowed even if it can also lead the NDE-machine to a state other than κ.

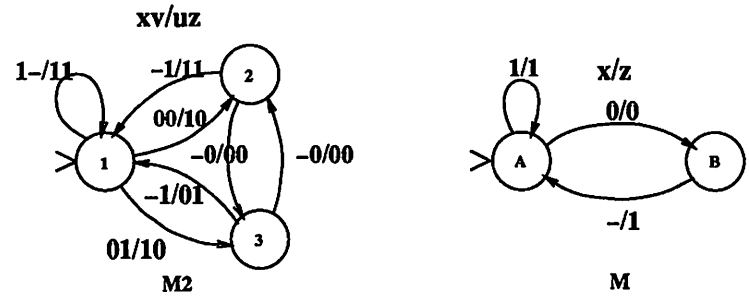

Figure 4.12: Example of M_2 and M

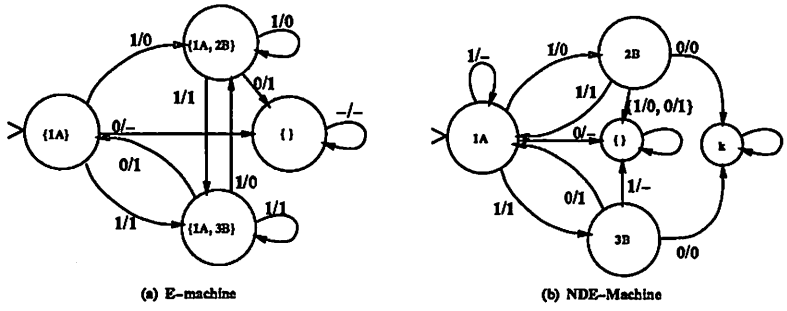

Figure 4.13: The E-machine (left) and the NDE-machine (right)

ure 4.13-(a) and Figure 4.13-(b) respectively.

Consider a behavior at M_1 which always outputs 0 for all input sequences. This is equivalent to setting the variable V to a constant 0, and thus the behavior is implementable. However, the behavior is not permissible since if a sequence σ_x of the global input X is set to $(0,0)$, then the corresponding pair of sequences (σ_u, σ_v) on U and V realized by the behavior and M_2 is given by $\sigma_u = (1,0)$ and $\sigma_v = (0,0)$, and thus the global output sequence σ_z is obtained as $\sigma_z = (0,0)$, while the global machine M requires that σ_z must be $(0,1)$. It is easy to see that the pair (σ_u, σ_v) above can lead the NDE-machine to the state κ through the states $1A$ and $2B$. Note that this behavior is not contained in the E-machine.

4.6.2 A Case where the NDE-machine Equals the E-machine

As just illustrated, the NDE-machine can contain behaviors that are implementable but not permissible. In order to remove such behaviors, we apply an operation similar to the subset construction to the NDE-machine. However, such an operation is necessary because of the fact that the NDE-machine is not pseudo non-deterministic, i.e. the automaton corresponding to the NDE-machine is non-deterministic. If the NDE-machine happens to be pseudo non-deterministic, then additional behaviors can be removed simply by deleting transitions which cause the NDE-machine to go to state κ, and thus an operation like the subset construction is not necessary. Intuitively, for a pseudo non-deterministic machine, this corresponds to complementing a deterministic finite automaton, where κ is treated as the accepting state and we want to remove set of strings that lead the automaton to κ.

It is then claimed that the resulting machine, i,e, the NDE-machine where transitions to κ have been removed, is isomorphic to the E-machine. This can be proved as follows. Suppose that the NDE-machine $(U, V, S_N, T_N, \varsigma_r)$ is pseudo non-deterministic. By definition of the NDE-machine, for each state ς_p and pair of input and output minterms $(\mathbf{u}, \mathbf{v})$, the NDE-machine always has at least one next state, i.e. a state ς_n such that $T_N(\varsigma_p, \mathbf{u}, \mathbf{v}, \varsigma_n) = 1$. Therefore, pseudo non-determinism implies that for each state ς_p and pair of input and output minterms $(\mathbf{u}, \mathbf{v})$, there exists exactly one state ς_n such that $T_N(\varsigma_p, \mathbf{u}, \mathbf{v}, \varsigma_n) = 1$. We first claim that the machine obtained by removing transitions to κ is isomorphic to the machine T'_D defined in the previous section, whose transitions are defined by Formula (4.2). In Formula (4.2), suppose that Σ_{D_p} consists of only two states of the NDE-machine, $\{\varsigma_p, \phi\}$, where one of them is ϕ. Then since the NDE-machine is pseudo non-deterministic, Σ_{D_n} defined by the formula for given $\mathbf{u}$ and $\mathbf{v}$ has exactly two states of the NDE-machine, where one of them is ϕ. Then Σ_{D_n} is included in a state of the machine T'_D if and only if it does not contain κ and is reachable from the reset state of T'_D. Since the reset state of T'_D is given by $\Sigma_{D_r} = \{\varsigma_r, \phi\}$, the machine T'_D is obtained by replacing each state ς_p of the NDE-machine by a set $\{\varsigma_p, \phi\}$, and by deleting transitions to κ. Hence T'_D is isomorphic to the machine obtained by deleting transitions to κ in the NDE-machine. By Theorem 4.6.1, T'_D is isomorphic to the E-machine, and thus it follows that if the NDE-machine is pseudo non-deterministic, then the E-machine can be obtained simply by deleting transitions to κ in the NDE-machine.

The remaining question is when does the NDE-machine become pseudo non-deterministic. One such case is the context discussed in Section 4.5.3, where the global inputs X directly drive M_1, as shown in Figure 4.9. If M_1 takes as input U and X, then the NDE-machine also has X

as input. Thus, if we apply the fixed point iteration to define $T_N^{(t+1)}(\varsigma_p, \mathbf{xu}, \mathbf{v}, \varsigma_n) = 1$ according to Equation (4.1), where the three conditions are checked for that particular $\mathbf{x} \in B^{|X|}$ given in $T_N^{(t+1)}$, then the next state ς_n is uniquely defined. Therefore, the NDE-machine is pseudo non-deterministic. Furthermore, the isomorphism between the NDE-machine and the E-machine provides another interpretation for the fact that in this case, the E-machine has a property that each state corresponds to a single pair of states of M_2 and M, as shown in Section 4.5.3.

4.7 Implementability of Interacting Machines

4.7.1 Implementability

As we have seen in the previous sections, the permissibility of M_1 requires that M_1 is implementable, i.e. there exists a pair of implementations for M_1 and M_2 where no combinational loop is created by connecting them together at U and V. Therefore, when a permissible machine is sought, we need to check whether the machine is implementable or not. In this section, we provide a condition on the implementability.

Let a completely specified deterministic machine $M_2 = (X \cup V, U \cup Z, S_2, \lambda_2, \delta_2, r_2)$ be given. We want to know if a completely specified deterministic machine $M_1 = (U, V, S_1, \lambda_1, \delta_1, r_1)$ is implementable with M_2. The key idea is to check the implementability by analyzing the *dependencies.*

Definition: Dependencies

For a set of Boolean variables $X = \{x_1, \ldots, x_n\}$ consider a function $f : B^n \rightarrow B$ defined with the input X. Given an input variable $x_i \in X$, f is **dependent** on x_i if $f|_{x_i=0} \neq f|_{x_i=1}$, where $f|_{x_i=0}$ designates the cofactor of f with respect to $x_i = 0$.

If f is not dependent on x_i, we say that f is independent of x_i. The dependency of f for an input x_i is related to whether it is possible to implement the function f with no combinational path from x_i to the output, where we define a combinational path as a sequence of gates which does not contain latches or flip-flops. More specifically, the following lemma is known.

Lemma 4.7.1 *Given a function $f : B^n \rightarrow B$ with the input $X = \{x_1, \ldots, x_n\}$, there exists an implementation for f such that there is no combinational path from x_i to the output, if and only if f is not dependent on x_i.*

Proof: Suppose there exists an implementation such that there is no combinational path from x_i to the output. Then for an arbitrary minterm $\mathbf{x} \in B^n$, the output value $f(\mathbf{x})$ does not change even if we flip the value of x_i in the minterm $\mathbf{x}$. Thus $f|_{x_i=0} = f|_{x_i=1}$.

Conversely, suppose that f is not dependent on x_i. Consider an implementation of f. If the implementation does not contain a combinational path from x_i to the output, the proof is done. Suppose there is a combinational path. We claim that the implementation given by setting x_i to a constant value, say 0, still implements f. Let $\tilde{f}$ be the function defined by the resulting implementation. Note that $\tilde{f}$ does not depend on x_i. The proof is done if we show that $\tilde{f}(\mathbf{x}) = f(\mathbf{x})$ for all $\mathbf{x} \in B^n$. Suppose $\tilde{f}(\mathbf{x}) \neq f(\mathbf{x})$. Then the value of x_i in the minterm $\mathbf{x}$ must be 1 since $\tilde{f}$ is obtained by setting $x_i = 0$ in f. Then $f|_{x_i=0} \neq f|_{x_i=1}$, which contradicts the fact that f is not dependent on x_i. ■

We now present a condition under which M_1 is implementable . Consider a directed bipartite graph $G(U \cup V, E)$, where the node set of G is divided into two classes U and V and a node of U (respectively a node of V) corresponds to a variable of the input variables U (respectively the output variables V) of M_1. The edges of G are defined as follows:

$$\begin{aligned} [u_i, v_j] \in E &\Leftrightarrow \lambda_1^{(v_j)} \text{ depends on } u_i, \\ [v_j, u_i] \in E &\Leftrightarrow \lambda_2^{(u_i)} \text{ depends on } v_j, \end{aligned}$$

where we denote by $\lambda_1^{(v_j)}$ the function of the j-th output variable v_j in M_1. The graph G is referred to as a *dependency graph.*

Theorem 4.7.1 *M_1 is implementable if and only if G is acyclic.*

Proof: Suppose M_1 is implementable. Then there exists a pair of implementations (C_1, C_2) for M_1 and M_2 respectively which does not create a combinational loop. Let $G_c(U \cup V, E_c)$ be a directed bipartite graph with the same node set of G, where the edges are defined as follows:

$$\begin{aligned} [u_i, v_j] \in E &\Leftrightarrow \text{there exists a combinational path from } u_i \text{ to } v_j \text{ in } C_1, \\ [v_j, u_i] \in E &\Leftrightarrow \text{there exists a combinational path from } v_j \text{ to } u_i \text{ in } C_2. \end{aligned}$$

Since the implementation does not contain a combinational loop, G_c is acyclic. Now, if $\lambda_1^{(v_j)}$ depends on u_i, Lemma 4.7.1 implies that C_1 has a combinational path from u_i to v_j. A similar argument holds for $\lambda_2^{(u_i)}$. Thus $E \subseteq E_c$. Hence G is a subgraph of G_c, and G is acyclic.

Conversely, suppose G is acyclic. Consider an implementation C_1 of M_1, where for each v_j, the function $\lambda_1^{(v_j)}$ is implemented independently, as described in Lemma 4.7.1, so that no gate

of the implementation is used in an implementation of another output of M_1, say v_k with $j \neq k$. Then C_1 has a property that there is a combinational path from u_i to v_j if and only if $\lambda_1^{(v_j)}$ depends on u_i. Similarly, let C_2 be an implementation of M_2 such that there is a combinational path from v_j to u_i if and only if $\lambda_2^{(u_i)}$ depends on v_j. The proof is done if we show that (C_1, C_2) does not create a combinational loop. Suppose to the contrary that there exists a combinational loop $c = (v_{j_0}, u_{i_0}, \ldots, v_{j_k}, u_{i_k}, v_{j_0})$. Then for each $l, 0 \leq l \leq k$, $\lambda_2^{(u_{i_l})}$ depends on v_{j_l}. Similarly, $\lambda_1^{(v_{j_l})}$ depends on $u_{i_{l-1}}$, where we define $u_{i_{-1}} = u_{i_k}$. Thus the cycle c exists in G, which conflicts with the fact that G is acyclic. ∎

Since the cyclicity of a directed bipartite graph can be checked in polynomial time in the size of the graph [55], we can efficiently check the implementability of M_1. Note that if either M_1 or M_2 is of Moore type, then G is always acyclic, and M_1 is implementable. Note also that the theorem above is claimed under the assumption that both U and V are Boolean variables. It is known that for symbolic variables, there exist cases where cyclic dependency observed for symbolic variables can be broken in an actual implementation by carefully encoding the variables [7].

4.7.2 Unimplementable Machines in the E-machine

In general, not all machines contained in the E-machine are implementable. By definition of implementable machines, if a machine M_1 contained in the E-machine is not implementable, then *any* implementation of M_2 will create a combinational loop for that particular M_1. Thus, for given M and M_2, if the resulting E-machine contains no implementable machines, then it is impossible to realize a behavior of M without combinational loops, as long as the behavior of M_2 is used.

We discuss what can be done with unimplementable machines of the E-machine. Specifically, we show that for a machine M_1 contained in the E-machine but not implementable, if M_1 satisfies a certain condition, then it is possible to realize a behavior of M with no combinational loops, as long as we are allowed to modify the behavior of M_2.

Let $M_1 = (U, V, S_1, \lambda_1, \delta_1, r_1)$ be a machine contained in the E-machine. Suppose that M_1 is not implementable. Suppose also that M_1 satisfies the following *stability* property:

Property 4.7.1 *For all pairs of states,* $(s_1, s_2) \in S_1 \times S_2$, *and for all* $\mathbf{x} \in B^{|X|}$, *there exists at least one* $(\mathbf{u}, \mathbf{v}) \in B^{|U|} \times B^{|V|}$ *such that* $\mathbf{v} = \lambda_1(s_1, \mathbf{u})$ *and* $\mathbf{u} = \lambda_2^{(u)}(s_2, \mathbf{xv})$.

This property can be checked by first computing a function $St : S_1 \times S_2 \rightarrow B$ such that

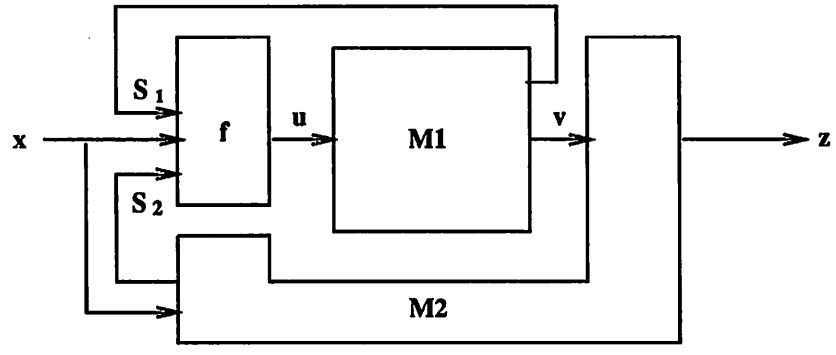

Figure 4.14: Modification for Unimplementable Machines

$St(s_1, s_2) = 1$ if and only if

$$\forall \mathbf{x} \in B^{|X|} : \exists (\mathbf{u}, \mathbf{v}) \in B^{|U|} \times B^{|V|} : \mathbf{v} = \lambda_1(s_1, \mathbf{u}) \text{ and } \mathbf{u} = \lambda_2^{(u)}(s_2, \mathbf{xv}),$$

and then checking whether St is tautologically equal to 1.

Consider a pair of implementations C_1 and C_2 for M_1 and M_2, respectively. Since M_1 is not implementable, the implementation made of C_1 and C_2 creates a combinational loop. Assume that we can scan the latches of C_1, i.e. it is possible to observe externally the state in which M_1 stays. Then we modify C_2 so that the resulting implementation has no combinational loop and realizes a behavior of M.

Consider a function whose inputs are the global inputs X as well as the states of M_1 and M_2, and the outputs are U. We denote the function by $f : S_1 \times S_2 \times B^{|X|} \rightarrow B^{|U|}$. For given $(s_1, s_2) \in S_1 \times S_2$ and $\mathbf{x} \in B^{|X|}$, the output $\mathbf{u} = f(s_1, s_2, \mathbf{x})$ is defined so that there exists $\mathbf{v} \in B^{|V|}$ such that $\mathbf{v} = \lambda_1(s_1, \mathbf{u})$ and $\mathbf{u} = \lambda_2^{(u)}(s_2, \mathbf{xv})$. Since M_1 satisfies Property 4.7.1, $f(s_1, s_2, \mathbf{x})$ is defined for every input. Let C_3 be an implementation of f. Note that C_3 is a combinational logic circuit. We break the connection from C_2 to C_1 at U by eliminating the outputs U from M_2, and let C_3 drive C_1, as shown in Figure 4.14. Since all the feedbacks from C_1 to C_3 and from C_2 to C_3 are to see the states of M_1 and M_2, there is no combinational loop in the resulting implementation.

Let us regard the circuit made of C_2 and C_3 as an implementation of a single deterministic finite state machine $\tilde{M}_2$. Note that the state space of $\tilde{M}_2$ is identical to that of M_2. By the construction of the function f, it is guaranteed that for all pairs of states, $(s_1, s_2) \in S_1 \times S_2$, and for all $\mathbf{x} \in B^{|X|}$, the pair $(\mathbf{u}, \mathbf{v}) \in B^{|U|} \times B^{|V|}$ realized by M_1 and $\tilde{M}_2$ has the property that

$\mathbf{v} = \lambda_1(s_1, \mathbf{u})$ and $\mathbf{u} = \lambda_2^{(u)}(s_2, \mathbf{xv})$. Furthermore, the state transition of $\tilde{M}_2$ does not depend on the states of M_1. Namely, for a given state s_2 of $\tilde{M}_2$ and input $\mathbf{xv} \in B^{|X \cup V|}$, the next state to which $\tilde{M}_2$ moves is uniquely defined and is given by $\delta_2(s_2, \mathbf{xv})$. Since M_1 is contained in the E-machine, exactly the same proof of Theorem 4.5.1 holds to claim that for an arbitrary sequence σ of $B^{|X|}$, the output sequence realized by $M_1 \times \tilde{M}_2$ can be realized by M, and therefore the behavior of $M_1 \times \tilde{M}_2$ is contained in M. We state this fact as a theorem below.

Theorem 4.7.2 *For a machine M_1 contained in the E-machine, suppose M_1 is not implementable and satisfies Property 4.7.1 above. Then for an arbitrary pair of implementations C_1 and C_2 for M_1 and M_2, if an additional circuitry C_3 given above is attached, the resulting circuitry has no combinational loop and its behavior is contained in M.*

4.8 Experimental Results

The method of computing the transition relation T of the E-machine has been implemented, and we conducted some experiments. The current implementation is limited to the case that the global machine M is deterministic, and thus a state of the E-machine corresponds to a subset of pairs of states of M_2 and M. Binary decision diagrams (BDD's) [12] are used to represent the transition relations of M_2 and M, where a set of states of each machine is represented by binary variables using log-based encodings. All set operations, such as intersection, union, complement, set comparisons, as well as quantifications, are performed on BDD's. We first compute the relation $T^{(K)}$ and then T. One straightforward way of computing $T^{(K)}$ is to first compute the relation given by the condition (2) and (3) of the definition of $T^{(K)}$ shown in Section 4.5.1, and then restrict it to the states that T can be led to by some sequences of $B^{|U|}$. However, since the total number of states of the finite state machine given by the conditions (2) and (3) is exponential in $|S_2||S|$, the BDD representing the transition relation of the machine may be too large. Instead, we perform a fixed point computation as stated in Section 4.5.1, where at each step t, instead of the set $\mathcal{S}^{(t)}$, we use a set which contains $\mathcal{S}^{(t)} \cap \neg\mathcal{S}^{(t-1)}$, is contained in $\mathcal{S}^{(t)}$, and is represented by a minimal-sized BDD. Such a set is computed by a BDD operation similar to the one known as generalized cofactor [13], and a detailed description is found in [52]. During the computation, we need to see if a given pair of states $(\tilde{s}_2, \tilde{s}) \in S_2 \times S$ is a member of Σ_p. For this purpose, we use a characteristic function $\chi(\tilde{s}_2, \tilde{s}, \Sigma_p)$ which is equal to 1 if and only if $(\tilde{s}_2, \tilde{s}) \in S_2 \times S$ is a member of Σ_p. However, a BDD representing the function itself, or a BDD obtained at an intermediate stage of the computations

	M_1			M_2					
	In	Out	S	In	Out	S	E-machine	Time	Iterations
mc9	2	1	4	3	5	4	4	0.1	3
mt52	5	6	22	7	7	4	9	0.4	4
tm02	4	4	20	5	6	20	10	0.9	3
tm32	3	4	19	6	5	3	9	1.7	7
pm11	8	8	26	10	10	24	9	1.9	5
tm01	4	4	20	5	6	20	10	2.9	3
am9	6	6	25	7	8	4	13	3.1	10
pm12	8	8	26	10	10	24	7	4.3	4
e69	2	1	4	5	8	8	8	4.5	3
L4	8	6	20	11	14	14	6	5.0	5
mt51	5	6	22	7	7	4	16	6.9	7
L3	2	3	76	7	3	19	17	8.5	8
e4bp1	5	5	24	6	9	14	11	9.9	10
pm33	6	6	25	7	8	4	21	10.2	11
e6tm	4	4	20	5	6	8	21	10.5	7
pm03	2	4	11	6	4	14	15	14.2	14
tm31	3	4	19	6	5	13	9	14.2	4
pm31	6	6	25	7	8	4	22	20.6	8
e4at2	5	4	21	6	9	14	14	27.4	13
pm50	2	4	11	6	4	14	22	37.3	17
s3p1	5	5	24	7	7	13	38	43.3	11
pm41	2	4	11	6	4	14	33	132.0	22

Table 4.1: Experimental Results

using the function, could be fairly large in practice. Therefore, we represent χ by several BDD's whose union forms χ. We modified the formula given in (2) and (3) in the definition of $T^{(K)}$ so that the union of these BDD's are taken as late as possible by applying other commutative operations earlier. These heuristics seem to be effective in controlling the size of BDD's.

Using the procedure implemented as stated above, we conducted some experiments. The examples were chosen mainly from mcnc91 benchmark examples. The objective of the experiment was to determine the size of machines that can be handled by the current implementation, as well as the size of the resulting E-machine T since its state-space size could be exponential in $|S_2||S|$. The experiments start with choosing two finite state machines, M_1 and M_2, in the structure shown in Figure 4.1. Both machines are completely specified deterministic finite machines. We then make an arbitrary connection between the machines, i.e. a subset of the input variables (the output variables,

respectively) of M_2 is arbitrarily chosen to connect with the output (the input, respectively) of M_1, so that M_1 is implementable for M_2. Then the E-machine is constructed by the proposed procedure, where a preprocessor is first invoked to obtain the product machine M of M_1 and M_2, so that $M_1 \times M_2$ is used as the specification M.

The results on these examples are shown in Table 4.1. Each row of the table corresponds to a single experiment, where **In**, **Out**, and **S** designate the number of input variables, the number of output variables, and the number of states respectively. **Time** is the CPU time used for each experiment in seconds on a DECstation 5000/240. **Iterations** shows the number of iterations required in the fixed point computation of $T^{(K)}$. During the experiments, we realized that the size of the resulting E-machine and the required CPU time vary widely depending upon the connections chosen between M_1 and M_2. Thus we cannot make any general statement on the size of the E-machine that we can handle in practice. Nevertheless, for these experimental results, we see that the number of states of T is negligibly smaller than $2^{|S_2||S|}$. This is not surprising in the sense that a state s_1 of the E-machine corresponds to a subset of $S_2 \times S$ with the property that M_2 and M are led to exactly the states of the subset by the input sequences of $B^{|X|}$ and the sequences of $B^{|V|}$ realized by transitions from the reset state of the E-machine to s_1. Thus if there exists a pair of states (s_2, s) not led to by any input sequence, then any of the subsets of $S_2 \times S$ which contains the pair will not appear in the E-machine, where there are $2^{(|S_2||S|-1)}$ such subsets.

4.9 Concluding Remarks

In this chapter, we addressed the problem of computing and representing the complete set of permissible sequential behaviors, where two finite state machines are interacting with each other as shown in Figure 4.1. We showed that the complete set can be computed and represented by a single non-deterministic finite state machine. The machine is called the E-machine and its transition relation is computed by a fixed point computation. We also considered the problem of implementing interacting finite state machines without introducing combinational loops, and provided a necessary and sufficient condition under which given machines are implementable. The proposed procedure for computing the E-machine was implemented and experimental results were presented.

In the following chapter, we address the problem of minimizing E-machines, i.e. finding the *best* permissible behavior of M_1 for given M_2 and M.

Chapter 5

Minimization of Pseudo Non-Deterministic FSM's

5.1 Introduction

In the previous chapter, we addressed the problem of optimizing a system of interacting finite state machines. Specifically, we considered how to find a set of sequential behaviors that can be realized at a particular component so that the resulting behavior of the entire system meets the specification. We called each of such behaviors a permissible behavior at the component, and showed that the complete set of permissible behaviors can be computed and represented by a single non-deterministic finite state machine, called the *E-machine*. In this chapter, we consider how to find an optimum permissible behavior, where we use as the cost function the number of states of a finite state machine required to represent a given behavior.

The key theorem on the property of the E-machine we derived in the previous chapter is that the set of implementable behaviors given in the E-machine precisely provides the complete set of permissible behaviors. Therefore, an optimum permissible behavior is given by finding a least-cost behavior over all the implementable behaviors contained in the E-machine. This is the problem we are concerned with in this chapter.

Note the difference of this problem from a problem known as *reduction of non-deterministic finite automata*, which finds a minimum-state automaton with the same language for a given non-deterministic finite automaton [21]. Our problem does not require the behaviors of the original finite state machine to be preserved during the minimization.

The problem is analogous to the state minimization of deterministic finite state machines, which finds a minimum-state deterministic machine whose behavior is contained in a given deterministic finite state machine [1, 23, 24, 40, 44]. Our problem is more general in that a given machine is non-deterministic. We also need to take into account the implementability of a behavior.

As shown in Section 4.6, the E-machine has a special type of non-deterministic finite state machine called a *pseudo non-deterministic machine.* Therefore, it is sufficient for our application to consider the problem only for the case where the given machine is pseudo non-deterministic. We show that the property of pseudo non-determinism can be effectively used to solve the problem.

In this chapter, we first present a theoretical analysis of the problem, in which we show how the basic concepts developed for the state minimization of deterministic machines can be generalized for our problem. The analysis leads to an exact formulation for solving *the state minimization of a pseudo non-deterministic machine*, i.e. find a minimum-state deterministic finite state machine whose behavior is contained in a given pseudo non-deterministic machine. We then discuss how to deal with the implementability of behaviors, and present an exact method for finding an optimum permissible behavior in the E-machine.

We also propose a heuristic approach for the state minimization of pseudo non-deterministic finite state machines. This procedure has been implemented with a restriction that we focus only on Moore behaviors [38], behaviors where the outputs depend only on the internal states of machines, and not on the inputs. The restriction was made in order to guarantee that the resulting behavior is implementable. This procedure has been implemented and experimental results are presented.

5.2 The Problem

5.2.1 Minimization of E-machines

In the previous chapter, we considered a system of interacting two finite state machines M_1 and M_2 as shown in Figure 5.1. The specification, or a set of behaviors allowed to realize at the entire system, is given by a non-deterministic finite state machine M. The problem is that for a given M_2 and M, find a set of behaviors that can be realized at M_1 so that the behavior of the entire system meets the specification M. Each of such behaviors is called a *permissible* behavior at M_1. Specifically, a behavior at M_1 is said to be *permissible* if it is *implementable*, i.e. there exist implementations for M_1 and M_2 respectively with which no combinational loop is created in the

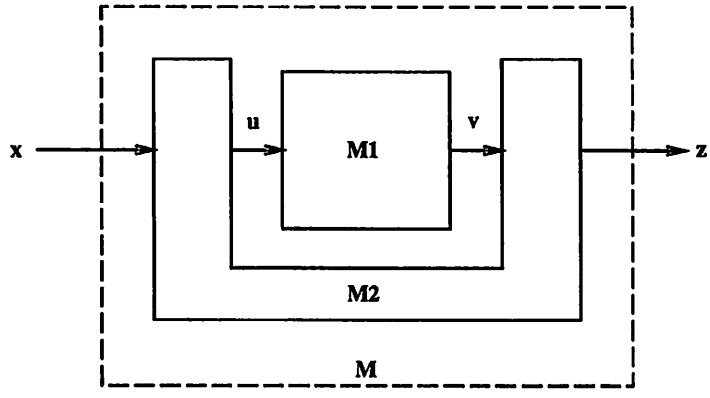

Figure 5.1: Interaction between Two Machines

resulting implementation, and the behavior composed of M_1 and M_2 is *contained* in M. Formal definitions of the terminology used in this chapter are given in Section 4.2 and in Section 4.3.

It was shown that the complete set of behaviors permissible at M_1 can be captured by a single non-deterministic finite state machine, called the *E-machine*. In this chapter, we find an optimum permissible behavior for M_1. As the cost function, we use the number of states of a finite state machine required to represent a given behavior. By Corollary 4.5.3, we see that the set of implementable behaviors given in the E-machine computed for M_1 precisely provides the complete set of permissible behaviors. Thus, an optimum permissible behavior is given by finding a behavior that can be represented by a completely specified deterministic machine with the minimum number of states over all the implementable behaviors contained in the E-machine.

Also shown in the previous chapter is that the E-machine has a special property called the *pseudo non-determinism* defined as follows :

Definition: Pseudo Non-Deterministic Finite State Machines

A finite state machine (I, O, S, T, r) is said to be **pseudo non-deterministic** if for all $(s_p, \mathbf{u}, \mathbf{v}) \in S \times B^{|I|} \times B^{|O|}$, $T(s_p, \mathbf{u}, \mathbf{v}, s_n) = 1 \Rightarrow s_n$ is unique.

Note that a pseudo non-deterministic finite state machine is a special type of non-deterministic machine. Also, a completely specified deterministic machine is trivially pseudo non-deterministic.

Hence, we consider the problem given by the following general statement: *for a given*

pseudo non-deterministic finite state machine $T = (U, V, S, T, r)$ *and a completely specified deterministic finite state machine* $M_2 = (X \cup V, U \cup Z, S_2, \lambda_2, \delta_2, r_2)$, *find a behavior represented by a completely specified deterministic machine with the minimum number of states over all the implementable behaviors contained in* T, *where the implementability is defined against* M_2. In the rest of the chapter, this problem is referred to as the *minimization of the E-machine* .

5.2.2 State Minimization of Pseudo Non-Deterministic Machines

A problem related to the above is one called *the state minimization of pseudo non-deterministic finite state machines*. The problem is that for a given pseudo non-deterministic finite state machine T, find a a behavior represented by a completely specified deterministic machine with the minimum number of states over all the behaviors contained in T. The difference between this problem and the original problem is that the implementability must be taken into account in the original.

The state minimization of pseudo non-deterministic finite state machines is a variation of the problem generally referred to as the state minimization of finite state machines. Specifically, it is a subproblem of the case where a given machine is a general non-deterministic machine, and is a generalization of the case where a given machine is deterministic. The research for the deterministic case has been done extensively [1, 23, 24, 40, 44], while little has been done for the non-deterministic case.

In the following section, we present a theoretical analysis of the state minimization of pseudo non-deterministic machines, in which we show how the basic concepts developed for the deterministic case can be generalized for the problem. The theory provides a basis for exact formulations of both this problem and our original problem, i.e. the minimization of the E-machine.

Before starting the theoretical analysis, we first argue the theoretical generality of the state minimization of pseudo non-deterministic machines. In other words, we show that the state minimization of general non-deterministic finite state machines can be reduced to that of pseudo non-deterministic machines, and thus the assumption that a given machine is pseudo non-deterministic does not affect the generality of the problem of the state minimization of finite state machines. Specifically, we claim that for an arbitrary finite state machine, there exists a pseudo non-deterministic finite state machine with exactly the same set of behaviors.

Theorem 5.2.1 *For a given finite state machine* $T = (U, V, S, T, r)$, *there exists a pseudo non-deterministic finite state machine which represents the same set of behaviors of* T.

Proof: The basic idea of the proof is to convert a non-deterministic finite state machine to a non-deterministic finite automaton by combining inputs and outputs. Then determinize the non-deterministic automaton using the subset construction and map the resulting automaton back to a finite state machine, observing that the result is pseudo non-deterministic.

Let Σ be a set with the cardinality equal to $2^{|U|+|V|}$. Let $\alpha : B^{|U|} \times B^{|V|} \rightarrow \Sigma$ be a one-to-one mapping from $B^{|U|} \times B^{|V|}$ to Σ. Namely, for $(\mathbf{u}, \mathbf{v}) \in B^{|U|} \times B^{|V|}$ and $(\tilde{\mathbf{u}}, \tilde{\mathbf{v}}) \in B^{|U|} \times B^{|V|}$, if $(\mathbf{u}, \mathbf{v}) \neq (\tilde{\mathbf{u}}, \tilde{\mathbf{v}})$, then $\alpha(\mathbf{u}, \mathbf{v}) \neq \alpha(\tilde{\mathbf{u}}, \tilde{\mathbf{v}})$. Note that the inverse function α^{-1} is well-defined and is also one-to-one.

Consider a finite automaton $T^a = (\Sigma, S, T^a, S, r)$ that has the same state set as T. The transition relation $T^a : S \times \Sigma \times S \rightarrow B$ has the property that $T(\tilde{s}, \mathbf{u}, \mathbf{v}, s) = 1$ if and only if $T^a(\tilde{s}, \alpha(\mathbf{u}, \mathbf{v}), s) = 1$. We assume that every state of T^a is a final state. Note that T^a is in general a non-deterministic finite automaton. Let $T^a_D = (\Sigma, S_D, T^a_D, r_D)$ be a deterministic automaton with the same language of T^a. Then consider a finite state machine $T_D = (U, V, S_D, T_D, r_D)$ that has the same state set as T^a_D. The transition relation $T_D : S_D \times B^{|U|} \times B^{|V|} \times S_D \rightarrow B$ is defined by the property that $T_D(\tilde{s}, \mathbf{u}, \mathbf{v}, s) = 1$ if and only if $T^a_D(\tilde{s}, \alpha(\mathbf{u}, \mathbf{v}), s) = 1$. It is easy to see that T_D is a pseudo non-deterministic finite state machine.

We claim that T_D represents the same set of behaviors as T. Namely, a behavior is contained in T if and only if it is contained in T_D. For a behavior $\mathcal{B}$ contained in T, consider an arbitrary pair of sequences (σ_u, σ_v) of $\mathcal{B}$, where $\sigma_u = (\mathbf{u}_0, \ldots, \mathbf{u}_k)$ and $\sigma_v = (\mathbf{v}_0, \ldots, \mathbf{v}_k)$. Then since all the states are final, the automaton T^a accepts the sequence of Σ given by $(\alpha(\mathbf{u}_0, \mathbf{v}_0), \ldots, \alpha(\mathbf{u}_k, \mathbf{v}_k))$, and so does T^a_D. Hence (σ_u, σ_v) is realized in T_D. Since (σ_u, σ_v) is an arbitrary pair, the behavior $\mathcal{B}$ is contained in T_D. A similar argument holds to claim the converse, which completes the proof.

■

Therefore, in terms of behaviors represented by single machines, we see that pseudo non-deterministic machines are as expressive as general non-deterministic machines. Note that this property does not hold for deterministic machines, i.e. there exists in general a non-deterministic machine whose set of behaviors cannot be represented by any single deterministic machine. This is illustrated by the following example.

Example 5.2.1 *Consider the behavior represented by the non-deterministic finite state machine T given in Figure 5.2. Suppose that there exists a deterministic machine M representing the same set of behaviors. Then at the reset state, M must output* 1 *for the input* 0 *and* 0 *for the input* 1. *Now, consider the next state that M moves to for the input* 1 *when it stays at the reset state. Since M is*

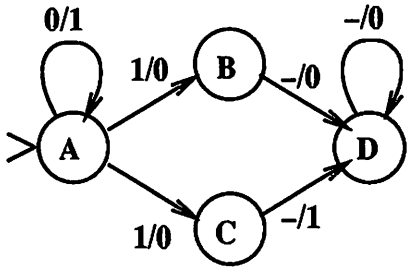

Figure 5.2: A Non-Deterministic Machine whose Behaviors cannot be Represented by Single Deterministic Machines

deterministic, either M can move to any state (in case M is incompletely specified), or it moves to a single state. If it can move to any state, then M may stay at the reset state for the input 1. *It follows that for the input sequence* 110, *M may first output* 0 *staying at the reset state, next output* 0 *again still staying at the reset state, and then output* 1 *for the third input* 0. *However, the resulting output sequence* 001 *is not allowed for the input sequence* 110 *in the original machine T. Therefore, for the input* 1 *at the reset state, M must move to a single state, say s. Since M represents the same set of behaviors as T, the outputs of the machine M at the state s must coincide with those of the states B and C in the original machine T. Specifically, the outputs at the state s must be that either M always outputs* 0 *for all the inputs, or it always outputs* 1 *for all the inputs. Therefore, we must specify more than one possible output for each input. However, since M is a deterministic machine, only the case where multiple outputs can be specified for a given input is that the output is* unspecified *for the input. Namely, we must specify in such a way that at the state s, M may output either* 0 *or* 1 *for each input. However, this specification allows the case that M outputs* 0 *for an input* 0 *and outputs* 1 *for an input* 1. *We see that this case should not be allowed in order for the outputs at the state s to coincide with those of the states B and C in the original machine T, since T's outputs at these states are either always equal to* 0 *or always equal to* 1. *Hence, there is no way to correctly specify the outputs of M at the state s. It follows that there is no single deterministic machine whose set of behaviors is identical with that of the original non-deterministic machine T.*

Note also that the construction given in the proof of the theorem above uses the subset construction for determinizing a non-deterministic finite automaton. Therefore, even though pseudo

non-deterministic finite state machines are as expressive as non-deterministic machines, the number of states of a pseudo non-deterministic machine could be exponentially larger than that of the corresponding non-deterministic machine.

5.3 Feasible Machines

5.3.1 Feasible Machines

In this section, we consider how to find a set of contained behaviors for a given pseudo non-deterministic finite state machine T. In other words, we want to establish a correspondence between the set of contained behaviors and the original machine T, so that we can interpret each of the contained behaviors in terms of the original machine.

We establish such a correspondence using a set of completely specified deterministic finite state machines, so that the set of those deterministic machines precisely represents the set of behaviors contained in T. Such a deterministic machine is called a *feasible machine*, and is defined as follows:

Definition: Feasible Machines

A completely specified deterministic finite state machine $M_1 = (U, V, S_1, \lambda_1, \delta_1, r_1)$ is said to be **feasible** if for each state $s_1 \in S_1$, there exists a subset $\Sigma(s_1) \subseteq S$ with the following property:

(a) $r \in \Sigma(r_1)$,

(b) $\forall(\tilde{s}_1, \mathbf{u}) \in S_1 \times B^{|U|} : \forall \tilde{s} \in \Sigma(\tilde{s}_1) : \exists s \in \Sigma(\delta_1(\tilde{s}_1, \mathbf{u}))$ s.t. $T(\tilde{s}, \mathbf{u}, \lambda_1(\tilde{s}_1, \mathbf{u}), s) = 1$, (5.1)

Associated with each state of a feasible machine M_1 is a set of states of the original pseudo non-deterministic machine T. The condition (a) means that the set of states of T corresponding to the reset state of M_1 must contain the reset state of the original machine. The condition (b) requires that for each state s_1 of a feasible machine M_1 and for each input minterm $\mathbf{u} \in B^{|U|}$, it is possible to move in the original machine T from any of the states of T associated with s_1 to some of the states associated with $\delta_1(\tilde{s}_1, \mathbf{u})$, the next state of s_1 in M_1 under the input $\mathbf{u}$, with the same output $\lambda_1(\tilde{s}_1, \mathbf{u})$. This condition is analogous to the *closure constraint* defined for the deterministic case [40].

5.3.2 Properties of Feasible Machines

The objective in this section is to claim that

1. the complete set of behaviors contained in T is precisely the set of feasible machines, and
2. a minimum state behavior contained in T is given by a feasible machine with the minimum number of states.

We first prove that the behavior represented by a feasible machine is contained in the original pseudo non-deterministic machine T.

Lemma 5.3.1 *Consider a feasible machine $M_1 = (U, V, S_1, \lambda_1, \delta_1, r_1)$. For a state $s_1 \in S_1$, let $\mathcal{B}_{s_1} = \{(\sigma_u, \sigma_v) \mid (\sigma_u, \sigma_v)$ is realized in M_1 at $s_1.\}$. Then $\mathcal{B}_{s_1}$ is a behavior between U and V such that for all $(\sigma_u, \sigma_v) \in \mathcal{B}_{s_1}$ and for all $s \in \Sigma(s_1)$, (σ_u, σ_v) is realized at s in T.*

Proof: The proof is done by induction on k for an arbitrary pair of sequences $(\sigma_u, \sigma_v) \in \mathcal{B}_{s_1}$ with $|\sigma_u| = k$. What we claim is that for an arbitrary state $s \in \Sigma(s_1)$ of T, there exists a state $s^{(k)}$ of T such that σ_u can lead T from s to $s^{(k)}$ with the output sequence σ_v. Thus (σ_u, σ_v) is realized by T. We also show that $s^{(k)} \in \Sigma(s_1^{(k)})$, where $s_1^{(k)}$ is the state of M_1 led to by σ_u from s_1.

The statement is true for $k = 1$, due to the condition (b) of feasible machines. Suppose that the statement is true for $k-1$, where $k > 1$. Consider an arbitrary pair $(\sigma_u, \sigma_v) \in \mathcal{B}_{s_1}$ with $|\sigma_u| = k$ and an arbitrary state s of T contained in $\Sigma(s_1)$. Let $(\tilde{\sigma}_u, \tilde{\sigma}_v)$ be the prefix pair of (σ_u, σ_v) of length $k-1$. Denote $\sigma_u = \tilde{\sigma}_u \mathrm{u}$ and $\sigma_v = \tilde{\sigma}_v \lambda_1(s_1^{(k-1)}, \mathrm{u})$. By the induction hypothesis, there exists a state $s^{(k-1)} \in \Sigma(s_1^{(k-1)})$ such that $\tilde{\sigma}_u$ can lead T from s to $s^{(k-1)}$ with the output sequence $\tilde{\sigma}_v$. Then by condition (b), there exists $s^{(k)} \in \Sigma(s_1^{(k)})$ for which $T(s^{(k-1)}, \mathrm{u}, \lambda_1(s_1^{(k-1)}, \mathrm{u}), s^{(k)}) = 1$. Therefore, σ_u can lead T from s to $s^{(k)} \in \Sigma(s_1^{(k)})$ with the output sequence σ_v. Thus the statement is true. ■

This lemma claims that the behavior of a feasible machine is contained in T, since the lemma holds at the reset state r_1 of M_1. Conversely, it is claimed that for every behavior contained in T, there exists a feasible machine with the behavior.

Theorem 5.3.1 *Given a pseudo non-deterministic finite state machine T, a behavior is contained in T if and only if it is represented by a feasible machine.*

Proof: It immediately follows by Lemma 5.3.1 that a feasible machine represents a behavior contained in T. We prove the converse. Consider a completely specified deterministic machine

$M_1 = (U, V, S_1, \lambda_1, \delta_1, r_1)$ whose behavior is contained in T. Suppose, without loss of generality, that every state of M_1 is reachable. We consider a procedure shown in Figure 5.3 which takes M_1 as input and returns another deterministic machine M_1'. We claim that M_1' is a feasible machine equivalent to M_1. The procedure first duplicates the machine M_1, and modifies the duplicated machine M_1' during the procedure. Specifically, it processes one state $\tilde{s}_1$ of M_1' at a time and for each input minterm $\mathbf{u} \in B^{|U|}$, associate a subset $\Sigma(s_1)$ of the states of T with the next state $s_1 = \delta_1'(\tilde{s}_1, \mathbf{u})$, where the corresponding next state may be changed to another possibly new state if necessary. Note that when a state $\tilde{s}_1$ is processed, a subset $\Sigma(\tilde{s}_1) \subseteq S$ has been already defined and associated with $\tilde{s}_1$. The notation $E(s_1)$ used in the procedure designates the equivalence class that a state s_1 belongs to. The equivalence classes are originally defined for the machine M_1. When M_1 is duplicated, the equivalence class is associated with each state s_1 of the duplicated machine M_1'. When a new state $\hat{s}_1$ is created, we define the transitions of the state so that it is equivalent to some state s_1 of the original machine M_1, and the equivalence class $E(\hat{s}_1)$ is set to $E(s_1)$.

The procedure uses two functions $C(s^*, E(s_1))$ and $N(\tilde{s}_1, \mathbf{u}, s_1)$. $C(s^*, E(s_1))$ is a characteristic function defined for a subset s^* of the states of T and an equivalence class $E(s_1)$ of M_1. It is 1 if and only if an arbitrary pair of input and output sequences (σ_u, σ_v) realized at the equivalence class $E(s_1)$ of M_1 can be realized in T at every state s of s^*. Thus the function C indicates if the subset s^* can have the same behavior as the equivalence class $E(s_1)$. The function $N(\tilde{s}_1, \mathbf{u}, s_1)$ is defined for a state $\tilde{s}_1$ of M_1', a minterm $\mathbf{u} \in B^{|U|}$, and the next state s_1 of $\tilde{s}_1$ in M_1' under the input $\mathbf{u}$, i.e. $s_1 = \delta_1'(\tilde{s}_1, \mathbf{u})$. It returns a non-empty subset s^* of the states of T with the property that for each $\tilde{s} \in \Sigma(\tilde{s}_1)$, there exists $s \in s^*$ such that $T(\tilde{s}, \mathbf{u}, \lambda_1'(\tilde{s}_1, \mathbf{u}), s) = 1$ and $C(s^*, E(s_1)) = 1$. Note that there might exist more than one subset s^* which satisfies this property. We only need assume that the function N returns any one of such subsets. The returned value of N is then used as the set $\Sigma(s_1)$ and is associated with the next state given by $\delta_1'(\tilde{s}_1, \mathbf{u})$.

We first claim that when a state $\tilde{s}_1$ is processed for a minterm $\mathbf{u} \in B^{|U|}$, the machine M_1' is equivalent to M_1 and the returned value of the function $N(\tilde{s}_1, \mathbf{u}, s_1)$ is well-defined, i.e. there exists a non-empty subset that satisfies the property stated in the definition of the function N. In the beginning, M_1' is equivalent to M_1 since we simply duplicate it. Also, since the behavior of M_1 is contained in T, there exists a subset $s^* \subseteq S$ such that $r \in s^*$ and $C(s^*, E(r_1)) = 1$ for the reset state r_1 of M_1', and thus the set $\Sigma(r_1)$ is well-defined. In general, when a state $\tilde{s}_1$ is processed for a minterm $\mathbf{u} \in B^{|U|}$, suppose that $\Sigma(\tilde{s}_1)$ has been defined and the machine M_1' is equivalent to M_1. Note that $C(\Sigma(\tilde{s}_1), E(\tilde{s}_1)) = 1$ since $\Sigma(\tilde{s}_1)$ is defined as the returned value of the function N. Now, suppose for the contrary that for any subset $s^* \subseteq S$ of the states of T

```
function feasible(M_1 = (U, V, S_1, λ_1, δ_1, r_1))
    /* let M'_1 = (U, V, S'_1, λ'_1, δ'_1, r_1) */
    M'_1 ← copy(M_1);
    for(each s_1 ∈ S'_1){ Σ(s_1) ← φ; }
    Σ(r_1) ← s* s.t. r ∈ s* and C(s*, E(r_1)) = 1;   /* r_1 ∈ S'_1 */
    mark r_1;
    while(there exists s̃_1 ∈ S'_1 that is marked){
        for(each u ∈ B^|U|){
            /* Let s_1 = δ'_1(s̃_1, u) */
            N ← N(s̃_1, u, s_1);
            if(∃ŝ_1 ∈ S'_1 : Σ(ŝ_1) = N and E(ŝ_1) = E(s_1))   δ'_1(s̃_1, u) ← ŝ_1;
            else if(Σ(s_1) = φ){   Σ(s_1) ← N;   mark s_1;   }
            else{   /* create a new state ŝ_1 */
                S'_1 ← S'_1 ∪ {ŝ_1};
                for(each ũ ∈ B^|U|){
                    δ'_1(ŝ_1, ũ) ← δ'_1(s_1, ũ);
                    λ'_1(ŝ_1, ũ) ← λ'_1(s_1, ũ);
                }
                δ'_1(s̃_1, u) ← ŝ_1;
                Σ(ŝ_1) ← N;
                E(ŝ_1) ← E(s_1);
                mark ŝ_1;
            }
        }
        remove the mark of s̃_1;
    }
    return M'_1;
```

Figure 5.3: Procedure for Generating a Feasible Machine

such that for all $\tilde{s} \in \Sigma(\tilde{s}_1)$, there exists $s \in s^*$ with $T(\tilde{s}, \mathbf{u}, \lambda'_1(\tilde{s}_1, \mathbf{u}), s) = 1$, $C(s^*, E(s_1)) = 0$. Since T is pseudo non-deterministic and since $C(\Sigma(\tilde{s}_1), E(\tilde{s}_1)) = 1$, for each state $\tilde{s} \in \Sigma(\tilde{s}_1)$, there exists a unique $s \in S$ such that $T(\tilde{s}, \mathbf{u}, \lambda'_1(\tilde{s}_1, \mathbf{u}), s) = 1$. Let $\mathcal{N}(\tilde{s}_1)$ be a subset of S given by $\mathcal{N}(\tilde{s}_1) = \{s \in S \mid \exists \tilde{s} \in \Sigma(\tilde{s}_1) : T(\tilde{s}, \mathbf{u}, \lambda'_1(\tilde{s}_1, \mathbf{u}), s) = 1\}$. Namely $\mathcal{N}(\tilde{s}_1)$ is the set of unique next states of the states of $\Sigma(\tilde{s}_1)$ under the transition $\mathbf{u}/\lambda'_1(\tilde{s}_1, \mathbf{u})$. Note that $\mathcal{N}(\tilde{s}_1)$ is not empty. Then $\mathcal{N}(\tilde{s}_1)$ satisfies the property that for all $\tilde{s} \in \Sigma(\tilde{s}_1)$, there exists $s \in s^*$ with $T(\tilde{s}, \mathbf{u}, \lambda'_1(\tilde{s}_1, \mathbf{u}), s) = 1$, where s^* is set to $\mathcal{N}(\tilde{s}_1)$, and also for all s^* with this property, $s^* \supseteq \mathcal{N}(\tilde{s}_1)$. Then by assumption, $C(\mathcal{N}(\tilde{s}_1), E(s_1)) = 0$, and thus there exists an input sequence σ_u and a state $s \in \mathcal{N}(\tilde{s}_1)$ such that the output sequence σ_v realized at the equivalence class $E(s_1)$ of M_1 for σ_u cannot be realized in T at s. However, since there exists a state $\tilde{s} \in \Sigma(\tilde{s}_1)$ for which s is the unique state of S such that $T(\tilde{s}, \mathbf{u}, \lambda'_1(\tilde{s}_1, \mathbf{u}), s) = 1$, the pair of sequences $(\mathbf{u}\sigma_u, \lambda'_1(\tilde{s}_1, \mathbf{u})\sigma_v)$, which is realized at $E(\tilde{s}_1)$ in M_1, cannot be realized in T at $\tilde{s}$. This conflicts with the fact that $C(\Sigma(\tilde{s}_1), E(\tilde{s}_1)) = 1$. Hence, there exists a subset $s^* \subseteq S$ such that for all $\tilde{s} \in \Sigma(\tilde{s}_1)$, there exists $s \in s^*$ with $T(\tilde{s}, \mathbf{u}, \lambda'_1(\tilde{s}_1, \mathbf{u}), s) = 1$ and $C(s^*, E(s_1)) = 1$, and therefore the returned value of the function N is well defined. We now show that the process for the state $\tilde{s}_1$ with the minterm $\mathbf{u}$ preserves the equivalency of M'_1 to M_1. It is true since if a new state $\hat{s}_1$ is created, the state is set to the new next state of $\tilde{s}_1$ for the input $\mathbf{u}$ and the transitions of $\hat{s}_1$ are set identical with those of s_1, the original next state.

Therefore, at the end of the procedure, we obtain a deterministic machine M'_1 equivalent to M_1. Also associated with each state s_1 of M'_1 is a subset $\Sigma(s_1) \subseteq S$ such that any pair of sequences (σ_u, σ_v) realized at $E(s_1)$ of M_1 can be realized in T at every state $s \in \Sigma(s_1)$. Thus M'_1 satisfies the condition (5.1)-(b) given in the definition of feasible machines. By definition of $\Sigma(r_1)$ given in the procedure shown in Figure 5.3, the condition (5.1)-(a) also holds. Therefore, M'_1 is a feasible machine. ■

By this theorem, we see that we can capture the complete set of contained behaviors by taking into account only feasible machines. However, as shown in the following example, the theorem does not hold for general non-deterministic machines. More specifically, there may exist behaviors contained in a non-deterministic machine that cannot be represented by feasible machines defined above[1].

Example 5.3.1 *Consider the non-deterministic machine T shown in Figure 5.4-(a). This machine is not pseudo non-deterministic. The behavior of a deterministic machine M_1 shown in Figure 5.4-(b) is contained in T. However, there is no feasible machine which represents the behavior.*

[1] In terms of the trace equivalence, this corresponds to the fact that trace equivalence does not coincide with simulation equivalence for non-deterministic finite automata [57].

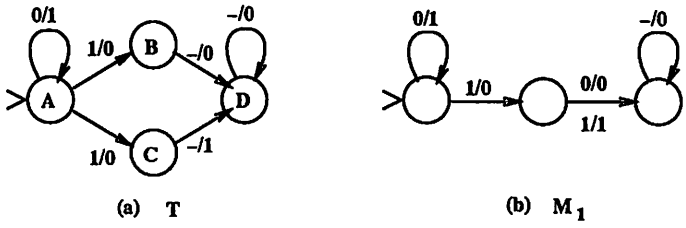

Figure 5.4: A Counterexample of Theorem 5.3.1 for General Non-Deterministic Machines

Due to the condition (5.1)-(b), for each state s_1 of a feasible machine M_1, the output function $\lambda_1|_{s_1}$ defined at s_1 in M_1 must be realized at every state $s \in \Sigma_1(s_1)$ in T. This requirement is not necessary for general non-deterministic machines. For the example above, even though no output function can be realized both at the states B and C in the machine T, the two states can be treated as a single state with arbitrary output functions since for any input sequence, if it can lead T to the state B, then it can also lead T to the state C with the same output sequence. For pseudo non-deterministic machines, we will see later that the condition (5.1)-(b) is effectively used to compute the feasible machines.

We now consider optimum (minimum-state) machines. By Theorem 5.3.1, we know that every behavior contained in a given pseudo non-deterministic machine T is represented by a feasible machine. However, the procedure used in the proof of this theorem, i.e. the one shown in Figure 5.3, may increase the number of states. Thus it is not obvious that there always exists a feasible machine whose number of states is minimum over all machines representing the behavior. In fact, we prove below that this statement is true, i.e. for any behavior contained in T, there exists a feasible machine that has the least number of states over all machines with the behavior. We use the following classical theorem to claim this statement.

Lemma 5.3.2 *For a completely specified deterministic finite state machine M, suppose that there is no equivalent pair of distinct states in M. Then the number of states of M is minimum over all machines representing the behavior given by M.*

Proof: See [27], [38], or [30]. ■

Lemma 5.3.3 *For any behavior $\mathcal{B}$ contained in a given pseudo non-deterministic finite state machine T, there exists a feasible machine whose number of states is minimum among all machines representing $\mathcal{B}$.*

Proof: Let $M_1 = (U, V, S_1, \lambda_1, \delta_1, r_1)$ be a feasible machine for the behavior $\mathcal{B}$ such that the number of states of M_1 is minimum among all the feasible machines representing the behavior. By Theorem 5.3.1, such M_1 always exists. We claim that the number of states of M_1 is minimum over all the machines representing the behavior $\mathcal{B}$, not just feasible machines. Since M_1 is a completely specified deterministic machine, Lemma 5.3.2 implies that the proof is done if we show that there is no equivalent pair of distinct states in M_1.

Suppose for the contrary that there exists an equivalence class E in M_1 which contains more than one state. Note first that every state of E is reachable from r_1 since otherwise we can find a feasible machine for the behavior with fewer states. Let $\Sigma(E)$ be the union of $\Sigma(s_1)$ over all the states s_1 of E. Consider a machine M_1' given by replacing each equivalence class E of M_1 by a single state and associate $\Sigma(E)$ with it. Specifically, for a pair of equivalence classes $(\tilde{E}, E)$, if we denote the states of M_1' corresponding to $\tilde{E}$ and E by $\tilde{s}_1'$ and s_1' respectively, then s_1' is the next state of $\tilde{s}_1'$ under an input $\mathbf{u}$ if and only if for each state $\tilde{s}_1 \in \tilde{E}$ of M_1, $\delta_1(\tilde{s}_1, \mathbf{u}) \in E$.

By definition of equivalence class, M_1' is equivalent to M_1. Note that the number of states of M_1' is strictly less than that of M_1, since there exists an equivalence class in M_1 which contains more than one state. We show that M_1' is also a feasible machine, which leads to a contradiction since M_1 has the minimum number of states over all the feasible machines for the behavior.

Since M_1 is a completely specified deterministic machine, there exists exactly one state r_1' in M_1' which corresponds to the equivalence class of the reset state r_1 of M_1. For this state r_1', the reset state of the original machine T is contained in the set of states of T associated with r_1', and thus the condition (5.1)-(a) holds.

Consider an arbitrary state $\tilde{s}_1'$ of M_1' and an arbitrary input minterm $\mathbf{u} \in B^{|U|}$. Let $\tilde{E}$ be the equivalence class of M_1 corresponding to $\tilde{s}_1'$. Let $\Sigma(\tilde{s}_1')$ be the set of states of T associated with $\tilde{s}_1'$. For every state $\tilde{s} \in \Sigma(\tilde{s}_1')$, there exists a state $\tilde{s}_1 \in \tilde{E}$ of M_1 for which $\tilde{s} \in \Sigma(\tilde{s}_1)$. Let s_1 be the next state of $\tilde{s}_1$ in M_1 under $\mathbf{u}$, i.e. $s_1 = \delta_1(\tilde{s}_1, \mathbf{u})$. Since M_1 is a feasible machine, there exists a state s of T contained in $\Sigma(s_1)$ such that $T(\tilde{s}, \mathbf{u}, \lambda_1(\tilde{s}_1, \mathbf{u}), s) = 1$. Denoting by E the equivalence class of M_1 which contains s_1, the state s_1' of M_1' corresponding to E is the next state of $\tilde{s}_1'$ under the input $\mathbf{u}$. The set of states of T associated with s_1' in M_1' is given by $\Sigma(E)$, and thus we see that $s \in \Sigma(E)$. Hence the condition (5.1)-(b) holds. Therefore, M_1' is a feasible machine. ∎

By this lemma, it immediately follows that a minimum-state behavior contained in a pseudo non-deterministic machine T is given by a feasible machine with the minimum number of states.

Theorem 5.3.2 *For a pseudo non-deterministic finite state machine T, let M_1 be a feasible machine with the minimum number of states. Then the behavior of M_1 is contained in T and its number of states is globally minimum over all the machines representing contained behaviors.*

5.4 Exact Methods

In this section, we present how to find exactly an optimum behavior for a given pseudo non-deterministic finite state machine. We first present in Section 5.4.1 an exact method for the state minimization of pseudo non-deterministic finite state machines, i.e. find a minimum-state behavior contained in a pseudo non-deterministic machine. We then discuss in Section 5.4.2 how the implementability of the resulting behavior can be taken into account, and present an exact method for the minimization of the E-machine, i.e. find a minimum-state permissible behavior. As we will see, both problems are formulated as a 0-1 integer linear programming problem. Namely, for a set of Boolean variables given as the input instance, we present a set of linear constraints on those variables so that a solution is given by an assignment for those Boolean variables with the minimum number of 1's which satisfies all the constraints. The solution space of each problem is defined by the Boolean space spanned by the set of Boolean variables given in the input instance. In Section 5.4.1 and Section 5.4.2, we present how to formulate the constraints for each problem. In Section 5.5, we show how to reduce the solution space, i.e. the number of Boolean variables given in the input instance, without affecting the optimality of the solution. Specifically, we introduce a notion of *compatible sets*, and show that optimum solutions are found by restricting the input instance so that a single Boolean variable is assigned to each compatible set. The notion of compatible sets are defined for pseudo non-deterministic machines, which is analogous to the notion of compatible sets introduced for deterministic machines [40].

5.4.1 Finding an Optimum Contained Behavior

The problem we address in this section is the state minimization of a pseudo non-deterministic machine T, i.e. find a behavior represented by a completely specified deterministic finite state machine with the minimum number of states over all the behaviors contained in T.

Let $C = \{s_1^*, \ldots, s_r^*\}$ be the set of subsets of states of a given pseudo non-deterministic machine T. For a moment, let us assume that C is the complete set of subsets of states, and thus the cardinality of C is $2^{|S|}$, where S is the state space of T. In Section 5.5, we show how to reduce the cardinality of C without affecting the optimality of the solution.

By Theorem 5.3.2, we see that an optimum contained behavior is given by finding a subset $\tilde{C}$ of C with the minimum cardinality for which a feasible machine can be composed so that each state of the feasible machine corresponds to an element of $\tilde{C}$. By definition of feasible machines, a subset $\tilde{C} \subseteq C$ can compose a feasible machine if and only if

(a) $\exists s_i^* \in \tilde{C} : r \in s_i^*$ and

(b) $\forall (s_i^*, \mathbf{u}) \in \tilde{C} \times B^{|U|} : \exists (s_j^*, \mathbf{v}) \in \tilde{C} \times B^{|V|}$ s.t. $\forall \tilde{s} \in s_i^* : \exists s \in s_j^* : T(\tilde{s}, \mathbf{u}, \mathbf{v}, s) = 1$.

By assigning a Boolean variable to each element of C, these conditions can be written in terms of a set of Boolean formulas. Suppose we associate a Boolean variable c_i for each $s_i^* \in C$. Then the first condition is given by $(\sum_{s_i^* : r \in s_i^*} c_i)$. In other words, we require that one must include in $\tilde{C}$ at least one element of C which contains the reset state r of the original machine T. For the second condition, we introduce, for each $s_i^* \in C$ and for each $\mathbf{u} \in B^{|U|}$, a Boolean formula $(c_i \Rightarrow \sum_{s_j^* \in n(s_i^*, \mathbf{u})} c_j)$, where $n(s_i^*, \mathbf{u})$ is the set of elements s_j^* of C such that there exists $\mathbf{v} \in B^{|V|}$ for which for all $\tilde{s} \in s_i^*$, $T(\tilde{s}, \mathbf{u}, \mathbf{v}, s) = 1$ for some $s \in s_j^*$. Then the problem is to find a minimum-weight assignment for the Boolean variables $\{c_1, \ldots, c_r\}$ which satisfies all the Boolean formulas, where the weight is equally assigned to every variable. This problem is a 0-1 integer linear programming problem.

Note the similarities of this formulation with the conventional approaches for the state minimization of deterministic machines [24]. The second condition above corresponds to the closure constraint, i.e. if an element c_i is chosen, then at least one element *implied* by c_i must also be chosen for each input $\mathbf{u}$. Implied elements, given in our case by $n(s_i^*, \mathbf{u})$, are those that can be treated as the next states of s_i^* in a feasible machine.

5.4.2 Finding an Optimum Permissible Behavior

The problem addressed in this section is the minimization of the E-machine, i.e. find a behavior represented by a completely specified deterministic finite state machine with the minimum number of states over all the implementable behaviors contained in a pseudo non-deterministic

machine T. The implementability is defined against an originally provided completely specified deterministic machine M_2 in Figure 5.1.

The solution of this problem is given by solving the state minimization problem addressed in the previous section with an additional constraint that the resulting feasible machine is implementable. Recall that a machine M_1 is said to be implementable if there exist implementations for M_1 and M_2 with which no combinational loop is created in the resulting implementation. As we showed in Section 4.7, the implementability is determined by checking the dependencies between the inputs and the outputs of M_1. It follows that once we obtain a feasible machine M_1, we need to see, for each state of M_1, the output function defined at the state so that we can identify which output variables V depend on which input variables U in the output function. This leads to a *dependency graph* G defined in Section 4.7, and by Theorem 4.7.1, we see that the machine is implementable if and only if G is acyclic.

Therefore, unlike the state minimization problem where the input instance is given by a set of subsets of states of T, we associate a function from $B^{|U|}$ to $B^{|V|}$ with each subset of states. Namely, an element c_i of the input instance is a pair (s^*, f), where $s^* \subseteq S$ is a subset of states of the original machine T and $f : B^{|U|} \rightarrow B^{|V|}$ is a function. The collection of all such pairs, $C = \{c_1, \ldots, c_r\}$, is the input instance of the problem. The idea is that one wants to compose a feasible machine using the elements of C so that each state of the feasible machine corresponds to the subset of states of an element $c_i \in C$ and the output function defined at the state in the feasible machine is given by the function given in c_i. Specifically, denoting $c_i^{(s^*)} = s^*$ and $c_i^{(f)} = f$ for each element $c_i = (s^*, f)$, we want to find a subset $\tilde{C}$ of C with the minimum cardinality which satisfies the following three conditions:

(a) $\exists c_i \in \tilde{C} : r \in c_i^{(s^*)}$ and

(b) $\forall (c_i, \mathbf{u}) \in \tilde{C} \times B^{|U|} : \exists c_j \in \tilde{C}$ s.t. $\forall \tilde{s} \in c_i^{(s^*)} : \exists s \in c_j^{(s^*)} : T(\tilde{s}, \mathbf{u}, c_i^{(f)}(\mathbf{u}), s) = 1$ and

(c) the dependency graph defined by the set of output functions of $\tilde{C}$ is acyclic.

Assigning a Boolean variable c_i for each element c_i of C, we write the conditions above by Boolean formulas. The conditions (a) and (b) can be handled in the same way as the previous section. For the condition (c), we specify for each subset $\hat{C}$ of C which creates a cycle in the corresponding dependency graph as $(\sum_{c_i \in \hat{C}} \overline{c}_i)$, which means that it is not allowed to choose all the elements of $\hat{C}$. Then the problem is to find a minimum-weight assignment for the Boolean variables

which satisfies all the Boolean formulas, where the weight is equally assigned to every variable. This problem is again a 0-1 integer linear programming problem.

5.4.3 Finding an Optimum Moore Behavior

In finding an optimum permissible behavior, we need to take into account the implementability of the resulting behavior. As shown in the previous section, the implementability is checked using a dependency graph. One compromise to avoid this additional difficulty on the implementability is to restrict attention to Moore behaviors only. Recall that for Moore behaviors, the outputs depend only on the internal states and not on the inputs [38]. Therefore, it is possible to implement a Moore behavior so that there is no combinational path from the inputs to the outputs. It follows that if M_1 is a Moore behavior, then there are no cycles in the corresponding dependency graph, i.e. it is always implementable no matter how M_2 is implemented. Hence, we don't need to use dependency graphs in setting constraints.

With this restriction, our problem is to find a behavior represented by a completely specified deterministic finite state machine with the minimum number of states over all the *Moore* behaviors contained in a given pseudo non-deterministic finite state machine T. By Theorem 5.3.2, we see that such a behavior is given by finding a feasible machine representing a Moore behavior with the minimum number of states over all the feasible Moore machines.

For Moore behaviors, an output function defined at each state is simply an output minterm, since the output pattern is unique for each state, independent of the inputs. Therefore, instead of output functions, we associate a minterm $\mathbf{v}$ of $B^{|V|}$ with a subset of states of T. Specifically, the input instance is the set $C = \{c_1, \ldots, c_r\}$, where an element c_i is a pair $(s^*, \mathbf{v})$ such that $s^* \subseteq S$ and $\mathbf{v} \in B^{|V|}$. As with the previous section, we may denote $c_i^{(s^*)} = s^*$ and $c_i^{(\mathbf{v})} = \mathbf{v}$. Our objectuve is to find a subset $\tilde{C}$ of C with the minimum cardinality which satisfies the following two conditions:

(a) $\exists c_i \in \tilde{C} : r \in c_i^{(s^*)}$ and

(b) $\forall (c_i, \mathbf{u}) \in \tilde{C} \times B^{|U|} : \exists c_j \in \tilde{C}$ s.t. $\forall \tilde{s} \in c_i^{(s^*)} : \exists s \in c_j^{(s^*)} : T(\tilde{s}, \mathbf{u}, c_i^{(\mathbf{v})}, s) = 1$.

We then assign a Boolean variable c_i for each element c_i of C, we write the conditions above by Boolean formulas. The problem is a 0-1 integer linear programming problem.

One might wonder why we need to associate an output minterm $\mathbf{v} \in B^{|V|}$ with each element of the input instance. In fact, it is possible to avoid it, while the linearity of the resulting problem cannot be maintained in this case. Specifically, let $C = \{s_1^*, \ldots, s_r^*\}$ be the set of subsets

of states of T. Then the solution of the problem is given by finding a subset $\tilde{C}$ of C with the minimum cardinality which satisfies the following two conditions:

(a) $\exists s_i^* \in \tilde{C} : r \in s_i^*$ and

(b) $\forall s_i^* \in \tilde{C} : \exists \mathbf{v} \in B^{|V|} : \forall \mathbf{u} \in B^{|U|} : \exists s_j^* \in \tilde{C}$ s.t. $\forall \tilde{s} \in s_i^* : \exists s \in s_j^* : T(\tilde{s}, \mathbf{u}, \mathbf{v}, s) = 1$.

Assigning a Boolean variable c_i for each element s_i^*, we can write these conditions as Boolean formulas. The first condition is same as the previous case. For the second condition, we introduce, for each $s_i^* \in C$, $(c_i \Rightarrow \sum_{\mathbf{v} \in B^{|V|}} \prod_{\mathbf{u} \in B^{|U|}} m(s_i^*, \mathbf{u}, \mathbf{v}))$. The notation $m(s_i^*, \mathbf{u}, \mathbf{v})$ designates the following:

$$m(s_i^*, \mathbf{u}, \mathbf{v}) = \begin{cases} \sum_{s_j^* \in n'(s_i^*, \mathbf{u}, \mathbf{v})} c_j & \text{if } n'(s_i^*, \mathbf{u}, \mathbf{v}) \neq \phi \\ 0 & \text{otherwise,} \end{cases}$$

where $n'(s_i^*, \mathbf{u}, \mathbf{v})$ is defined as the set of elements s_j^* of C such that for all $\tilde{s} \in s_i^*$, $T(\tilde{s}, \mathbf{u}, \mathbf{v}, s) = 1$ for some $s \in s_j^*$. We see that this formula contains in general a product of disjunctions of Boolean literals. Therefore, the formula is non-linear in general, and this formulation results in a 0-1 integer non-linear programming problem.

Note that the non-linearity arises from the intersection over all the input minterms $\mathbf{u} \in B^{|U|}$. The intersection is necessary in order to guarantee that the same output minterm $\mathbf{v} \in B^{|V|}$ can be used over all the inputs to associate with the subset s_i^*. It does not arise in our original formulation since we explicitly associate an output minterm with each subset of states in the input instance, so that we make a distinction between a pair $(s_i^*, \mathbf{v})$ and a pair $(s_i^*, \tilde{\mathbf{v}})$ for different output minterms.

5.4.4 A Summary of Exact Methods

We provide a brief summary of the three exact methods. Our original problem was the minimization of the E-machine computed for M_1 in Figure 5.1, i.e. find a behavior with the minimum number of states over all the implementable behaviors contained in a given pseudo non-deterministic finite state machine T.

The method given in Section 5.4.1 finds an optimum *contained* behavior in T. For this method, the input instance is the set of subsets of states of T, and the constraints are given by two sets of Boolean formulas. However, the method does not guarantee the implementability of

the resulting behavior. Therefore, the method can be used for our problem only if it is somehow guaranteed in advance that every contained behavior is implementable. One such case is when the machine M_2 is a Moore machine, and it is known that no combinational loop is created for an arbitrary M_1.

If there is no such a guarantee on the implementability, we need to use the method given in Section 5.4.2, which finds an optimum *permissible* behavior. For this method, however, we need to associate an output function with each subset of states of T. Furthermore, we need to take into account an additional constraint on the implementability using a dependency graph. In order to avoid the complication of dependency graphs, Section 5.4.3 presented an exact method for finding an optimum *Moore* behavior contained in T. Since the behavior is a Moore behavior, it is always implementable, and thus we don't need to check the implementability using dependency graphs in setting the constraints. For this method, instead of an output function, we associate an output minterm with each subset. For all three cases, the problem can be described as a 0-1 integer linear programming problem.

5.5 Compatible Sets

5.5.1 Compatible Sets

In the exact methods presented in the previous section, we assume that the input instance is the set of subsets of states of a given pseudo non-deterministic machine T. For the exact method of finding an optimum permissible behavior, we further associate an output function for each subset. Thus, the number of elements given in the input instance is exponential in the number of states of T. In this section, we consider how to reduce the number of elements of the input instance so that an optimum solution is still found by using the same constraints given for each of the three methods of Section 5.4 over those reduced elements.

For each exact method, our objective is to find a feasible machine with some property. Recall that each state of a feasible machine corresponds to a subset of states of T. Therefore, for a subset s^* of states of T, if there is no feasible machine in which there exists a state corresponding to s^*, s^* need not to be included as an element of the input instance since the element is never included to compose a feasible machine. Similarly, a pair (s^*, f), where $s^* \subseteq S$ and $f : B^{|U|} \rightarrow B^{|V|}$, need not be included if there is no feasible machine in which there exists a state corresponding to s^* with the output function f. These elements can then be removed from the input instance without

affecting the optimality of the solution.

Then the question is what is the set of pairs (s^*, f) that can be used to compose feasible machines. More specifically, a pair (s^*, f) is in the set if and only if there exists a feasible machine in which there exists a state such that the corresponding set of states of T is s^* and the output function defined at the state in the feasible machine is equal to f.

A superset of the set defined above is the one called *output-consistent* set. A pair (s^*, f) is said to be *output-consistent* if the following condition holds:

$$\forall(\tilde{s}, \mathbf{u}) \in s^* \times B^{|U|} : \exists s \in S : T(\tilde{s}, \mathbf{u}, f(\mathbf{u}), s) = 1,$$

where S is the set of states of the pseudo non-deterministic machine T. Namely, the condition requires that it is possible to realize the same output function f at every state of s^* in T. If a pair (s^*, f) is not output-consistent, then no feasible machine has a state corresponding to that pair. This is because the condition (5.1)-(b) requires that the same output function must be realized at every state of T associated with a single state of a feasible machine. However, note that the output-consistency is simply a necessary condition, and does not imply the existence of a feasible machine with a state corresponding to the pair.

We now define the set of compatible sets as follows:

Definition: Compatible Sets

Given a subset $s^* \subseteq S$ and a function $f : B^{|U|} \rightarrow B^{|V|}$, (s^*, f) is a **compatible set** if there exists a behavior $\mathcal{B}$ between U and V satisfying the following condition:

Containedness:

For all $(\sigma_u, \sigma_v) \in \mathcal{B}$ and for all $s \in s^*$, (σ_u, σ_v) can be realized in T at state s and $\mathbf{v}_0 = f(\mathbf{u}_0)$, where $\mathbf{u}_0$ and $\mathbf{v}_0$ are the first elements of σ_u and σ_v respectively.

The central idea here is that for a compatible set (s^*, f), there exists a behavior that can be realized at every state of s^* in T with the output function f. More specifically, for each $s \in s^*$, there must exist an edge labeled $\mathbf{u}_0/\mathbf{v}_0$ for each pair (σ_u, σ_v) of the behavior $\mathcal{B}$, where $\mathbf{u}_0/\mathbf{v}_0$ denotes the first input/output pair of (σ_u, σ_v) and $\mathbf{v}_0 = f(\mathbf{u}_0)$. Further, we have the condition that the edge can be continued to produce the given sequence (σ_u, σ_v).

It is then claimed that the set of compatible sets is precisely the set of pairs (s^*, f) that can be used to compose a feasible machine.

Theorem 5.5.1 *For a given pseudo non-deterministic finite state machine T, suppose that every state of T is reachable from the reset state. Then (s^*, f) is a compatible set if and only if there exists a feasible machine in which there exists a state corresponding to (s^*, f).*

Proof: Suppose that there exists a feasible machine M_1 for T in which there exists a state s_1 such that the corresponding subset of states of T is s^*, i.e. $s^* = \Sigma(s_1)$, and the output function defined at s_1 is f. Let $\mathcal{B} = \{(\sigma_u, \sigma_v) \mid (\sigma_u, \sigma_v)$ is realized in M_1 at $s_1.\}$. Then by Lemma 5.3.1, $\mathcal{B}$ is a behavior between U and V such that for all $(\sigma_u, \sigma_v) \in \mathcal{B}$ and for all $s \in \Sigma(s_1)$, (σ_u, σ_v) is realized at s in T. Since the output function defined at s_1 in T is f, the containedness condition above holds. Hence (s^*, f) is a compatible set.

Conversely, suppose (s^*, f) is a compatible set. We show that there exists a feasible machine with a state corresponding to (s^*, f). Let $s \in s^*$ be an arbitrary state of s^*. Consider a pseudo non-deterministic machine T', identical to T except that s is the reset state of T'. Let $\mathcal{B}'$ be a behavior satisfying the containedness condition for (s^*, f) in the definition of compatible sets. Then $\mathcal{B}'$ is contained in T'. Therefore, there exists a feasible machine M_1' for T' which represents $\mathcal{B}'$. Then we can associate s^* with the reset state r_1' of M_1' without violating the conditions (5.1) given in the definition of feasible machines. Since the output function defined at r_1 is f, (s^*, f) corresponds to r_1'.

Now, since s is reachable from the reset state r of T, there exists a pair of input and output sequences $(\sigma_u^{(s)}, \sigma_v^{(s)})$ such that $\sigma_u^{(s)}$ leads T to s with the output sequence $\sigma_v^{(s)}$. Also, there exists a behavior $\mathcal{B}$ such that (1) $\mathcal{B}$ is contained in T, and (2) for all $(\sigma_u, \sigma_v) \in \mathcal{B}$ such that $(\sigma_u^{(s)}, \sigma_v^{(s)})$ is a pair of prefix subsequences of (σ_u, σ_v), i.e. there exists a pair of sequences (σ_u', σ_v') with $(\sigma_u, \sigma_v) = (\sigma_u^{(s)}\sigma_u', \sigma_v^{(s)}\sigma_v')$, the remaining pair of sequences is a member of $\mathcal{B}'$, i.e. $(\sigma_u', \sigma_v') \in \mathcal{B}'$. Let M_1 be a feasible machine for T which represents $\mathcal{B}$. Let s_1 be the state of M_1 such that $\sigma_u^{(s)}$ leads M_1 to s_1. Let $\tilde{s}_1$ be the preceding state of s_1 in M_1, i.e. M_1 moves from $\tilde{s}_1$ to s_1 when the last element of $\sigma_u^{(s)}$ is applied. Denote the last element of $\sigma_u^{(s)}$ by u. Suppose we change the transition from $\tilde{s}_1$ to s_1 so that M_1 moves from $\tilde{s}_1$ to r_1', the reset state of M_1', under the input u with the same output. Since M_1' is a feasible machine for T' representing the behavior $\mathcal{B}'$ and since T' is identical with T except for the reset state, the resulting machine is a feasible machine for T representing $\mathcal{B}$, in which the state r_1' corresponds to (s^*, f). This completes the proof. ■

Hence, we see that it is sufficient to use only the compatible sets for the input instance in the exact method for finding an optimum permissible behavior given in Section 5.4.2. For the exact method of finding an optimum Moore behavior given in Section 5.4.3, since we restrict attention

to Moore behaviors only, the input instance is given by the set of compatible sets (s^*, f) such that $f(\mathbf{u})$ is identical over all input minterms $\mathbf{u}$. Thus we may denote such a compatible set by $(s^*, \mathbf{v})$, where $\mathbf{v} \in B^{|V|}$ is the output value of the associated function.

For the exact method of finding an optimum contained behavior given in Section 5.4.1, we are not concerned with implementability. Thus we don't need to associate an output function with a subset of states. This was first indicated by Damiani in [14]. In fact, by Theorem 5.3.1, the set of contained behaviors is precisely given by the set of feasible machines, and therefore it is sufficient to use as the input instance of this problem the set of subsets of states s^* with the property that there exists a function $f : B^{|U|} \rightarrow B^{|V|}$ such that (s^*, f) is a compatible set.

5.5.2 Computing Compatible Sets

In this section, we show how to compute the set of compatible sets. Consider the following iterative computation of a function $R^{(k)} : 2^S \times \mathcal{F} \rightarrow B$, where 2^S designates the power set of S and $\mathcal{F}$ is the set of functions from $B^{|U|}$ to $B^{|V|}$.

$$R^{(1)}(\tilde{s}^*, \tilde{f}) = 1 \quad \Leftrightarrow \quad \forall(\tilde{s}, \mathbf{u}) \in \tilde{s}^* \times B^{|U|} : \exists s \in S : T(\tilde{s}, \mathbf{u}, \tilde{f}(\mathbf{u}), s) = 1,$$

$$R^{(k)}(\tilde{s}^*, \tilde{f}) = 1 \quad \Leftrightarrow \quad \forall \mathbf{u} \in B^{|U|} : \exists(s^*, f) \in 2^S \times \mathcal{F} : R^{(k-1)}(s^*, f) = 1 \text{ and}$$
$$\forall \tilde{s} \in \tilde{s}^* : \exists s \in s^* : T(\tilde{s}, \mathbf{u}, \tilde{f}(\mathbf{u}), s) = 1$$

Intuitively, $R^{(k)}$ is the characteristic function of a set of pairs $(\tilde{s}^*, \tilde{f})$, where $\tilde{s}^*$ is a subset of states of a given pseudo non-deterministic machine T and $\tilde{f} : B^{|U|} \rightarrow B^{|V|}$ is an output function, with the property that for any input sequence σ_u with the length no greater than k, there exists an output sequence σ_v with the same length such that (σ_u, σ_v) can be realized in T at every state $\tilde{s} \in \tilde{s}^*$ and $\mathbf{v}_0 = \tilde{f}(\mathbf{u}_0)$, where $\mathbf{u}_0$ and $\mathbf{v}_0$ are the first elements of σ_u and σ_v respectively. Note the analogy between this statement and the containedness condition given in the definition of compatible sets in the previous section. More specifically, if the integer k is infinitely large, then the collection of such pairs of sequences (σ_u, σ_v) leads to a behavior between U and V which satisfies the containedness condition. Note also that for each state $\tilde{s} \in S$, there exists a function $\tilde{f}$ such that $R^{(k)}(\{\tilde{s}\}, \tilde{f}) = 1$ for all $k \geq 1$.

Let $R : 2^S \times \mathcal{F} \rightarrow B$ be the function obtained at the fixed point of the computation above. Namely, for an integer $K > 1$ such that $R^{(K)} = R^{(K-1)}$, we set R equal to $R^{(K)}$. Note that such K always exists since for $s^* \subseteq S$ and $f : B^{|U|} \rightarrow B^{|V|}$, if $R^{(k-1)}(s^*, f) = 0$, then $R^{(k)}(s^*, f) = 0$ for all $k > 1$, and thus the number of elements (s^*, f) contained in $R^{(k)}$ does not increase as k increases, while the total number of such elements is finite.

We claim that the function R is the characteristic function of the compatible sets for T.

Theorem 5.5.2 *For $s^* \subseteq S$ and $f : B^{|U|} \rightarrow B^{|V|}$, (s^*, f) is a compatible set if and only if $R(s^*, f) = 1$.*

Proof: First, for the trivial case where s^* is empty, the theorem holds since (s^*, f) is always a compatible set and $R^{(k)}(s^*, f) = 1$ for all $k \geq 1$.

Consider the case where s^* is not empty. We show by induction on $k \geq 1$ that $R^{(k)}(\tilde{s}^*, \tilde{f}) = 1$ if and only if there exists a set of pairs, $\mathcal{B}^{(k)} \subseteq \{(\sigma_u, \sigma_v) \mid |\sigma_u| = |\sigma_v| \leq k\}$, such that (1) $\mathcal{B}^{(k)}$ satisfies the completeness and the prefix conditions, (2) for all $(\sigma_u, \sigma_v) \in \mathcal{B}^{(k)}$, $\mathbf{v}_0 = \tilde{f}(\mathbf{u}_0)$, where $\mathbf{u}_0$ and $\mathbf{v}_0$ are the first elements of σ_u and σ_v respectively, and (3) for all $(\sigma_u, \sigma_v) \in \mathcal{B}^{(k)}$ and for all $\tilde{s} \in \tilde{s}^*$, (σ_u, σ_v) can be realized in T at the state $\tilde{s}$. Then the theorem directly follows. The completeness and the prefix conditions are defined in Section 4.2 where a definition of a behavior is given.

The statement is true for $k = 1$ by construction. Suppose it is true for $k - 1$. Consider a subset $\tilde{s}^* \subseteq S$ and a function $\tilde{f} : B^{|U|} \rightarrow B^{|V|}$ such that $R^{(k)}(\tilde{s}^*, \tilde{f}) = 1$. We show that there exists a set of pairs of sequences $\mathcal{B}^{(k)}$ with the property above. Since $R^{(k)}(\tilde{s}^*, \tilde{f}) = 1$, for each $\mathbf{u} \in B^{|U|}$, there exists (s^*, f) such that $R^{(k-1)}(s^*, f) = 1$ and for all $\tilde{s} \in \tilde{s}^*$, there exists $s \in s^*$ for which $T(\tilde{s}, \mathbf{u}, \tilde{f}(\mathbf{u}), s) = 1$. By the induction hypothesis, there exists a set $\mathcal{B}^{(k-1)} \subseteq \{(\sigma_u, \sigma_v) \mid |\sigma_u| = |\sigma_v| \leq k-1\}$ associated with such (s^*, f) which satisfies the property above. Define a function χ such that $\chi(\mathbf{u})$ returns one such set of pairs of sequences $\mathcal{B}^{(k-1)}$. Although there might exist more than one such (s^*, f) for a given $\mathbf{u} \in B^{|U|}$ and $(\tilde{s}^*, \tilde{f})$, and for these, possibly more than one such set $\mathcal{B}^{(k-1)}$, we choose one particular set $\mathcal{B}^{(k-1)}$ as the return value of $\chi(\mathbf{u})$. Thus for a given $\mathbf{u} \in B^{|U|}$, $\chi(\mathbf{u})$ uniquely defines the pair (s^*, f) as well as the associated set $\mathcal{B}^{(k-1)}$. Consider an arbitrary input minterm $\mathbf{u}_0 \in B^{|U|}$. Let $\mathcal{B}^{(k-1)} = \chi(\mathbf{u}_0)$. For each $(\sigma'_u, \sigma'_v) \in \mathcal{B}^{(k-1)}$, consider the pair of sequences $(\mathbf{u}_0\sigma'_u, \tilde{f}(\mathbf{u}_0)\sigma'_v)$, and include this pair in the set $\mathcal{B}^{(k)}$. Consider the set $\mathcal{B}^{(k)}$ obtained this way by processing all $\mathbf{u}_0$. Also add the pair of null sequences to $\mathcal{B}^{(k)}$. Then by construction, any pair $(\sigma_u, \sigma_v) \in \mathcal{B}^{(k)}$ can be realized in T at any state of $\tilde{s}^*$ with $\mathbf{v}_0 = \tilde{f}(\mathbf{u}_0)$. Also for this $\mathcal{B}^{(k)}$, the completeness condition holds since we processed all $\mathbf{u}_0$ and $\mathcal{B}^{(k-1)}$ is complete. Furthermore, $\mathcal{B}^{(k)}$ satisfies the prefix condition since $\mathcal{B}^{(k-1)}$ satisfies the prefix condition. Hence for a pair $(\tilde{s}^*, \tilde{f})$ such that $R^{(k)}(\tilde{s}^*, \tilde{f}) = 1$, there exists a set $\mathcal{B}^{(k)}$ with the property above.

Conversely, consider a pair $(\tilde{s}^*, \tilde{f})$ for which there exists a set $\mathcal{B}^{(k)} \subseteq \{(\sigma_u, \sigma_v) \mid |\sigma_u| = |\sigma_v| \leq k\}$ with the property above. We show $R^{(k)}(\tilde{s}^*, \tilde{f}) = 1$. Let $\mathbf{u}_0 \in B^{|U|}$ be an arbitrary input

minterm. Since T is pseudo non-deterministic, for each $\tilde{s} \in \tilde{S}^*$, there exists a unique $s \in S$ such that $T(\tilde{s}, \mathbf{u}_0, \tilde{f}(\mathbf{u}_0), s) = 1$. Let s^* be a subset of the states of T given by $s^* = \{s \in S \mid \exists \tilde{s} \in \tilde{S}^* : T(\tilde{s}, \mathbf{u}_0, \tilde{f}(\mathbf{u}_0), s) = 1\}$. Namely, s^* is the set of unique next states of the states of $\tilde{s}^*$ under the transition $\mathbf{u}_0/\tilde{f}(\mathbf{u}_0)$. Note that s^* is not empty. Note also that s^* has the property that for all $\tilde{s} \in \tilde{s}^*$, there exists $s \in s^*$ such that $T(\tilde{s}, \mathbf{u}_0, \tilde{f}(\mathbf{u}_0), s) = 1$. Let $f : B^{|U|} \rightarrow B^{|V|}$ be a function such that $f(\mathbf{u}_1) = \mathbf{v}_1$ if and only if $(\mathbf{u}_0\mathbf{u}_1, \tilde{f}(\mathbf{u}_0)\mathbf{v}_1) \in \mathcal{B}^{(k)}$. By the completeness condition, such $\mathbf{v}_1$ and hence f is uniquely defined. We claim that $R^{(k-1)}(s^*, f) = 1$. Consider a set $\mathcal{B}^{(k-1)}$ of pairs of sequences with the length less than k defined as follows. For a given sequence σ_u of $B^{|U|}$ with $|\sigma_u| \leq k-1$, let σ_v be the sequence of $B^{|V|}$ such that $(\mathbf{u}_0\sigma_u, \tilde{f}(\mathbf{u}_0)\sigma_v) \in \mathcal{B}^{(k)}$. We include the pair (σ_u, σ_v) in $\mathcal{B}^{(k-1)}$. For the set $\mathcal{B}^{(k-1)}$ defined in this way for all sequences σ_u with $|\sigma_u| \leq k-1$, since $\mathcal{B}^{(k)}$ satisfies the completeness and the prefix conditions, so does $\mathcal{B}^{(k-1)}$. We claim that for all $(\sigma_u, \sigma_v) \in \mathcal{B}^{(k-1)}$ and for all $s \in s^*$, (σ_u, σ_v) can be realized in T at the state s. Suppose for the contrary that there exists a state $s \in s^*$ at which (σ_u, σ_v) cannot be realized in T. By definition of s^*, there exists $\tilde{s} \in \tilde{S}^*$ for which s is the unique next state in S such that $T(\tilde{s}, \mathbf{u}, \tilde{f}(\mathbf{u}_0), s) = 1$. It follows that the pair of sequences $(\mathbf{u}_0\sigma_u, \tilde{f}(\mathbf{u}_0)\sigma_v)$ cannot be realized in T at $\tilde{s}$, which conflicts with the fact that $(\mathbf{u}_0\sigma_u, \tilde{f}(\mathbf{u}_0)\sigma_v)$ is in $\mathcal{B}^{(k)}$. Thus the set $\mathcal{B}^{(k-1)}$ satisfies the property. Also by construction, if we denote by $\mathbf{u}_1$ and $\mathbf{v}_1$ the first element of σ_u and σ_v respectively, then $\mathbf{v}_1 = f(\mathbf{u}_1)$. Therefore, the induction hypothesis implies that $R^{(k-1)}(s^*, f) = 1$. Hence, $R^{(k)}(\tilde{s}^*, \tilde{f}) = 1$. ■

As noted in the previous section, in case our focus is only on Moore behaviors, we need only to compute the set of compatible sets (s^*, f) such that the output value of f is invariant with the input values. The characteristic function of such a set can be given using the computation above, where a function $\tilde{f}$ is replaced by an output minterm $\mathbf{v} \in B^{|V|}$.

Similarly, if we want to find an optimum behavior contained in T, then it is sufficient to compute the set of subsets $s^* \subseteq S$ with the property that there exists a function f such that (s^*, f) is a compatible set. Namely, denoting the characteristic function of such a set by $R_s : 2^S \rightarrow B$, we see that $R_s(s^*) = 1$ if and only if there exists $f : B^{|U|} \rightarrow B^{|V|}$ such that $R(s^*, f) = 1$, where R is the characteristic function of the compatible sets computed above. Therefore, the characteristic function R_s of the set of subsets used for the input instance of the exact method for finding an optimum contained behavior can be easily given using the original characteristic function R.

Note that such a characteristic function R_s can be computed without using R. Specifically, as an analogy to the computation of R, R_s is given by the fixed point of the following iterative

computation:

$$R_s^{(1)}(\tilde{s}^*) = 1 \quad \Leftrightarrow \quad \forall \mathbf{u} \in B^{|U|} : \exists \mathbf{v} \in B^{|V|} : \forall \tilde{s} \in \tilde{s}^* : \exists s \in S : T(\tilde{s}, \mathbf{u}, \mathbf{v}, s) = 1,$$
$$R_s^{(k)}(\tilde{s}^*) = 1 \quad \Leftrightarrow \quad \forall \mathbf{u} \in B^{|U|} : \exists (s^*, \mathbf{v}) \in 2^S \times B^{|V|} : \quad R_s^{(k-1)}(s^*) = 1 \text{ and}$$
$$\forall \tilde{s} \in \tilde{s}^* : \exists s \in s^* : T(\tilde{s}, \mathbf{u}, \mathbf{v}, s) = 1$$

Intuitively, $R_s^{(k)}(\tilde{s}^*) = 1$ if and only if for any input sequence σ_u whose length is no greater than k, there exists an output sequence σ_v such that (σ_u, σ_v) can be realized in T at every state of $\tilde{s}^*$. Therefore, at the fixed point of the computation above, denoting by R_s the resulting function, we see that $R_s(\tilde{s}^*) = 1$ if and only if there exists a behavior $\mathcal{B}$ between the inputs U and the outputs V such that every pair $(\sigma_u, \sigma_v) \in \mathcal{B}$ can be realized in T at every state of $\tilde{s}^*$. Hence, if there exists a function $\tilde{f}$ such that $(\tilde{s}^*, \tilde{f})$ is a compatible set, then $R_s(\tilde{s}^*) = 1$. Conversely, if $R_s(\tilde{s}^*) = 1$, then defining a function $\tilde{f} : B^{|U|} \rightarrow B^{|V|}$ so that $(\mathbf{u}, \tilde{f}(\mathbf{u}))$ is a member of the corresponding behavior $\mathcal{B}$ for all $\mathbf{u}$, we see that $(\tilde{s}^*, \tilde{f})$ is a compatible set. Thus the computation above provides the characteristic function R_s of the set of subsets of states that can be used in a feasible machine for T.

We close this section by describing how R can be recovered from R_s, i.e. how one can compute the characteristic function R of the compatible sets if R_s is given. Specifically, we show that for a given $s^* \subseteq S$ and a function $f : B^{|U|} \rightarrow B^{|V|}$, $R(s^*, f) = 1$ if and only if

$$\forall \mathbf{u} \in B^{|U|} : \exists \hat{s}^* \subseteq S : \quad R_s(\hat{s}^*) = 1 \text{ and} \quad \forall s \in s^* : \exists \hat{s} \in \hat{s}^* : T(s, \mathbf{u}, f(\mathbf{u}), \hat{s}) = 1. \tag{5.2}$$

Theorem 5.5.3 *For a given $s^* \subseteq S$ and a function $f : B^{|U|} \rightarrow B^{|V|}$, $R(s^*, f) = 1$ if and only if (s^*, f) satisfies the condition (5.2).*

Proof: It is certainly true that $R(s^*, f) = 1$ implies the condition (5.2), since the condition is identical with the computation of the function $R^{(k)}$ given in the beginning of this section.

Conversely, suppose that (s^*, f) satisfies the condition (5.2). For a given input minterm $\mathbf{u} \in B^{|U|}$, consider $\hat{s}^* \subseteq S$ given in the condition above. We refer to this $\hat{s}^*$ as the next-state subset corresponding to $\mathbf{u}$. Since $R_s(\hat{s}^*) = 1$, by definition, there exists a function $\hat{f}$ for which $(\hat{s}^*, \hat{f})$ is a compatible set. Let $\hat{\mathcal{B}}$ be a behavior between U and V satisfying the containedness condition for this pair $(\hat{s}^*, \hat{f})$ as stated in the definition of compatible sets. For each pair of input and output sequences $(\hat{\sigma}_u, \hat{\sigma}_v) \in \hat{\mathcal{B}}$, consider a pair of sequences $(\mathbf{u}\hat{\sigma}_u, f(\mathbf{u})\hat{\sigma}_v)$, and include this in a set $\mathcal{B}$. Repeat this process over all the input minterms $\mathbf{u} \in B^{|U|}$, and consider the resulting set of pairs of

sequences $\mathcal{B}$. The set $\mathcal{B}$ satisfies the prefix and completeness conditions since we processed all the input minterms and the set $\hat{\mathcal{B}}$ defined for each input minterm $\mathbf{u} \in B^{|U|}$ satisfies both conditions. Now, for each pair of input and output sequences $(\sigma_u, \sigma_v) \in \mathcal{B}$, denote $\sigma_u = \mathbf{u}\hat{\sigma}_u$ and $\sigma_v = f(\mathbf{u})\hat{\sigma}_v$, where $\mathbf{u}$ is the first element of σ_u. Then for the next-state subset $\hat{s}^*$ corresponding to $\mathbf{u}$, which was used when the set $\mathcal{B}$ was constructed, it is possible to move in the machine T from any of the states of s^* to some of the states of $\hat{s}^*$ under the transition $\mathbf{u}/f(\mathbf{u})$. Furthermore, since the pair $(\hat{\sigma}_u, \hat{\sigma}_v)$ is a member of a behavior $\hat{\mathcal{B}}$ satisfying the containedness condition at $\hat{s}^*$, $(\hat{\sigma}_u, \hat{\sigma}_v)$ can be realized in T at any state of $\hat{s}^*$. Therefore (σ_u, σ_v) can be realized in T at any state of s^*. This statement is true for all the input minterms $\mathbf{u}$, and thus the set $\mathcal{B}$ is a behavior between U and V that satisfies the containedness condition for (s^*, f). Hence (s^*, f) is a compatible set and $R(s^*, f) = 1$. ■

5.6 A Heuristic Method

We present a heuristic method for the state minimization problem of pseudo non-deterministic machines. The problem is to find a behavior with the minimum number of states over all the behaviors contained in a pseudo non-deterministic machine. In order to apply the heuristic for the minimization of the E-machine, we need to guarantee the implementability of the resulting behavior. We make this guarantee by restricting to Moore behaviors. This restriction can be made by a trivial modification (discussed later) of the algorithms employed in the heuristic. In this section, we describe technical details of the algorithms used in the heuristic, starting with an overview.

5.6.1 Irredundant Compatible Sets

Let $T = (U, V, S, T, r)$ be a given pseudo non-deterministic finite state machine. The proposed procedure keeps track of a set of subsets of states of T, and tries to decrease the cardinality of the set while maintaining the invariance that the set of subsets can compose a feasible machine. Although a compatible set is defined as a subset of states along with an output function, the procedure keeps only a set of subsets. This is because the cost function is the number of subsets, and we do not care which output function is associated with each subset, just that one exists.

Definition: Closed Set

A set $C = \{s_1^*, \ldots, s_n^*\}$, where $s_i^* \subseteq S$, is **closed** if for all $s_i^* \in C$, there exists a function

$f : B^{|U|} \rightarrow B^{|V|}$ such that

$$\forall \mathbf{u} \in B^{|U|} : \exists s_j^* \in C : \tau(s_i^*, \mathbf{u}, f(\mathbf{u}), s_j^*) = 1,$$

where $\tau(s_i^*, \mathbf{u}, f(\mathbf{u}), s_j^*) = 1$ if and only if for all $s_i \in s_i^*$, there exists $s_j \in s_j^*$ such that $T(s_i, \mathbf{u}, f(\mathbf{u}), s_j) = 1$.

Intuitively, C is closed if for each element $s_i^* \in C$, there exists an element that can be treated as the next state of s_i^* for each input. We say C is *feasible* if it is closed and there exists $s_i^* \in C$ that contains the reset state r of T. Namely, C is feasible if a feasible machine can be composed using the elements of C. By Theorem 5.5.1, each state of a feasible machine corresponds to a compatible set defined in Section 5.5. Specifically, for each element s_i^* of a feasible set C, (s_i^*, f) is a compatible set, where f is the function given in the definition of closed sets above.

One might wonder if the closedness can be defined without explicitly associating a function f as above. Specifically, C is *closed* if for all $s_i^* \in C$,

$$\forall \mathbf{u} \in B^{|U|} : \exists (s_j^*, \mathbf{v}) \in C \times B^{|V|} : \tau(s_i^*, \mathbf{u}, \mathbf{v}, s_j^*) = 1.$$

In fact, this is an equivalent definition. However, we use the former since it makes it easier to restrict our attention to Moore behaviors, as discussed later.

Among feasible sets C, we are interested in those with the property of irredundancy.

Definition: Redundant Sets

Given a feasible set C, an element $s^* \in C$ is **redundant** if $C - \{s^*\}$ is also feasible. Otherwise, s^* is **irredundant.**

We say that a feasible set C is *redundant* if there exists $\tilde{C} \subset C$ such that $\tilde{C}$ is also feasible. Otherwise C is *irredundant.* Note that C might be redundant even if every single $s^* \in C$ is irredundant. Irredundancy of C is a necessary condition for optimum solutions of the problem. The proposed heuristic procedure tries to introduce redundancy into a given set C by replacing each element of C with another, and then make the resulting set irredundant, so that the the cardinality decreases. The subprocedure which makes C irredundant, called IRREDUNDANT, is described in Section 5.6.5.

To introduce redundancy into C, suppose $s_j^* \in C$ is irredundant, i.e. $C - \{s_j^*\}$ is not feasible. One reason for this is that s_j^* is the unique element of C containing the reset state r. In case s_j^* is not a unique such element, the reason for the irredundancy is that there exists another

element $s_i^* \in C$, $s_i^* \neq s_j^*$, which *needs* s_j^*. More specifically, there exists s_i^* such that for each function $f : B^{|U|} \rightarrow B^{|V|}$, either

1. (s_i^*, f) is not a compatible set, or
2. there exists $\mathbf{u} \in B^{|U|}$ for which no element in $C - \{s_j^*\}$ can be treated as the next state of s_i^*, i.e. $\forall s^* \in C - \{s_j^*\} : \tau(s_i^*, \mathbf{u}, f(\mathbf{u}), s^*) = 0$.

Therefore, we need procedures which decrease these possibilities, in order to introduce redundancy into C. We use two procedures for this purpose, described in the following section.

5.6.2 Overview

In designing a heuristic procedure, we made a decision that the procedure always maintains the feasibility of a set C and never increases the cardinality of C. The procedure consists of three subprocedures. Besides IRREDUNDANT, the other two, called REDUCE and EXPAND respectively, are used to introduce redundancy into C.

REDUCE replaces each element of C by another so as to increase the number of output functions f that can be associated with it as compatible sets. This procedure is concerned with the first case of "*needs*" above. EXPAND replaces each element of C so as to increase the number of elements s_j^* in C that are not unique next states of any other element of C. This procedure contributes to the second case. Specifically, REDUCE is based on the observation that if (s^*, f) is a compatible set, then any subset of s^* is also compatible with f. We replace a given element s^* by its smallest subset such that the replacement of s^* by the subset maintains the feasibility of the resulting set. This increases possibly the number of functions that can be associated with this subset. EXPAND, on the other hand, replaces an element s^* by another element $\hat{s}^*$ so that the replacement of s^* by $\hat{s}^*$ causes a maximal number of elements of C to be redundant. For efficiency, we restrict $\hat{s}^*$ to contain s^*. Since both REDUCE and EXPAND can make C redundant, we invoke IRREDUNDANT whenever REDUCE or EXPAND is applied.

In summary, REDUCE increases the number of functions associated with each element, while EXPAND increases the number of possible next states. The proposed procedure takes as input the transition relation of a pseudo non-deterministic finite state machine T. After a feasible set C is found as an initial set, the procedure iteratively applies EXPAND and REDUCE, invoking IRREDUNDANT after each procedure. The basic paradigm is similar to ESPRESSO [8] except that IRREDUNDANT is called even after REDUCE. It is also similar to the procedure proposed

in [1] for the state minimization of incompletely specified deterministic machines, except that the detailed techniques of the proposed procedures are completely different.

5.6.3 REDUCE

REDUCE takes as input a feasible set C and processes one element of C at a time replacing it by its smallest subset, while maintaining feasibility.

Suppose $s^* \in C$ is being processed. Denote the characteristic function of the set $C - \{s^*\}$ by $C_1 : 2^S \rightarrow B$. Consider a function $L : 2^S \rightarrow B$ defined for the set of subsets of the states of T such that $L(\hat{s}^*) = 1$ if and only if $\hat{s}^* \subseteq s^*$ and $C - \{s^*\} \cup \{\hat{s}^*\}$ is closed. Specifically, for $\hat{s}^* \subseteq s^*$, $L(\hat{s}^*) = 1$ if and only if the following formula is satisfied.

$$\begin{aligned} \forall s_j^* \subseteq S: \ & C_1(s_j^*) = 1 \text{ or } s_j^* = \hat{s}^* \\ & \Rightarrow \exists f \in \mathcal{F}: \ \forall \mathbf{u} \in B^{|U|}: \exists s_k^* \subseteq S: \ C_1(s_k^*) = 1 \text{ or } s_k^* = \hat{s}^* \\ & \qquad\qquad \text{and} \\ & \qquad\qquad \tau(s_j^*, \mathbf{u}, f(\mathbf{u}), s_k^*) = 1 \end{aligned}$$

We represent L using a BDD, where a single Boolean variable is assigned for each state of T, i.e. a minterm of those Boolean variables corresponds to a subset of states and a state is in the subset if and only if the corresponding variable is 1 in the minterm. Then a smallest subset of s^* whose replacement preserves the closedness of the resulting set is given by a minterm $\hat{s}^*$ with the minimum number of 1's such that $L(\hat{s}^*) = 1$. It is known that such a minterm is given by a shortest path from the root of the BDD representing L to a terminal node with a label 1 in the BDD [32], where a weight of 1 (a unit weight) is assigned to every edge with a label 1 while the edges with label 0 have no weight. A shortest path of a BDD can be computed in linear time in the number of nodes of the BDD.

Therefore, the REDUCE procedure first sorts the elements of C, and for each element s^*, computes the function L. We actually restrict the domain of L to the states that are originally contained in s^* since our interests are in the subsets of s^* and the rest of the states are never included. If s^* does not contain the reset state r of T, then it simply computes $\hat{s}^*$ with the minimum cardinality such that $L(\hat{s}^*) = 1$, and replaces s^* by $\hat{s}^*$. In case s^* contains the reset state r and it is the last element that can contain r, i.e. all the other elements of C containing r have been already processed and none of the resulting elements contains r, then we restrict L so that all the members of L contain r, and then find one with the minimum cardinality among them to replace s^*. It is guaranteed that

the resulting set is feasible. The ordering of the elements of C currently used is decreasing order of the cardinality, i.e. we reduce the largest element first.

Note that the smallest subset $\hat{s}^*$ for s^* is computed with respect to the rest of the elements available in C at that time. Therefore, if another element of C is also processed, then we may be able to further replace $\hat{s}^*$ by an even smaller subset. For this reason, REDUCE has an option to iterate the replacement procedure over all the elements until no change in C occurs.

5.6.4 EXPAND

EXPAND also processes one element $s^* \in C$ at a time. It replaces s^* by $\hat{s}^* \supseteq s^*$ so that a maximal number of elements can be eliminated from C while feasibility is maintained.

Suppose that for given s^*, we compute $\hat{s}^* \supseteq s^*$ such that for some s_i^* in C with $s^* \neq s_i^*$, $C - \{s^*, s_i^*\} \cup \{\hat{s}^*\}$ is feasible. Consider how the relationship among the elements of C will be influenced if s^* is replaced by $\hat{s}^*$. Recall that s_j^* *needs* s_k^* in C if for every function f such that (s_j^*, f) is a compatible set, there exists $\mathbf{u} \in B^{|U|}$ for which s_k^* is the unique element in C that can be treated as the next state of s_j^*. Then the following might happen if s^* is replaced by $\hat{s}^*$.

(a) There exist s_j^* and s_k^* in $C - \{s^*\}$ such that s_j^* needs s_k^* in C but not in $C - \{s^*\} \cup \{\hat{s}^*\}$.

(b) There exists $s_k^* \in C - \{s^*\}$ such that s^* does not need s_k^* in C but $\hat{s}^*$ needs s_k^* in $C - \{s^*\} \cup \{\hat{s}^*\}$.

The case (a) happens since $\hat{s}^*$ contains more states than s^*, and we might be able to associate a new output function f with s_j^* for which s_k^* is not the unique next state of s_j^*. The case (b) happens since by expanding s^*, it might be no longer possible to associate some of the output functions with $\hat{s}^*$, which can be originally associated with s^*. Furthermore, if we eliminate s_i^* after the replacement, then the following might happen.

(c) There exist s_j^* and s_k^* in $C - \{s^*, s_i^*\}$ such that s_j^* does not need s_k^* in C but it does in $C - \{s^*, s_i^*\} \cup \{\hat{s}^*\}$.

This happens since by removing s_i^*, we might not be able to associate some output functions with s_j^*, even though they can be originally associated.

We regard the case (a) above as a positive influence while (b) and (c) are negative. However, we see that the case (c) does not happen if $\hat{s}^*$ contains s_i^*, since then every function that can be associated with s_j^* in C can be associated in $C - \{s^*, s_i^*\} \cup \{\hat{s}^*\}$. Also, if $\hat{s}^*$ contains s_i^*, then the set of states contained in some elements of $C - \{s^*, s_i^*\} \cup \{\hat{s}^*\}$ is identical with those

covered in $C - \{s^*\} \cup \{\hat{s}^*\}$. We need to keep track of such states during the procedure, and with this assumption, we can simply add the states of $\hat{s}^*$ to those covered in $C - \{s^*\}$ which are computed only once at the beginning of the procedure for s^*. Due to this efficiency as well as the influence for the case (c) above, we make the restriction, when $\hat{s}^*$ is computed to eliminate s^* and s_i^*, that $\hat{s}^*$ also contains s_i^*. Note that this restriction might increase the possibility of the case (b), but the current procedure does not take this into account.

Therefore, for each $s_i^* \in C - \{s^*\}$, we compute, if it exists, a smallest subset $\hat{s}^*(i)$ such that (1) $\hat{s}^*(i)$ contains both s^* and s_i^* and (2) $C - \{s^*, s_i^*\} \cup \{\hat{s}^*\}$ is closed. We choose a smallest such subset since we want to allow freedom of the expansion of s^* to eliminate other elements of C. To compute $\hat{s}^*(i)$, we first compute a function $H_i : 2^S \rightarrow B$ defined for the set of subsets of states of T such that $H_i(\bar{s}^*) = 1$ if and only if $\bar{s}^*$ contains both s^* and s_i^* and $C - \{s^*, s_i^*\} \cup \{\bar{s}^*\}$ is closed. Specifically, for $\bar{s}^*$ such that $\bar{s}^* \supseteq s^*$ and $\bar{s}^* \supseteq s_i^*$, $H_i(\bar{s}^*) = 1$ if and only if

$$\exists f \in \mathcal{F}: \quad \forall \mathbf{u} \in B^{|U|} : \exists s_k^* \subseteq S : \quad C_1(s_k^*) = 1 \text{ or } s_k^* = \bar{s}^* \text{ and } \tau(\bar{s}^*, \mathbf{u}, f(\mathbf{u}), s_k^*) = 1,$$

where C_1 is the characteristic function of $C - \{s^*\}$. Then by an argument similar to REDUCE, we see that $\hat{s}^*(i)$ is given by a shortest path of the BDD for H_i. Note that, unlike REDUCE, the function H_i may be empty, in which case there is no such $\hat{s}^*(i)$. In the actual computation, we restrict the domain of H_i to the set of states that are not included in either s^* or s_i^* since the rest of the states must be included in $\hat{s}^*(i)$.

If $\hat{s}^*(i)$ is obtained, we compute a set e_i of elements of $C - \{s^*\}$ that are contained in $\hat{s}^*(i)$. These are redundant elements, and can be eliminated. Note that there might exist elements in C that can be eliminated even though they are not contained in $\hat{s}^*(i)$, but we ignore them due to the computational efficiency. Once $\hat{s}^*(i)$'s are computed for all s_i^*, we choose $\hat{s}^*$ as the one with the largest cardinality over e_i's. Then s^* is replaced by $\hat{s}^*$, and all the elements contained in $\hat{s}^*$ are removed. We iterate this procedure, until no element can be removed by further expanding s^*.

Suppose the procedure above has been applied and s^* has been replaced by $\hat{s}^*$. At this point, although we know that no element of C can be further removed, there still might exist a superset $\bar{s}^* \supset \hat{s}^*$ such that the replacement of $\hat{s}^*$ by $\bar{s}^*$ preserves the feasibility of the resulting set. In this case, we compute a largest such $\bar{s}^*$ and replace $\hat{s}^*$. By doing this, we hope that the next time REDUCE is invoked, the element may be replaced by one different from s^*, by which a different portion of the solution space may be searched. Such $\bar{s}^*$ is obtained first by computing the function

H with the property that $H(\tilde{s}^*) = 1$ if and only if $\tilde{s}^* \supset \hat{s}^*$ and $C - \{\hat{s}^*\} \cup \{\tilde{s}^*\}$ is closed, and then find a longest path of the BDD for H. The function H can be computed in a way similar to that for H_i above.

As with REDUCE, it might be possible to further expand s^* by processing another element of C, and thus EXPAND has an option to iteratively apply the procedure above over all the elements, until none can be expanded further. Currently, EXPAND sorts the elements of C in increasing order of cardinality, before processing each element.

5.6.5 IRREDUNDANT

Given a feasible set C, the objective of IRREDUNDANT is to find a subset $\tilde{C}$ of C such that no proper subset of $\tilde{C}$ is feasible.

One might think that the goal can be achieved by checking the irredundancy for one element of C at a time and by successively removing redundant ones. Although the method is not expensive, there is no guarantee that an irredundant set is obtained at the end, since C might be redundant even though every single element of C is irredundant. In fact, according to our experiments, this approach is likely to get stuck at a bad solution at an early stage, unless an ordering to process the elements of C is carefully determined, which is very difficult in general.

The method we propose is an iterative computation of a feasible set starting from a seed set of elements of C. Let $C^{(k)}$ be a subset of C. Suppose, without loss of generality, that $C^{(k)}$ contains an element of C which contains the reset state r of T. If $C^{(k)}$ is closed, then we terminate the iteration. Otherwise, we compute a new set $C^{(k+1)}$ by adding the minimum number of elements of C to $C^{(k)}$ so that for each $s_i^* \in C^{(k)}$, there exists a function $f : B^{|U|} \rightarrow B^{|V|}$ such that for all $\mathbf{u} \in B^{|U|}$, there exists $s_j^* \in C^{(k+1)}$ that can be treated as the next state of s_i^*, i.e. $\tau(s_i^*, \mathbf{u}, f(\mathbf{u}), s_j^*) = 1$.

Let $s_i^* \in C^{(k)}$ be an element that does not satisfy the condition above. We introduce two sets of constraints for each such s_i^*. First, let $\mathcal{F}_{s_i^*}$ be a set of functions f such that for all $\mathbf{u}$, there exists s_j^* in C that can be treated as a next state, i.e. $\tau(s_i^*, \mathbf{u}, f(\mathbf{u}), s_j^*) = 1$. Associating a Boolean variable $w(s_i^*, f)$ for each $f \in \mathcal{F}_{s_i^*}$, we introduce a constraint $(\sum_{f \in \mathcal{F}_{s_i^*}} w(s_i^*, f))$. This constraint implies that at least one such function f must be chosen to associate with s_i^*. Secondly, for each function $f \in \mathcal{F}_{s_i^*}$, let $I(s_i^*, f)$ be the set of minterms $\mathbf{u} \in B^{|U|}$ for which the next state of s_i^* does not exist in $C^{(k)}$ under f. Namely, $\mathbf{u} \in I(s_i^*, f)$ if and only if there exists no $s_j^* \in C^{(k)}$ such that $\tau(s_i^*, \mathbf{u}, f(\mathbf{u}), s_j^*) = 1$. Then for each $\mathbf{u} \in I(s_i^*, f)$, we compute the set $D(s_i^*, f, \mathbf{u})$ of elements

of C that can be treated as the next states of s_i^* for $\mathbf{u}$ and f, i.e. $s_j^* \in C$ is in the set if and only if $\tau(s_i^*, \mathbf{u}, f(\mathbf{u}), s_j^*) = 1$. Then we introduce a constraint $(w(s_i^*, f) \Rightarrow \sum_{s_j^* \in D(s_i^*, f, \mathbf{u})} \gamma(s_j^*))$, where $\gamma(s_j^*)$ is a Boolean variable associated with s_j^*. The constraint means that if we choose the function f to associate with s_i^*, then at least one of the elements of C that can be treated as the next state of s_i^* for a given input $\mathbf{u}$ must be included to form $C^{(k+1)}$. Once these constraints are generated over all s_i^*, we find a minimum-weight assignment for the Boolean variables w and γ which satisfies all the constraints, where a unit weight is assigned to each variable $\gamma(s_j^*)$ while a variable $w(s_i^*, f)$ has no weight. This problem is known as a covering problem, and our procedure finds a solution using a method proposed in [33].

For the initial set of the computation above, we choose $C^{(0)} \subseteq C$ with the minimum cardinality such that there exists an element in $C^{(0)}$ which contains the reset state r, and for each $s_i^* \in C^{(0)}$, if s_i^* *needs* $s_j^* \in C$, then s_j^* is in $C^{(0)}$ as well. Note that there is no proper subset of $C^{(0)}$ that is irredundant.

Unfortunately, this method does not guarantee that the resulting set is irredundant because, even though we add a minimum number of elements at each iteration, there might exist s_i^* and s_j^* in $C^{(k)}$ such that s_i^* needs s_j^* in $C^{(k)}$ but not in $C^{(k+1)}$.

5.7 Experimental Results

The heuristic procedure proposed in the previous section has been implemented. As stated earlier, we use a BDD to represent the transition relation of a given pseudo non-deterministic machine T, where a single Boolean variable is used for each state. The implementation is restricted so that it only finds a behavior that can be represented by a Moore machine [38], i.e. the output function of the machine depends only on the present states of the machine and not on the inputs. There are two reasons for this restriction. The first is that we need to guarantee that the resulting behavior is implementable when the heuristic is applied for minimizing E-machines. Since the behavior is a Moore behavior, it is always possible to implement it so that there is no combinational loop in the resulting implementation no matter how M_2 is implemented. Secondly, with this restriction, an output function defined at each state simply becomes a minterm of $B^{|V|}$, and thus a compatible set is defined as a subset of states of T along with a minterm of $B^{|V|}$. In other words, all the functions f used in the procedure can be represented by single minterms of $B^{|V|}$. Hence, the characteristic functions used in the procedure are all represented by BDD's.

We conducted experiments using the implementation for optimizing systems of interacting finite state machines. The objective of the experiments was to see the effectiveness of taking into account information derived from other interacting components. We used the same examples as in the previous chapter for computing E-machines. Namely, in Figure 5.1, we chose two finite state machines, one for M_1 and the other for M_2. These are completely specified deterministic machines. We minimized M_1 in the number of states using an exact method for deterministic machines, similar to the one proposed in [44]. Thus, M_1 was originally made optimum in terms of the number of states without taking into account the interaction with M_2. Therefore, the number of states of M_1 is the optimum number of states required to represent the behavior given by M_1. We then made an arbitrary connection between M_1 and M_2, and confirmed that M_1 is implementable for M_2. Then we computed the E-machine T for M_1. Recall that the E-machine is pseudo non-deterministic with the same inputs and outputs as M_1 and captures the complete set of permissible behaviors, i.e. those that can be implemented at the position of M_1 while preserving the total product machine behavior of M_1 and M_2. Note that the number of states of the E-machine does not reflect any kind of optimality; it merely provides an upper bound on the minimized machines. Finally we invoked our heuristic procedure to find a feasible machine M_1'. At the end of the procedure, we verified the correctness of the solution. Note that due to our restriction of the implementation stated above, M_1' is a Moore machine. We then compared the number of states of M_1 and M_1'. The difference reflects the effectiveness of taking into account the interaction between M_1 and M_2 in further optimizing M_1, since M_1 was initially optimum.

In order to see better the effectiveness of the proposed method, i.e. the optimization of M_1 using the E-machine with the proposed heuristic, we computed the number of states of M_1 reachable when interacting with M_2. Specifically, after we minimized the number of states of M_1 by itself, we computed the set of reachable states of the product machine $M_1 \times M_2$, where each state of the product machine corresponds to a pair of states of M_1 and M_2. We then computed the set of states of M_1 which appear in at least one reachable state of the product machine. This is another technique of optimizing M_1 taking into account some of the interaction with M_2, although it is less powerful than the E-machine method. For example, the result depends on the structure of the initially provided machine M_1 whereas the E-machine captures the complete set of permissible behaviors. We also implemented an exact procedure for finding a minimum-state Moore behavior contained in the E-machine, which is described in Section 5.4.3. As stated in Section 5.4.3, the problem is reduced to a class of 0-1 integer linear programming problems, and the current implementation solves it using a method proposed in [28]. We applied the exact procedure for the same examples and compared

	M_1			M_2			E-machine	M_1'	Time (minimization)	M_{1r}
	In	Out	S	In	Out	S				
mc9	2	1	4	3	5	4	4	1	0.2 (0.1)	1
tm02	4	4	20	5	6	20	10	1	1.4 (0.5)	2
tm32	3	4	19	6	5	13	9	2	2.2 (0.5)	2
mt52	5	6	22	7	7	4	9	2	2.4 (2.0)	3
tm01	4	4	20	5	6	20	10	1	4.6 (1.7)	2
e69	2	1	4	5	8	8	8	1	4.7 (0.2)	1
pm11	8	8	26	10	10	24	9	1	5.0 (3.1)	7
pm12	8	8	26	10	10	24	7	3	5.2 (0.9)	5
e4bp1	5	5	24	6	9	14	11	1	12.3 (2.4)	7
L4	8	6	20	11	14	14	6	1	12.7 (7.7)	14
mt51	5	6	22	7	7	4	16	4	14.3 (7.4)	6
tm31	3	4	19	6	5	13	9	1	14.6 (0.4)	3
am9	6	6	25	7	8	4	13	5	15.6 (12.5)	9
pm03	2	4	11	6	4	14	15	1	16.3 (2.1)	8
L3	2	3	76	7	3	19	17	2	17.0 (8.5)	9
e4at2	5	4	21	6	9	14	14	4	31.7 (4.3)	4
e6tm	4	4	20	5	6	8	21	3	41.3 (30.8)	4
pm33	6	6	25	7	8	4	21	5	45.5 (35.3)	11
pm50	2	4	11	6	4	14	22	3	47.0 (9.7)	6
s3p1	5	5	24	7	7	13	38	6	140.3 (97.0)	7
pm41	2	4	11	6	4	14	33	5	162.5 (30.5)	8
pm31	6	6	25	7	8	4	22	6	187.0 (166.4)	9

	M_1			M_2			E-machine	Heuristic		Exact	
	In	Out	S	In	Out	S		S	Time	S	Time
tm32	3	4	19	6	5	13	9	2	0.5	2	392.1
mt52	5	6	22	7	7	4	9	2	2.0	2	164.9
pm12	8	8	26	10	10	24	7	3	0.9	3	74.1
L3	2	3	76	7	3	19	17	2	8.5	2	18.9

Table 5.1: Experimental Results

the resulting number of states with the one obtained by the heuristic procedure.

Table 5.1 shows the experimental results. Each row of the table shows the number of inputs **In** and the number of outputs **Out** for M_1 and M_2, as well as the number of states **S** of M_1, M_2, the E-machine, and M_1', the minimized machine using the heuristic. The number of inputs and outputs of the E-machine and M_1' are the same as M_1. **Time** is the CPU time in seconds used on a DECstation 5000/240 for the entire optimization in each experiment, including the computation of the E-machine as well as the minimization, where the number in a parenthesis is the time required for minimizing the E-machine by the proposed heuristic procedure. The column M_{1r} represents the number of states of M_1 reachable in the interaction with M_2, which is computed as described above. The small table attached at the bottom shows a comparison between the heuristic procedure and the exact procedure for finding a minimum-state Moore behavior contained in the E-machine. We applied the exact procedure only if a behavior found by the heuristic procedure had more than one state, since a behavior with a single state is already known to be optimum. The current implementation of the exact procedure was able to complete the computation only for four examples as shown in the table. The resulting number of states is shown in the column **S**, while the CPU time (seconds) is given in the column **Time**. The CPU time is the one required for the minimization only, and does not include the time for computing the corresponding E-machine.

Although the examples are rather small, we see that the number of states of the minimized machine M_1' is generally much smaller than that of the original machine M_1. The same observation can be made in the comparison between M_1 and M_{1r}, while further optimization can be achieved by using E-machines. For those examples where the exact procedure completed the computation, the heuristic always reproduced the optimum solutions in much less time. Considering the fact that M_1 was made optimum by itself in the beginning of the experiments, we see that the results demonstrate the effectiveness of accounting for the interaction between M_1 and M_2 in further optimizing M_1.

We conducted another type of experiments to see how much we restrict ourselves by focusing only on Moore behaviors. Recall that a machine M_1 does not have to be a Moore machine as long as it is possible to implement both M_1 and M_2 without introducing a combinational loop. This is always true if M_2 is a Moore machine. In this case, any behavior contained in the corresponding E-machine is implementable, and thus is a permissible behavior for M_1. Then the question is how much we lose by finding only a Moore behavior. To answer this question, we implemented an exact procedure for finding a minimum-state behavior contained in the E-machine, which is described in Section 5.4.1. We then applied this procedure as well as the exact procedure for finding a minimum-state Moore behavior contained in the E-machine for examples in which

	M_1			M_2					
	In	**Out**	**S**	**In**	**Out**	**S**	**E-machine**	**Contain**	**Moore**
ex1	2	3	20	4	4	20	22	3	3
ax4	5	6	20	8	8	26	11	1	1
ax5	5	6	8	8	8	26	26	2	2
ax6	6	5	13	8	8	26	22	1	1
ax7	3	5	4	6	6	25	20	2	2
ex10	3	4	19	5	6	22	13	1	2
bx7	3	5	4	6	6	25	23	2	2
ex12	3	4	19	6	4	14	13	1	1

Table 5.2: Comparison between Optimum Moore Behaviors and Optimum Contained Behaviors

M_2 is a Moore machine. Due to the computational complexity, there were only a small number of examples for which both of the exact methods were able to complete the computations. The results are shown in Table 5.2. The column **Contain** represents the number of states of a minimum-state behavior contained in the E-machine, while the column **Moore** shows the number of states of a minimum-state Moore behavior. The number of states of the E-machine is also shown for each example, where the number of inputs and outputs are the same as M_1. The minimum number of states is small for each of these examples, and therefore we cannot make a general statement on the limitation of finding Moore behaviors. However, as far as these examples are concerned, the number of states of an optimum Moore behavior is equal to the number of states of an optimum contained behavior, except for one example.

5.8 Concluding Remarks

We considered the problem of minimizing E-machines. The E-machine is a non-deterministic finite state machine, computed for a machine M_1 in Figure 5.1, with the property that the set of implementable behaviors in the E-machine is precisely the set of permissible behaviors for M_1, i.e. those that can be implemented at M_1 to meet the specification of the entire system. Using the number of states of a finite state machine as the cost function, we addressed the problem of finding an optimum permissible behavior in the E-machine, i.e. a behavior represented by a completely specified finite state machine with the minimum number of states over all the implementable behaviors in the E-machine.

This problem is analogous to the state minimization problem of finite state machines. Furthermore, a result given in the previous chapter shows that the E-machine is a special type of non-deterministic machine, called a pseudo non-deterministic machine. Hence, we first presented a theoretical analysis on the state minimization problem of pseudo non-deterministic finite state machines. We showed that the state minimization problem of general non-deterministic machines can be reduced to that for pseudo non-deterministic machines. Also, we presented how the basic concepts developed in the literature for the state minimization of deterministic machines can be generalized for this problem, in which we showed that the property of pseudo non-determinism can be effectively used to establish a correspondence between the original machine and the set of contained behaviors.

Based on this analysis, we presented an exact method for the state minimization of pseudo non-deterministic machines. We also presented an exact method for our original problem, i.e. the minimization of the E-machine. The exact method requires us to check the implementability of behaviors using dependency graphs introduced in the previous section. To avoid this complication, we presented an exact method for finding an optimum Moore behavior given in the E-machine.

We also presented a heuristic procedure for the state minimization problem of pseudo non-deterministic machines. The procedure keeps track of a feasible set of subsets of the states of a given pseudo non-deterministic machine, and tries to decrease its cardinality by iteratively introducing or removing redundancy in the set. The proposed procedure has been implemented with the restriction that the resulting machine is a Moore machine, and experiments were conducted. The results demonstrate the effectiveness of taking into account information derived from other components in optimizing systems of interacting finite state machines.

In the future, we want to remove the restriction of Moore behaviors, so that we can directly find a permissible behavior, rather than just a Moore behavior. Recall that the set of permissible behaviors is given by finding implementable behaviors among the behaviors contained in the E-machine. Currently, we need to explicitly use a dependency graph defined in Section 4.7 for identifying implementability. Since a dependency graph is defined with respect to a given particular behavior for M_1, we will need to construct a number of dependency graphs in order to find a set of implementable behaviors. If the implementability can be identified without explicitly formulating dependency graphs, then the proposed heuristic procedure can be extended for finding permissible behaviors.

Another problem to be addressed in the future is the state encoding of non-deterministic finite state machines. The proposed heuristic procedure keeps track of a set C of subsets of states

of a given pseudo non-deterministic machine T, and maintains the feasibility of C, i.e. a feasible machines can be composed using the elements of C. Now, note that although a feasible machine uniquely defines a behavior contained in T, a feasible machine defined by the set C may not be unique. In other words, it is possible in general to define a set of feasible machines for a given set C, for which there is a one-to-one correspondence between an element of C and a subset of states of T associated with each state of a feasible machine. Therefore the set C obtained by the heuristic defines a set of behaviors contained in T. Such a set of behaviors is given by the set of behaviors contained in a finite state machine T_C such that each state of T_C corresponds to an element of C and its transition relation is given by the relation τ defined in Section 5.6.1. Namely, for a pair of states of T_C corresponding to elements s_i^* and s_j^* of C respectively, T_C has a transition from s_i^* to s_j^* under an input $\mathbf{u}$ and an output $\mathbf{v}$ if and only if $\tau(s_i^*, \mathbf{u}, \mathbf{v}, s_j^*) = 1$, i.e. for all $s_i \in s_i^*$, there exists $s_j \in s_j^*$ such that $T(s_i, \mathbf{u}, \mathbf{v}, s_j) = 1$. The machine T_C is non-deterministic in general[2]. Then the question is which behavior should be chosen, in order to achieve the *best* implementation. For this question, one needs to address the encodings of states of T_C, i.e. how to assign a binary representation for each state so that a least-cost implementation of a behavior contained in T_C is optimum among all possible implementations of all the contained behaviors over all the encodings of the states.

[2]We conjecture that if C is irredundant, there exists a pseudo non-deterministic machine with $|C|$ states which contains the same set of behaviors with T_C.

Chapter 6

Conclusions

6.1 Summary of Thesis

This thesis addressed the problem of optimizing a synchronous digital system of interacting components. Specifically, we considered how to compute the set of behaviors that can be realized at a particular component of a system while preserving the behavior of the entire system. In addition we addressed how to find an *optimum* behavior in the set.

In Chapter 2 and Chapter 3, we considered combinational logic behaviors. The problem addressed in Chapter 2 is how to compute a set of Boolean functions that can be realized at a given component of the system, where the component may have multiple inputs and outputs. Each such function is called a permissible Boolean function of the component. We showed a condition under which a set of Boolean functions defined over the same inputs and outputs can be represented by a relation between the input and output spaces. We demonstrated that the complete set of permissible Boolean functions can be represented by a single Boolean relation, first shown by Brayton and Somenzi in [11]. Also, the problem of finding compatible sets of permissible functions was addressed. A set of compatible sets of permissible functions, one set for each component of the system, has the property that an arbitrary combination of functions, one from each of the compatible sets, results in an allowed behavior for the entire system. We considered how to compute a set of *maximally* compatible sets of permissible functions over the components, where compatible sets are maximal if there is no permissible function that can be newly added to any one of the sets without destroying the property of compatibility. We showed that each of the maximally compatible sets can also be represented by a single relation, and presented a procedure for computing such sets.

The problem of finding an optimum function realized at a given component was addressed

in Chapter 3. The problem is reduced to the minimization of a Boolean relation, and we proposed a heuristic for the problem with the cost function being the number of product terms required in a sum-of-products expression representing a function.

Chapter 4 and Chapter 5 is an analogous investigation for sequential logic behaviors. We considered a system in which the behavior of each component is modeled by a finite state machine and all the components are synchronizing. The definition of a permissible sequential behavior is identical to the combinational case, except that an additional constraint on implementability is required. A sequential behavior is said to be implementable if it is possible to implement the behavior so that no combinational loop is created in the entire system. This additional constraint was necessary because of our assumption that combinational loops, i.e. loops with no flip-flops or latches, typically do not appear in circuit implementations of practical synchronous digital systems. Furthermore, the constraint does not arise for the combinational case since the connections defined by the components of the system do not introduce a cycle for combinational behaviors. We presented, in Chapter 4, an analogous conclusion that the complete set of permissible sequential behaviors at a given component can be represented by a single non-deterministic finite state machine, which we call the E-machine. We proposed a procedure for computing the E-machine. We discussed how to identify implementable behaviors, and presented a necessary and sufficient condition under which a given sequential behavior is implementable.

The minimization of the E-machine was considered in Chapter 5, where the cost function was the number of states of a finite state machine required to represent a behavior. We showed that the E-machine is a special type of non-deterministic finite state machine, a pseudo non-deterministic machine. This property was effectively used for solving the problem. We proposed both exact and heuristic methods.

6.2 Future Directions

In the future, further investigation will be necessary to see how this work can be made practical. We describe our view for each type of behavior.

For combinational behaviors, intensive research has been done on the problem addressed in Chapter 2 for the case where each component has exactly one output [9]. As mentioned in Chapter 2, our experimental results do not provide significant effectiveness of the method for multiple outputs over the existing methods for single outputs. However, we think it is still too early to conclude that for practical applications, it is sufficient to only use single output methods. The experimental

results depend on how multi-output components are composed. Our experiments started with a system of single-output components, and clustered those to obtain multi-output components. Alternatively, one might use the proposed optimization procedure in conjunction with a factorization or a decomposition over multi-output components. A factorization or a decomposition is a technique used for modifying the structure of a system by changing its connections or introducing new components. In state-of-the-art optimization techniques developed for single-output combinational logic components, it is common that such techniques and optimizations for individual components are iteratively applied. Therefore, to see the effectiveness of optimization techniques for multi-output components, it will be necessary to consider how a factorization or a decomposition can be achieved for such components, and to use a procedure for optimizing individual components, e.g. the one proposed in Chapter 2, together with those global optimization procedures.

For sequential behaviors, on the other hand, the decomposition techniques for interacting finite state machines is still little explored, and such methods are not used in practice. However, often a hardware system is described as a set of interacting finite state machines, and thus it is important to identify the flexibility allowed for optimization in such a system by addressing the problems as done in Chapters 4 and 5. For the procedures discussed in this thesis, we still need further research to make them practical. One bottleneck for the current procedures is the time to compute the E-machine. One approach is to investigate a more efficient representation of a finite state machine. Another way, more directly related to the procedure of computing the E-machine, is to approximate the computation, i.e. compute a "subset" of permissible behaviors. An important question here is what is a good subset of permissible behaviors and how to find it. One way of computing a subset of permissible behaviors is to perform the same computation proposed in Chapter 4 for the E-machine, but to cease the iteration as soon as the computational time reaches a user-specified upper bound. Since the E-machine is constructed gradually starting from its reset state, all the implementable behaviors contained in the resulting machine are permissible, and thus the machine provides a subset of permissible behaviors. However, we do not know how good this subset is.

Alternately, we can abstract away many components of the surrounding components and then apply our procedures to get a conservative approximation. As an analogy to the combinational case, we can also define compatible sets of permissible sequential behaviors over all the components of the system. An interesting theoretical question is whether an analogy to the combinational case holds, i.e. if a maximally compatible set of permissible sequential behaviors can be represented by a single non-deterministic finite state machine. If this is the case, one could compute such a set for

each component by modeling the rest of the components using a single finite state machine M_2. In this case, unlike the context considered in Chapter 4, the machine M_2 would be a non-deterministic finite state machine.

Besides technical improvements on the efficiency of the proposed procedures, we also need to identify how effective it is in practice to take into account the interaction among components in the optimization. The example set used in our experiments were not obtained during a practical design process, and thus we have yet to see the practical effectiveness. As with the case of combinational logic behaviors, in order to discuss the practical effectiveness of the proposed procedures, it will be necessary to address how to perform a factorization or decomposition for a system of finite state machines, and to optimize the system by iteratively applying such global optimization techniques together with the proposed procedures for optimizing individual components.

Bibliography

[1] M. Avedillo, J. Quintana, and L. Huertas. Efficient State Reduction Methods for PLA-based Sequential Circuits. *IEE Proceedings-E*, 139(6):491–499, November 1992.

[2] T. Bartee, I. Lebow, and I. Reed. *Theory and Design of Digital Machines*. McGraw-Hill Book Company, Inc., 1962.

[3] K. Bartlett, R. K. Brayton, G. D. Hachtel, R. Jacoby, C. Morrison, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level Logic Minimization using Implicit Don't Cares. *IEEE Transactions on Computer-Aided Design*, CAD-7, June 1988.

[4] A. Booth and K. Booth. *Automatic Digital Calculators*. Butterworth Scientific Publications, 1953.

[5] E. Braun. *Digital Computer Design - Logic, Circuitry, and Synthesis*. Academic Press Inc., 1963.

[6] R. K. Brayton. New Directions in Logic Synthesis. In *Proceedings of the Synthesis and Simulation Meeting and International Interchange*, Kyoto, Japan, 1990.

[7] R. K. Brayton. private communication, 1994.

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

[9] R. K. Brayton, G. D. Hachtel, and A. Sangiovanni-Vincentelli. Multilevel Logic Synthesis. *Proceedings of the IEEE*, Vol. 78(No. 2), February 1990.

[10] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: Multiple-Level Logic Optimization System. *IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems*, Vol. CAD-6(No. 6):1062 – 1081, November 1987.

[11] R. K. Brayton and F. Somenzi. Boolean Relations and the Incomplete Specification of Logic Networks. In *International Conference on Very Large Scale Integration*, Munich, August 1989.

[12] R. E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, Vol. C-35(No. 8):677–691, August 1986.

[13] O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Based on Symbolic Execution. In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, 1989.

[14] M. Damiani. Nondeterministic Finite-State Machines and Sequential Don't Cares. In *European Conference on Design Automation*, 1994.

[15] M. Damiani and G. De Micheli. Derivation of Don't Care Conditions by Perturbation Analysis of Combinational Multiple-Level Logic Circuits. In *International Workshop on Logic Synthesis*, 1991.

[16] J. Darringer, W. Joyner, L. Berman, and L. Trevillyan. Logic Synthesis through Local Transformations. *IBM J. Res. Develop.*, pages 272–280, July 1981.

[17] G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal State Assignment for Finite State Machines. *IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems*, CAD-4:269 – 285, July 1985.

[18] S. Devadas. Approaches to Multi-Level Sequential Logic Synthesis. In *26th ACM/IEEE Design Automation Conference*, 1989.

[19] J. Eckert, Jr. Types of Circuits - General. In *Theory and Techniques for Design of Electronic Computers. Lectures given at the Moore School, 8 July 1946-31 August 1946.* University of Pennsylvania, 1947. Also in *The Moore School Lectures*, Vol. 9 in the Charles Babbage Institute Reprint Series for the History of Computing. The MIT Press, 1985.

[20] M. Fujita. Methods for Automatic Design Error Correction in Sequential Circuits. In *The European Conference on Design Automation with The European Event in ASIC Design*, February 1993.

[21] M. R. Garey and D. S. Johnson. *Computers and Intractability - A Guide to the Theory and NP-Completeness.* W.H. Freeman And Company, 1979.

[22] A. Ghosh, S. Devadas, and A. R. Newton. Heuristic Minimization of Boolean Relations using Testing Techniques. In *IEEE International Conference on Computer Design*, Cambridge, September 1990.

[23] S. Ginsburg. Synthesis of Minimal-State Machines. *IRE Transactions on Electronic Computers*, pages 441–419, December 1959.

[24] A. Grasselli and F. Luccio. A Method for Minimizing the Number of Internal States in Incompletely Specified Sequential Networks. *IEEE Transactions on Electronic Computers*, pages 350–359, June 1965.

[25] S. J. Hong, R. G. Cain, and D. L. Ostapko. MINI: A Heuristic Approach for Logic Minimization. *IBM J. Res. Develop.*, pages 443–458, September 1974.

[26] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, 1979.

[27] D. Huffman. The Synthesis of Sequential Switching Circuits. *Journal of Franklin Institute*, Vol. 257:161–190, 275–303, 1954. Also in E. Moore, editor, *Sequential Machines selected papers*. Addison-Wesley Publishing Company, 1964.

[28] T. Kam, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli. A Fully Implicit Algorithm for Exact State Minimization. In *31st ACM/IEEE Design Automation Conference*, 1994.

[29] J. Kim and M. Newborn. The Simplification of Sequential Machines with Input Restrictions. *IEEE Transactions on Computers*, C-21:1440–1443, December 1972.

[30] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill Book Company, New York, 1970.

[31] L. Lavagno, S. Malik, R. K. Brayton, and A. Sangiovanni-Vincentelli. MIS-MV: Optimization of Multi-level Logic with Multiple-valued Inputs. In *IEEE International Conference on Computer-Aided Design*, 1990.

[32] B. Lin and F. Somenzi. Minimization of Symbolic Relations. In *IEEE International Conference on Computer-Aided Design*, November 1990.

[33] H. Mathony. Universal Logic Design Algorithm and its Application to the Synthesis of Two-level Switching Circuits. *IEE Proceedings*, 136 Pt. E(3), May 1989.

[34] E. J. McCluskey Jr. Minimization of Boolean Functions. *Bell System Technical Journal*, Vol. 35:1417–1444, November 1956.

[35] G. Mealy. A Method for Synthesizing Sequential Circuits. Technical Report J. 34, Bell System Tech., 1955.

[36] Merrian Webster's Collegiate Dictionary. Merrian-Webster, Inc., tenth edition, 1993.

[37] J. Millman. *Microelectronics: Digital and Analog Circuits and Systems*. McGraw-Hill Book Company, 1979.

[38] E. Moore. Gedanken-experiments on Sequential Machines. In C. Shannon and J. McCarthy, editors, *Automata Studies*. Princeton University Press, 1956.

[39] S. Muroga, Y. Kambayashi, C. H. Lai, and J. N. Culliney. The Transduction Method - Design of Logic Networks based on Permissible Functions. *IEEE Transactions of Computers*, 1989.

[40] M. Paull and S. Unger. Minimizing the Number of States in Incompletely Specified Sequential Switching Functions. *IRE Transactions on Electronic Computers*, pages 356–367, September 1959.

[41] R. Puri and J. Gu. An Efficient Algorithm to Search for Minimal Closed Covers in Sequential Machines. *IEEE Transactions on Computer-Aided Design*, pages 737–745, June 1993.

[42] P. Ramadge and W. Wonham. Supervisory Control of a Class of Discrete Event Processes. *SIAM Journal of Control and Optimization*, Vol. 25(No. 1):206–230, January 1987.

[43] J. Rho, G. D. Hachtel, and F. Somenzi. Don't Care Sequences and the Optimization of Interacting Finite State Machines. In *International Workshop on Logic Synthesis*, 1991.

[44] J. Rho, G. D. Hachtel, F. Somenzi, and R. Jacoby. Exact and Heuristic Algorithms for the Minimization of Incompletely Specified State Machines. In *European Conference on Design Automation*, 1991.

[45] R. L. Rudell and A. Sangiovanni-Vincentelli. Multiple-Valued Minimization for PLA Optimization. *IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems*, Vol. CAD-6(No. 6):727 – 750, September 1987.

[46] T. Sasao. An Application of Multiple-Valued Logic to a Design of Programmable Logic Arrays. In *International Symposium on Multiple Valued Logic*, 1978.

[47] T. Sasao. Input Variable Assignment and Output Phase Optimization of PLA's. In *IEEE Transaction on Computers*, October 1984.

[48] H. Savoj. *Don't Cares in Multi-Level Network Optimization.* PhD thesis, U.C. Berkeley, March 1992.

[49] H. Savoj and R. K. Brayton. The Use of Observability and External Don't Cares for the Simplification of Multi-Level Networks. In *27th ACM/IEEE Design Automation Conference*, 1990.

[50] H. Savoj, R. K. Brayton, and H. Touati. Extracting Local Don't Cares for Network Optimization. In *IEEE International Conference on Computer-Aided Design*, 1991.

[51] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *IEEE International Conference on Computer Design*, 1992.

[52] T. R. Shiple, R. Hojati, A. Sangiovanni-Vincentelli, and R. K. Brayton. Heuristic Minimization of BDDs Using Don't Cares. In *31st ACM/IEEE Design Automation Conference*, June 1994.

[53] F. Somenzi and R. K. Brayton. An Exact Minimizer for Boolean Relations. In *IEEE International Conference on Computer-Aided Design*, November 1989.

[54] A. Srinivasan, T. Kam, S. Malik, and R. K. Brayton. Algorithms for Discrete Function Manipulation. In *IEEE International Conference on Computer-Aided Design*, pages 92–95, November 1990.

[55] R. E. Tarjan. *Data Structures and Network Algorithm.* Society for Industrial and Applied Mathematics, CBMS-NSF Regional Conference Series in Applied Mathematics, 1983.

[56] S. Unger. Flow Table Simplification - some useful aids. *IEEE Transactions on Electronic Computers*, June 1965.

[57] R. van Glabbeek. The Linear Time - Branching Time Spectrum. In Baeten J. and Klop J., editors, *CONCUR '90, Theories of Concurrency: Unification and Extension*, pages 278–297. Springer-Verlag, August 1990. Volume 458 of Lecture Notes in Computer Science.

[58] J. Vasudevamurthy and J Rajski. A Method for Concurrent Decomposition and Factorization of Boolean Expressions. In *IEEE International Conference on Computer-Aided Design*, 1990.

[59] H.-Y. Wang and R. K. Brayton. Input Don't Care Sequences in FSM Networks. In *IEEE International Conference on Computer-Aided Design*, November 1993.

[60] Y. Watanabe and R. K. Brayton. Incremental Synthesis for Engineering Changes. In *IEEE International Conference on Computer Design*, October 1991.

Index