STATE MINIMIZATION OF FINITE STATE MACHINES USING IMPLICIT TECHNIQUES

by

Timothy Yee-Kwong Kam

Memorandum No. UCB/ERL M95/37

16 May 1995

STATE MINIMIZATION OF FINITE STATE MACHINES USING IMPLICIT TECHNIQUES

by

Timothy Yee-Kwong Kam

Memorandum No. UCB/ERL M95/37

16 May 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

State Minimization of Finite State Machines
using Implicit Techniques

by

Timothy Yee-Kwong Kam

Doctor of Philosophy in Electrical Engineering and Computer Sciences

University of California at Berkeley

Professor Robert K. Brayton, Chair

State minimization is an important step in sequential synthesis of VLSI circuits. This dissertation addresses state minimization problems of various classes of finite state machines (FSM's). An exact algorithm usually consists of generation of compatibles and solution of a binate covering problem. State-of-the-art explicit minimizers fail on FSM's requiring an exponential number of compatibles, or huge binate tables. Such difficult examples do arise in practice.

This dissertation first contributes a fully implicit algorithm for exact state minimization of incompletely specified FSM's, and a software implementation called ISM. Novel techniques are developed to represent and generate various subsets of compatibles implicitly. ISM can handle sets of compatibles and prime compatibles of cardinality up to 2^{1500}. The dissertation also presents the first published algorithm for fully implicit exact binate covering. ISM can reduce and solve binate tables with up to 10^6 rows and columns. The entire branch-and-bound procedure is carried out implicitly.

To handle a more general and more useful class of FSM's, the first implicit algorithm for exact state minimization of pseudo non-deterministic FSM's is presented. Its implementation ISM2 is shown experimentally to be superior to a previous explicit formulation. ISM2 could solve exactly all but one problem of a set of published benchmarks, while the explicit program could complete approximately one half of the examples, and in those cases with longer running times.

A theoretical solution is presented for the problem of exact state minimization of general non-deterministic FSM's, based on the proposal of generalized compatibles. This gives an algorithmic foundation for exploring behaviors contained in an NDFSM.

Recent research in sequential synthesis, verification and supervisory control relies, as a final step, on the selection of an optimum behavior to be implemented at a component FSM within a network of FSM's. This dissertation contributes an exact condition characterizing when an FSM-composition is well-defined. Exact and heuristic algorithms are proposed to find a minimum behavior contained in a PNDFSM (capturing all permissible behaviors) such that it is Moore or its composition with another DFSM (representing the environment) is well-defined, and the global behavior meets a specification.

R. K. Brayton

Professor Robert K. Brayton
Dissertation Committee Chair

To my parents and my wife

Contents

List of Figures

List of Tables

Acknowledgements

But the Lord said to me,
"My grace is sufficient for you,
for my power is made perfect in weakness."
Therefore I will boast all the more gladly about my weaknesses,
so that Christ's power may rest on me.

– Apostle Paul, "2 Corinthians 12:9"

I am indebted to Professor Robert Brayton who has been my research advisor for more than five years. I learned from Bob, by example, how to be a researcher and scholar. I am grateful for his patience while I was trying to prove my ideas by experiments, and for the flexibility he allowed while I explored different research avenues. I am also very thankful for his guidance in my research and my career search.

I sincerely thank Professor Alberto Sangiovanni-Vincentelli for his strong interest in my research, and for being on my qualifying examination and dissertation committees and on most papers I published. Also, Alberto improved my presentation skills through several conference rehearsals. I also want to thank Professor Dorit Hochbaum who has kindly reviewed my dissertation, and together with Professor Jan Rabaey, served on my qualifying examination committee.

I would like to thank my family for their unfailing love and support, without whom my research and dissertation would not have been possible. I want to thank my parents for sowing seeds of hope and expectation in my life. I thank my mother who (literally) introduced me to libraries of books. I thank my father for all those nights he spent tutoring me when I was a teenager. My wife made my life as a graduate student so much more enjoyable and worthwhile; I just cannot thank her enough. I would also like to thank my sister and my brother-in-law, my parents-in-law, my grandmother, and Uncle Pui for their continual support and encouragements.

I owe a lot to Tiziano Villa, my research partner and good friend, who contributed substantially to the ideas in this dissertation. Being an optimistic person, at times he believed in my research more than I do, and as a result encouraged me to strive on. I always enjoy his company, with his wife and his daughter.

Beside Tiziano, there are many who have contributed to my research. I thank Dr. Yosinori Watanabe for generously sharing his ideas, programs, and benchmark examples with me. I thank Huey-Yih Wang, Dr. Arlindo Oliveira, and Stephen Edwards for providing me with interesting benchmark examples. Also I thank Professor Fabio Somenzi for some stimulating discussions.

I have the privilege to work with other brilliant researchers, including Dr. P.A. Subrahmanyam (thanking him for the summer at Bell Labs), Dr. Arvind Srinivasan, Dr. Luciano Lavagno, Yuji Kukimoto, Tianxiong Xue, and Professor Sharad Malik. I gratefully acknowledge research support from the Advanced Research Projects Agency under contracts J-FBI-90-073 and N00039-87-C-0182, the California State MICRO program, Digital Equipment Corporation and Intel.

I thank all members of the Berkeley CAD group who contribute the most wonderful place to conduct research. I thank Dr. William Lam and Dr. Edward Liu for the special friendship we developed over many lunches we had together. My special thanks are to: Adnan Aziz, Dr. Felice Balarin, Szu-Tsung Cheng, Sunil Khatri, Desmond Kirkpatrick, Chris Lennard, Dr. Cho Moon, Dr. Rajeev Murgai, Jagesh Sanghavi, Tom Shiple, Vigyan Singhal, Krishnan Sriram, Gitanjali Swamy, and Serdar Tasiran.

Patrick Byrne, Dr. Joshua Wong and Chiu-Ming Yip have been mentors to me in electronics. With their foresight and help, I ventured into the EECS graduate program at U.C. Berkeley which I'll never regret. I also thank Delon Cho, Kenneth Wong and Ka-Yin Li for their friendship. I am blessed by my many Christian friends: Simon Zeigler, the Gavel's (my American host family), Ole (thanking him for reviewing my dissertation) and Kelly Bentz, Jolly and Liz Chen, Alan and Dorene Marco, Tom Truman and Andrea Hoover, Gary and Patty Painter.

Finally, I thank God for bringing me to Berkeley and faithfully carrying me through every hurdle. He always prepares the best for me, and His grace has been more than sufficient for all these years. Thanks be to God!

Part I

Preliminaries

Chapter 1

Introduction

VLSI (Very large scale integrated) circuits are widely used in modern electronic products. Since the invention of the planar integrated circuit by Robert Noyce and Jack Kelby in 1959, the number of transistors that can be successfully fabricated on a single chip has doubled almost every year. As the complexity and performance requirements of VLSI circuits are increasing exponentially, the design process has to be automated by using computer-aided design (CAD) tools. A CAD tool is a computer program that can help an IC designer in the VLSI design process. This dissertation is concerned with the problem of automatically synthesizing a class of digital circuits.

Manual Design Vs. Automatic Synthesis

To best motivate the need for automatic synthesis, let us consider, as an example of the state-of-the-art VLSI design, the Intel's P6 microprocessor [15]. To stay ahead in the race for the next generation microprocessors, the most crucial factor in such a VLSI circuit design is probably the *design time*, which has a big effect on its *time-to-market* and ultimately to product success in the market. CAD tools hold the promise of shortening the design cycle by automating part of it. P6 consists of a CPU core with 5.5 million transistors and a secondary cache with 15.5 million transistors. For circuits with millions of transistors, the *design cost* may be considerable in terms of the number of design engineers involved and in term of the computer resources it requires. It is simply impossible to design such a complicated circuit manually, without any form of design automation. A hidden cost is when some design errors are not discovered early in the design cycle. This calls for *verification* tools which include *simulation*, and *formal verification*. Verification will not be covered in this dissertation, but instead we will concentrate on logic synthesis, i.e., CAD

tools that aids the design process.

Beside reducing the design time and cost, an important design objective is to improve the quality of the circuit. There are at least four aspects of design quality: *area*, *delay*, *power* and *testability*. Continuing on the P6 example, as SRAM has a regular structure, the 15.5 million transistors can be packed into a die of 202 mm^2. As the CPU probably contains a big portion of random logic, a die area of 306 mm^2 is used for the 5.5 million transistors there. Given a specification, one would like to obtain an implementation with a small area on silicon. Clock frequency is an important measure of the *performance* of a microprocessor. At a clock frequency of 133 MHz, the P6 microprocessor is claimed to deliver 200 SPECint92. The maximum clock frequency is dictated by the *delay* time of the critical path in the circuit. *Power consumption* is becoming more important as more transistors are packed into a single package. With both the CPU and SRAM dies housed in a single package, the peak power consumed by the P6 is estimated to be 20 Watts at a clock frequency of 133 MHz. *Testability* measures the ability to determine during production testing faults that occur within a chip. Note that both delay and power can be reduced as we scale the operating voltage and the process technology. To attain a small power-delay product, P6 is fabricated using a 2.9V, 0.6 micron, 4-layer-metal BiCMOS process technology. Further gain in circuit performance can be obtained only by the use of CAD tools.

Now let us concentrate on CAD tools that perform logic synthesis.

1.1 Logic Synthesis

Logic synthesis is a design automation process that generates a physical layout of a design from a user-input behavioral specification. In this section, we present a typical design flow and describe briefly each synthesis step involved . Each individual step can also be seen as a mapping, from a more abstract design representation to a less abstract one. Usually higher levels of abstraction are used to describe behaviors while the lower levels are more structural in nature.

High Level Synthesis

High level synthesis [55] translates a behavioral specification originated from the user into a register transfer level (RTL) description of the design. High level synthesis usually involves two tasks: *scheduling* and *allocation*. In the scheduling step, a specific time slot is determined for each of the operations in the specification such that some design constraints (such as area,

delay, etc) are satisfied while other factors are optimized. In the allocation step, operations and variables are assigned to functional units and memory units respectively. The end result of high level synthesis is an RTL description which consists of two parts: datapath and controller. The former is an interconnection of functional blocks (e.g., arithmetic units such as adders and multipliers), and memory units (e.g., registers). The control sequence for the datapath is generated by a controller which is described as a finite state machine (FSM). We will concentrate on the design of the control logic, which is usually time consuming.

Sequential Synthesis

The goal of sequential synthesis is to optimize control-oriented designs at the level of finite state machines. The result of sequential synthesis is a Boolean network of combinational logic blocks and memory elements. As the work presented in this dissertation is concerned with design tools for sequential synthesis, we shall discuss it in more detail in Section 1.2.

Combinational Logic Synthesis

After synthesis at the sequential level, the next step is to perform an optimization on the combinational portion of a circuit. The goal of combinational logic synthesis [8] is to find an optimal interconnection of primitive logic gates that realizes the same functionality as the original circuit. This process involves two steps: *technology independent optimization* and *technology mapping*. Technology independent optimization derives an optimal structure of the circuit independent of the gates available in a particular technology. Technology mapping is the optimization step to select the particular gates from a pre-designed library, to implement an optimized logic network. The optimization criteria are again: area, delay, power and testability.

Layout Synthesis

The final step in the synthesis flow is physical layout synthesis [41]. From a gate level description, it generates mask geometries of transistors and wires to be fabricated on silicon. Different layout tools are used for different design methodologies, such as full-custom, standard cell and gate array (including FPGA and sea-of-gates). Some common steps in layout synthesis are floorplanning, placement, routing, layout editing and module generation.

1.2 Sequential Logic Synthesis

In general, synthesis tools working at lower levels of abstraction are more mature than the ones operating at higher levels. The problems and solutions for layout synthesis and combinational logic (in particular two-level logic) synthesis are quite well understood. This fact is reflected by the wide use of commercial CAD tools in these synthesis domains. From a research perspective, sequential synthesis represents the next frontier in logic synthesis. Digital circuits, in general, are sequential in nature. It is desirable to work at the sequential level because CAD tools working at a higher level of abstraction can explore a wider design space. The main challenge is to efficiently explore this huge design space and obtain a quality solution within some reasonable memory space and CPU time. The subject of this dissertation is the design of algorithms and techniques for sequential synthesis with an emphasis on the exact state minimization problem.

Finite state machine (FSM) is a restricted class of sequential circuit called *synchronous* circuits which assumes the existence of a common global clock. An FSM is a discrete dynamic system that translates sequences of input vectors into sequences of output vectors. FSM's are a formalism growing from the theory of finite automata in computer science [32]. An FSM has a set of states and a set of transitions between states; the transitions are triggered by input vectors and produce output vectors. The states can be seen as recording the past input sequence, so that when the next input is seen, a transition can be taken based on the information of the past history. If a system is such that there is no need to look into past history to decide what output to produce, it has only one state and therefore it yields a *combinational circuit.* On the other hand, systems whose past history cannot be condensed in a finite number of states are not physically realizable.

FSM's are obtained either as a by-product of high level synthesis, or are directly given by the user. FSM's generated by high level synthesis tools are usually not minimum in the number of states. When directly specified by the user, FSM's are often designed for ease of description, instead of for compactness in representation. It is desirable to state minimize the FSM representation because usually it is a good starting point for other sequential synthesis and verification tools. Also a smaller number of states might result in less memory elements and simpler next state logic in the final implementation.

States in an FSM specification can either be *symbolic* or *binary-encoded.* Usually an FSM with symbolic states is initially given. To optimize the circuit at the sequential level, one can first perform *state minimization,* followed by *state assignment.* Given an FSM, state minimization produces another FSM whose behavior is equivalent to or contained in the behavior(s) of the

original FSM. Given an FSM with symbolic states, state assignment encodes these symbolic states into binary codes. An FSM can also be *partitioned* into a network of interacting FSM's. FSM partitioning is useful as current tools for exact state minimization and exact state assignment can handle only small to medium examples.

There is no theoretical guarantee that a state minimized FSM is always a better starting point for state assignment than an FSM that has not been state minimized [30], yet in practice this leads to excellent solutions, because it couples a step of behavioral optimization on the state transition graph (STG) with an encoding step on a reduced STG, so that the complexity of the latter's task is reduced.

1.3 Implicit Techniques

Efficient representation and exploration of design space is key to any sequential synthesis tool. We say that a representation is *explicit* if the objects it represents are listed one by one internally. Objects are manipulated explicitly, if they are processed one after another. *Implicit* representation means a shared representation of the objects, such that the size of the representation is not proportional to the number of objects in it. By implicit manipulation, we mean that in one step, many objects are processed simultaneously.

Seminal work by researchers at Bull [16] and improvements at U.C. Berkeley [80] produced powerful techniques for implicit enumeration of subsets of states of an FSM. These techniques are based on the idea of operating on large sets of states by their characteristic functions [13] represented by binary decision diagrams (BDD's) [11]. In many cases of practical interest, these sets have a regular structure that translates into small-sized BDD's. Once the related BDD's can be constructed, the most common Boolean operations on them (including satisfiability) have low complexity, and this makes it feasible to carry on computations unaffordable in the traditional case where all states must be explicitly represented. Of course, it may be the case that the BDD cannot be constructed, because of the intrinsic structure of the function to represent or because a good ordering of the variables is not found.

More recent work at Bull [19, 48] has shown how implicants, primes and essential primes of a two-valued or multi-valued function can also be computed implicitly. Reported experiments show a suite of examples where all primes could be computed, whereas explicit techniques implemented in ESPRESSO [7] failed to do so.

Therefore it is important to investigate how far these techniques based on implicit com-

putations can be pushed to solve the core problems of logic synthesis and verification. When exact solutions are sought, explicit techniques run out of steam easily because too many elements of the solution space must be enumerated. It appears that implicit techniques offer the most realistic hope to increase the size of problems that can be solved exactly. This dissertation is a first step on the application of implicit techniques to solve optimization problems in the area of sequential synthesis.

1.4 Contributions to State Minimization

State minimization is an important step in sequential synthesis of VLSI circuits. This dissertation addresses state minimization problems of various classes of finite state machines (FSM's). An exact algorithm [60, 28] usually consists of the generation of compatibles and the solution of a binate covering problem [68]. The state-of-the-art explicit minimizer [65] fails on FSM's requiring an exponential number of compatibles, or huge binate tables. We indicate also where such examples arise in practice.

This dissertation first contributes a fully implicit algorithm for exact state minimization of incompletely specified FSM's, and a software implementation called ISM. Novel techniques are developed to represent and generate various subsets of compatibles implicitly. ISM can handle sets of compatibles and prime compatibles of cardinality up to 2^{1500}. The dissertation also presents the first published algorithm for fully implicit exact binate covering. ISM can reduce and solve binate tables with up to 10^6 rows and columns. The entire branch-and-bound procedure is carried out implicitly.

To handle a more general and useful class of FSM's, the first implicit algorithm for exact state minimization of pseudo non-deterministic FSM's is presented. Its implementation ISM2 is shown experimentally to be superior to a previous explicit formulation [83]. ISM2 could solve exactly all but one problem of a set of published benchmarks, while the explicit program could complete approximately one half of the examples, and in those cases with longer running times.

A theoretical solution is presented for the problem of exact state minimization of general non-deterministic FSM's, based on the proposal of generalized compatibles. This gives an algorithmic foundation for exploring behaviors contained in an NDFSM.

Recent research [83, 4, 2] in sequential synthesis, verification and supervisory control relies, as a final step, on the selection of an optimum behavior to be implemented at a component FSM within a network of FSM's. This dissertation addresses the abstracted problem of finding a minimum behavior contained in a PNDFSM (capturing all permissible behaviors) such that its

composition with another DFSM (representing the environment) is well-defined and the global behavior meets a specification. This dissertation contributes a necessary and sufficient condition characterizing when such a composition is well-defined. Furthermore, exact and heuristic algorithms are proposed for selecting such a behavior contained in a PNDFSM which is either Moore, or well-defined with respect to another DFSM.

1.5 Organization of the Dissertation

The dissertation is divided into three parts. The first part presents some preliminary definitions, theories and techniques related to state minimization. The second part presents an implicit algorithm for exact state minimization of incompletely specified finite state machines (ISFSM's). The third part addresses state minimization problems of non-deterministic finite state machines (NDFSM's).

Chapter 2 provides a taxonomy on some useful classes of finite state machines, and their state minimization problems. At the end of this chapter, we prove the correctness of our state minimization algorithms for pseudo non-deterministic FSM's (of which ISFSM's are a subclass).

In Chapter 3, we show how we can represent and manipulate sets and sets-of-sets efficiently using a suite of implicit techniques.

An exact algorithm for state minimization of incompletely specified FSM's consists of two steps. Chapter 4 describes an implicit algorithm for the generation of compatibles and prime compatibles. Chapter 5 presents the first published algorithm for a fully implicit binate covering table solver.

Chapter 6 explores the state minimization problem of non-deterministic FSM's, and presents a fully implicit algorithm for exact state minimization of pseudo non-deterministic FSM's, a useful subclass of NDFSM's.

Recent advances indicate that synthesis of a component machine within a network of interacting FSM's requires the solution of specialized versions of the state minimization problem of PNDFSM's. Chapter 7 presents solutions to these problems, based on variations on our implicit minimization algorithm given in Chapter 6.

Finally Chapter 8 provides a conclusion of the theoretical and practical results obtained in this research.

Chapter 2

Taxonomy

The aims of this chapter are to provide a unified notational framework for the dissertation, and to introduce the theory behind state minimization. In this chapter, we first define some useful classes of finite state machines (FSM's) and finite automata (FA's), and investigate their inter-relationship. We will show that a non-deterministic FSM (NDFSM) can be used to specify a set of behaviors. Then we will describe how different behaviors can be explored within an NDFSM specification. These concepts introduced are central to exact algorithms for state minimization for FSM's. At the end, the correctness of our state minimization algorithms will be proved for a class of FSM's called the pseudo non-deterministic FSM's.

2.1 Taxonomy of Finite State Machines

We use characteristic functions proposed by Cerny [13] to represent sets and relations, both in theory and in practice. In the sequel, $B = \{0, 1\}$.

Definition 2.1 *Given a subset $S \subseteq U$ where U is some finite domain, the characteristic function of S, $\chi_S : U \rightarrow B$, is defined as follows. For each element $x \in U$,*

$$\chi_S(x) = \begin{cases} 0 \text{ if } x \notin S, \\ 1 \text{ if } x \in S. \end{cases}$$

Definition 2.2 *Given a relation $R \subseteq X \times Y$ where X and Y are some finite domains, the characteristic function of R, $\chi_R : X \times Y \rightarrow B$, is defined as follows. For each pair $(x, y) \in X \times Y$,*

$$\chi_R(x, y) = \begin{cases} 0 \text{ if } x \text{ and } y \text{ are not in relation } R, \\ 1 \text{ if } x \text{ and } y \text{ are in relation } R. \end{cases}$$

The above definition can be extended to any n-ary relations.

Definition 2.3 *A* **non-deterministic FSM** *(NDFSM), or simply an* **FSM**, *is defined as a 5-tuple* $M = \langle S, I, O, T, R \rangle$ *where* S *represents the finite state space,* I *represents the finite input space and* O *represents the finite output space.* T *is the transition relation defined as a characteristic function* $T : I \times S \times S \times O \rightarrow B$. *On an input* i, *the NDFSM at present state* p *can transit to a next state* n *and output* o *if and only if* $T(i, p, n, o) = 1$ *(i.e.,* (i, p, n, o) *is a transition). There exists one or more transitions for each combination of present state* p *and input* i. $R \subseteq S$ *represents the set of reset states.* [1]

Note that in this and subsequent definitions, the state space S, the input space I and the output space O can be generic discrete spaces and so S, I and O can assume symbolic values [21, 70]. A special case is when S, I and O are the Cartesian product of copies of the space $B = \{0, 1\}$, i.e., they are binary variables. The fact that the FSM's have symbolic versus binary encoded input and output variables does not change the formulation of problem, nor the solution based on the computation of compatibles. The theory extends in a straightforward manner to encoded state spaces.

The above is the most general definition of an FSM and it contains, as special cases, different well-known classes of FSM's. An FSM can be specified by a state transition table (STT) which is a tabular list of the transitions in T. An FSM defines a transition structure that can also be described by a state transition graph (STG). By a labeled edge $p \xrightarrow{i/o} n$, the FSM transits from state p on input i to state n with output o.

Definition 2.4 *Given an FSM* $M = \langle S, I, O, T, R \rangle$, *the state transition graph of* M *is a labeled directed graph,* $STG(M) = \langle V, E \rangle$, *where each state* $s \in S$ *corresponds to a vertex in* V *labeled* s *and each transition* $(i, p, n, o) \in T$ *corresponds to a directed edge in* E *from vertex* p *to vertex* n, *and the edge is labeled by the input/output pair* i/o.

To capture flexibility/choice/don't-care/non-determinism in the next state n and/or the output o from a present state p on an input i, one can specify one or more transitions $(i, p, n, o) \in T$. As said above, we assume that the state transition relation T is complete with respect to i and p, i.e., there is always at least one transition from each state on each input. This differs from the situation in formal verification where incomplete automata are considered.

[1] In subsequent definitions, R represents the set of reset states while r represents the unique reset state.

Definition 2.5 *A state transition relation T is complete if*

$$\forall i, p \, \exists n, o \, [T(i, p, n, o)] = 1.$$

$\forall$ is used to denote universal quantification, whereas $\exists$ denotes existential quantification. The above equation is an abbreviation of the following statement:

$$\forall i \in I \, \forall p \in S \, \exists n \in S \, \exists o \in O \text{ such that } T(i, p, n, o) = 1$$

Relational representation of T allows non-deterministic transitions with respect to next states and/or outputs, and also allows correlations between next states and outputs. More specialized forms of FSM's are derived by restricting the form of transitions that is allowed in T. FSM's can be categorized by answering the following questions:

1. Instead of a single transition relation, can the next state information and output information of the FSM be represented separately by two functions or relations?
2. Can the FSM be represented by functions instead of relations? [2]
3. Are the next states and outputs in the relations and/or functions correlated? Next state information is not correlated with the output if the former is related to the inputs and present states only. Uncorrelated output is similarly defined.

Now we introduce different classes of FSM's. An NDFSM is a pseudo non-deterministic FSM (PNDFSM) if for each triple $(i, p, o) \in I \times S \times O$, there is a unique state n such that $T(i, p, n, o) = 1$. It is non-deterministic because for a given input and present state, there may be more than one possible output; it is called pseudo non-deterministic because edges (i.e., transitions) carrying different outputs must go to different next states [3].

Definition 2.6 *A* **pseudo non-deterministic FSM** *(PNDFSM) is a 6-tuple $M = \langle S, I, O, \delta, \Lambda, R \rangle$. S represents the finite state space, I represents the finite input space and O represents the finite output space. δ is the next state function defined as $\delta : I \times S \times O \rightarrow S$ where each combination of input, present state and output is mapped to a unique next state. Λ is the output relation defined as a characteristic function $\Lambda : I \times S \times O \rightarrow B$ where each combination of input and present state is related to one or more outputs. $R \subseteq S$ represents the set of reset states.*

[2] Relations are denoted by upper-case letters (e.g., Δ denotes a next state relation) while functions are denoted by lower-case ones (e.g., δ represents a next state function).

[3] The underlying finite automaton of a PNDFSM is deterministic.

In other words for a PNDFSM, $T(i,p,n,o) = 1$ if and only if $\Lambda(i,s,o) = 1$ and $n = \delta(i,s,o)$. Since the next state n is unique for a given combination of input, present state and output, it can be given by a next state function $n = \delta(i,p,o)$. Since the output is non-deterministic in general, it is represented by the relation Λ.

Theorem 2.1 *An NDFSM is pseudo non-deterministic if*

$$\forall i,p,o\, !n\, [T(i,p,n,o)] = 1$$

where ! denotes a new operator called the *unique* quantifier which will be introduced in Section 3.3.2. $!x\ F(x,y) \stackrel{\text{def}}{=} \{y | y \text{ is related to a unique } x \text{ in } F\}$.

One can extend the previous definition to get a numerable family of machines as follows. An NDFSM is a k-step pseudo non-deterministic FSM (k-PNDFSM), where $k \in \omega$ (i.e., k is a natural number), if at any present state, the choice of the next state can be uniquely identified by observing input-output sequences of length up to k. An NDFSM is a k-step pseudo non-deterministic FSM (k-PNDFSM), where $k \in \omega$, if for each tuple $(i, i_2, \ldots, i_k, p, o, o_2, \ldots, o_k) \in I \times I \times \cdots \times I \times S \times O \times O \times \cdots \times O$, there is a unique next state $n \in S$ and there are states $s_2, \cdots, s_k \in S$ such that $T(i,p,n,o) = 1$ and $T(i_2, n, s_2, o_2) = T(i_3, s_2, s_3, o_3) = \cdots = T(i_k, s_{k-1}, s_k, o_k) = 1$.

Definition 2.7 *A* **k-step pseudo non-deterministic FSM** *(k-PNDFSM) is a 6-tuple $M = \langle S, I, O, \delta, \Lambda, R\rangle$. δ is the next state function defined as $\delta : I \times \cdots \times I \times S \times O \times \cdots \times O \to S$ where each combination of present state, k inputs and k outputs gives a unique next state. Λ is the output relation defined as a characteristic function $\Lambda : I \times S \times O \to B$ where each combination of input and present state is related to one or more outputs. $R \subseteq S$ represents the set of reset states.*

By definition, a PNDFSM is an 1-PNDFSM. Cerny in [14] has given a polynomial algorithm to convert a k-PNDFSM to a PNDFSM. A k-PNDFSM has a representation smaller or equal to that of an equivalent PNDFSM. We shall consider the state minimization of 1-PNDFSM only and assume that the k-PNDFSM can be handled by first going through such a conversion step.

Classical texts usually describe the Mealy and Moore model of FSM's. For completeness, they are also defined here as subclasses of NDFSM. A Mealy NDFSM is an NDFSM where there exists a next state relation [4] $\Delta : I \times S \times S \to B$ and an output relation [5] $\Lambda : I \times S \times O \to B$

[4] The next state relation Δ can be viewed as a function $\Delta : I \times S \to 2^S$; $n \in \Delta(i,p)$ if and only if n is a possible next state of state p on input i.

[5] The output relation Λ can be viewed as a function $\Lambda : I \times S \to 2^O$; $o \in \Lambda(i,p)$ if and only if o is a possible output of state p on input i.

such that for every $(i, p, n, o) \in I \times S \times S \times O$, $T(i, p, n, o) = 1$ if and only if $\Delta(i, p, n) = 1$ and $\Lambda(i, p, o) = 1$.

Definition 2.8 *A* **Mealy NDFSM** *is a 6-tuple $M = \langle S, I, O, \Delta, \Lambda, R \rangle$. S represents the finite state space, I represents the finite input space and O represents the finite output space. Δ is the next state relation defined as a characteristic function $\Delta : I \times S \times S \rightarrow B$ where each combination of input and present state is related to a non-empty set of next states. Λ is the output relation defined as a characteristic function $\Lambda : I \times S \times O \rightarrow B$ where each combination of input and present state is related to a non-empty set of outputs. $R \subseteq S$ represents the set of reset states.*

Note that next states and outputs are not correlated in the state transition relation of a Mealy machine.

A Moore NDFSM is an NDFSM where there exists a next state relation $\Delta : I \times S \times S \rightarrow B$ and an output relation $\Lambda : S \times O \rightarrow B$ such that for all $(i, p, n, o) \in I \times S \times S \times O$, $T(i, p, n, o) = 1$ if and only if $\Delta(i, p, n) = 1$ and $\Lambda(p, o) = 1$.

Definition 2.9 *A* **Moore NDFSM** *is a 6-tuple $M = \langle S, I, O, \Delta, \Lambda, R \rangle$. S represents the finite state space, I represents the finite input space and O represents the finite output space. Δ is the next state relation defined as a characteristic function $\Delta : I \times S \times S \rightarrow B$ where each combination of input and present state is related to a non-empty set of next states. Λ is the output relation defined as a characteristic function $\Lambda : S \times O \rightarrow B$ where each present state is related to a non-empty set of outputs. $R \subseteq S$ represents the set of reset states.*

As a special case of Mealy machine, Moore machines have its output depends on its present state only (but not input).

The definition of Moore machine presented here is the same as the one given by Moore in [58] and followed by authors in the field (e.g., [86]). The key fact to notice is that the output is associated with the present state. In other words, the common output associated to a given state, goes on all transitions that leave that state. This is a reasonable assumption when modeling an hardware system. However, it is common to find in textbooks [40, 32] a "dual" definition where the output is associated with the next state. In other words, the common output associated to a given state, is on all edges that go into that state, while edges leaving a given state may carry different outputs.

This second definition enjoys the nice property that it is always possible to convert a Mealy machine into a Moore machine. This may be the reason of its popularity. Instead with the

first definition, there are Mealy machines that have no Moore equivalent. For example, a wire can be consider a Mealy machine with one state and with its input connecting directly to its output. It does not have an equivalent Moore machine.

An NDFSM is an incompletely specified FSM (ISFSM) if for each pair $(i,p) \in I \times S$ such that $T(i,p,n,o) = 1$, (1) the machine can transit to a unique next state n or to any next state, and (2) the machine can produce a unique output o or produce any output.

Definition 2.10 *An* **incompletely specified FSM** *(ISFSM) can be defined as a 6-tuple* $M = \langle S, I, O, \Delta, \Lambda, R \rangle$. *$S$ represents the finite state space, I represents the finite input space and O represents the finite output space. Δ is the next state relation defined as a characteristic function $\Delta : I \times S \times S \rightarrow B$ where each combination of input and present state is related to a single next state or to all states. Λ is the output relation defined as a characteristic function $\Lambda : I \times S \times O \rightarrow B$ where each combination of input and present state is related to a single output or to all outputs. $R \subseteq S$ represents the set of reset states.*

Incomplete specification is used here to express some types of don't cares in the next states and/or outputs. We warn that even though "incompletely specified" is established terminology in the sequential synthesis literature, it conflicts with the fact that ISFSM's have a transition relation T that is actually completely specified with respect to present state p and input i, because there is at least one transition for each (i,p) pair in T.

A deterministic FSM (DFSM) or completely specified FSM (CSFSM) is an NDFSM where for each pair $(i,p) \in I \times S$, there is a unique next state n and a unique output o such that $T(i,p,n,o) = 1$, i.e., there is a unique transition from (i,p). In addition, R contains a unique reset state.

Definition 2.11 *A* **deterministic FSM** *(DFSM) or* **completely specified FSM** *(CSFSM) can be defined as a 6-tuple* $M = \langle S, I, O, \delta, \lambda, r \rangle$. *$S$ represents the finite state space, I represents the finite input space and O represents the finite output space. δ is the next state function defined as $\delta : I \times S \rightarrow S$ where $n \in S$ is the next state of present state $p \in S$ on input $i \in I$ if and only if $n = \delta(i,p)$. λ is the output function defined as $\lambda : I \times S \rightarrow O$ where $o \in O$ is the output of present state $p \in S$ on input $i \in I$ if and only if $o = \lambda(i,p)$. $r \in S$ represents the unique reset state.*

Note that an ISFSM and a DFSM are both next-state output uncorrelated because we can represent the next state and output information separately. But a PNDFSM (and k-PNDFSM) is next-state output correlated as the next state is correlated with the output by $n = \delta(i,p,o)$.

A Moore DFSM is a Moore NDFSM where for each pair $(i, p) \in I \times S$, there is a unique next state n and for each $p \in S$, a unique output o such that $T(i, p, n, o) = 1$. In addition, R contains a unique reset state.

Definition 2.12 *A* **Moore DFSM** *can be defined as a 6-tuple $M = \langle S, I, O, \delta, \lambda, r \rangle$. S represents the finite state space, I represents the finite input space and O represents the finite output space. δ is the next state function defined as $\delta : I \times S \rightarrow S$ where $n \in S$ is the next state of present state $p \in S$ on input $i \in I$ if and only if $n = \delta(i, p)$. λ is the output function defined as $\lambda : S \rightarrow O$ where $o \in O$ is the output of present state $p \in S$ if and only if $o = \lambda(p)$. $r \in S$ represents the reset state.*

Definition 2.13 *Given an NDFSM, a state s is output-deterministic if for every input i, there exists a unique output o such that s outputs o under input i. An NDFSM is* **output-deterministic** *(i.e., the NDFSM is an ODFSM) if every reachable state is output-deterministic.*

Theorem 2.2 *An NDFSM is output-deterministic if and only if*

$$\forall i, p \, !o \, \exists n \, [T(i, p, n, o)] = 1.$$

Definition 2.14 *An NDFSM is* **spurious non-deterministic** *if for each input sequences, there is a unique output sequence.*

A spurious non-deterministic FSM, as opposed to a true non-deterministic one, can be viewed as a DFSM. Is spurious non-determinism the same as output-determinism for NDFSM's?

Theorem 2.3 *If an NDFSM is spurious non-deterministic, then all its reachable states are output-deterministic.*

Proof: Given an NDFSM that is spurious non-deterministic. Assume that a reachable state s is not output-deterministic, there is an input i on which s can produce two different outputs. As s is reachable, say with input sequence σ_i from a reset state, the machine will produce different output sequences under the input sequence $\sigma_i i$. As this contradicts the fact that the NDFSM is spurious non-deterministic, our assumption was false. ■

2.2 Taxonomy of Finite Automata

Definition 2.15 *Given a finite set of symbols Σ (i.e., a finite alphabet), a* **language** *$\mathcal{L}$ is a set of strings of symbols from Σ.*

Definition 2.16 *A* **deterministic finite automaton** *(DFA), or simply* **finite automaton** *(FA), is defined as a 5-tuple $A = \langle S, \Sigma, \delta, r, F \rangle$. S represents the finite state space and Σ represents a finite alphabet. δ is the next state function defined as $\delta : \Sigma \times S \rightarrow S$ where $n \in S$ is the next state of present state $p \in S$ on symbol $i \in \Sigma$ if and only if $n = \delta(i, p)$. $r \in S$ represents the reset state. $F \subseteq S$ is the set of final states.*

The next state function δ can be extended to have as argument strings in Σ^*, (i.e., $\delta : \Sigma^* \times S \rightarrow S$) by $\delta(\rho i, s) = \delta(i, \delta(\rho, s))$.

A string x is said to be **accepted** *by the DFA A if $\delta(x, r)$ is a state in F. The* **language** *accepted by A, designated $\mathcal{L}(A)$, is the set of strings $\{x | \delta(x, r) \in F\}$.*

Definition 2.17 *A* **non-deterministic finite automaton** *(NDFA) is defined as a 5-tuple $A = \langle S, \Sigma, \Delta, r, F \rangle$. S represents the finite state space and Σ represents the finite alphabet. Δ is the next state relation defined as as a characteristic function $\Delta : \Sigma \times S \times S \rightarrow B$ where $n \in S$ is a next state of present state $p \in S$ on symbol $i \in \Sigma$ if and only if $n \in \Delta(i, p)$* [6]*. $r \in S$ represents the reset state.*

The next state relation can be extended to have as argument strings in Σ^* (i.e., $\Delta : \Sigma^* \times S \times S \rightarrow B$) as follows: $\Delta(\rho i, s, s'') = 1$ if and only if there exists $s' \in S$ such that $\Delta(\rho, s, s') = 1$ and $\Delta(i, s', s'') = 1$.

A string x is said to be **accepted** *by the NDFA A if there exists a sequence of transitions corresponding to x such that $\Delta(x, r)$ contains a state in F. The* **language** *accepted by A, designated $\mathcal{L}(A)$, is the set of strings $\{x | \Delta(x, r) \cap F \neq \emptyset\}$.*

Theorem 2.4 *Let $\mathcal{L}$ be the language accepted by a non-deterministic finite automaton. Then there exists a deterministic finite automaton that accepts $\mathcal{L}$.*

Proof: By subset construction [32]. ■

Corollary 2.5 *DFA's and NDFA's (and NDFA's with ϵ-transitions and regular expressions) accept the same set of languages.*

Proof: The class of languages accepted by NDFA's includes the set of languages accepted by DFA's (regular sets). Converse is also true because of the previous theorem. ■

[6] The next state relation Δ can be viewed as a function $\Delta : I \times S \rightarrow 2^S$; $n \in \Delta(i, p)$ if and only if n is a possible next state of state p on input i.

As DFA's and NDFA's accept the same set of languages, we shall refer to both as finite automata (FA's). However given an NDFA, its equivalent DFA may have an exponential number of states as compared with the NDFA.

Whereas a DFA is restricted to having exactly one transition from a state for each symbol, an NDFA may have any number of transitions for a given symbol, including zero transition. If when processing a string, an NDFA comes to a state from which there is no transition labeled with a symbol, the path followed is terminated. Actually, given an NDFA with missing transitions, an equivalent NDFA without any missing transition can be constructed by adding a unacceptable dummy state, and pointing all missing transitions to this dummy state.

An NDFA allows multiple reset states. Again given an NDFA with multiple reset states, an equivalent NDFA can be constructed with a new unique reset state.

2.3 Conversions between Finite State Machines and Finite Automata

Theorem 2.6 (DFA $\rightarrow$ DFSM) *Given a DFA $A = \langle S, I, \delta, r, F\rangle$, a DFSM $M = \langle S, I, O, \delta, \lambda, r\rangle$ can be constructed such that $O = \{0, 1\}$ and for every $i \in I$ and $s \in S$, $\lambda(i, s) = 1$ if and only if $\delta(i, s) \in F$.*

The DFSM M will output 1 after an input string if and only if the string read since the reset state is also accepted by the DFA A. We ignore the ϵ-string (i.e., the fact that the reset state of A may be acceptable) as it cannot be modeled by a Mealy DFSM.

Theorem 2.7 (NDFSM $\rightarrow$ NDFA) *Given an NDFSM $M = \langle S, I, O, T, R\rangle$, an NDFA $A = \langle S, I \times O, \Delta, R, S\rangle$ can be constructed such that for each $io \in I \times O$ (i.e., a symbol containing an input and an output part) and for each $s \in S$, $\Delta(io, p, n) = 1$ if and only if $T(i, p, n, o) = 1$. All states are acceptable in A.*

Theorem 2.8 *An FSM is pseudo non-deterministic if and only if the FA associated with it is deterministic.*

Note that although a given FSM is deterministic, the constructed finite automaton can still be non-deterministic because the next state n of a present state p is not always specified under every io pair in $I \times O$.

Theorem 2.9 (NDFSM $\rightarrow$ PNDFSM) *Given an NDFSM $M = \langle S, I, O, T, R\rangle$, an NDFA $A = \langle S, I \times O, \Delta, R, S\rangle$ can be constructed according to the above Theorem 2.7. The NDFA can then be*

"determinized" via subset construction [32] to obtain a DFA $A^{det} = \langle 2^S, I \times O, \Delta^{det}, R^{det}, 2^S \rangle$. *Then consider an NDFSM* $M^{det} = \langle 2^S, I, O, T^{det}, R^{det}, 2^S \rangle$ *where each symbol of* A^{det} *is splitted into an input and an output, and* $T^{det}(i, p, n, o) = 1$ *if and only if* $\Delta^{det}(io, p, n) = 1$. M^{det} *is pseudo non-deterministic.*

Theorem 2.10 (k-PNDFSM → 1-PNDFSM) *Construction given by Cerny in [14].*

2.4 Trace Sets and Behaviors

In the remaining sections of this chapter, we shall introduce different concepts that lead up to an exact algorithm for state minimization. Our discussion will be based on the state minimization of PNDFSM's because we believe it is the largest subclass of NDFSM's that our state minimization procedure can handle correctly. Miller has given a proof in [56] for the classical exact algorithm for ISFSM minimization [28]. We contribute a concise proof for our exact state minimization algorithm for PNDFSM's, by presenting a number of theorems in the following sections. In this section, we first show that a DFSM realizes a behavior while an NDFSM realizes a set of behaviors.

Definition 2.18 *Given a finite set of inputs* I *and a finite set of outputs* O, *a* **trace** *between* I *and* O *is a pair of input and output sequences* (σ_i, σ_o) *where* $\sigma_i \in I^*$, $\sigma_o \in O^*$ *and* $|\sigma_i| = |\sigma_o|$.

Definition 2.19 *A* **trace set** *is simply a set of traces.*

Definition 2.20 *An NDFSM* $M = \langle S, I, O, T, R \rangle$ **realizes** *a trace set between* I *and* O *from state* $s_0 \in S$, *denoted by* $\mathcal{L}(M|_{s_0})$ [7], *if for every trace* $(\{i_0, i_1, \ldots, i_j\}, \{o_0, o_1, \ldots, o_j\})$ *in the trace set, there exists a state sequence* $s_1, s_2, \ldots, s_{j+1}$ *such that* $\forall k : 0 \leq k \leq j, T(i_k, s_k, s_{k+1}, o_k) = 1$.

Definition 2.21 *An ISFSM* $M = \langle S, I, O, \Delta, \Lambda, R \rangle$ **realizes** *a trace set between* I *and* O *from state* $s_0 \in S$, *denoted by* $\mathcal{L}(M|_{s_0})$, *if for every trace* $(\{i_0, i_1, \ldots, i_j\}, \{o_0, o_1, \ldots, o_j\})$ *in the trace set, there exists a state sequence* $s_1, s_2, \ldots, s_{j+1}$ *such that* $\forall k : 0 \leq k \leq j$,

- $s_{k+1} \in \Delta(i_k, s_k)$, *and*
- $o_k \in \Lambda(i_k, s_k)$.

[7] If the NDFSM M is viewed as a NFA A which alphabet is $\Sigma = I \times O$, the trace set of M from a state s_0 corresponds to the language of A from s_0, and both will be denoted by $\mathcal{L}(M|_{s_0})$.

The trace set realized by a deterministic FSM with inputs I and outputs O is called a behavior between the inputs I and the outputs O. A formal definition follows.

Definition 2.22 *Given a finite set of inputs I and a finite set of outputs O, a* **behavior** *between I and O is a trace set, $\mathcal{B} = \{(\sigma_i, \sigma_o) \mid |\sigma_i| = |\sigma_o|\}$, 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. *Regularity:*
   There exists a DFSM $M = \langle S, I, O, \delta, \lambda, s_0 \rangle$ such that, for each $((i_0, \ldots, i_j), (o_1, \ldots, o_j)) \in \mathcal{B}$, there is a sequence of states $s_1, s_2, \ldots, s_{j+1}$ with the property that $s_{k+1} = \delta(i_k, s_k)$ and $o_k = \lambda(i_k, s_k)$ for every $k : 0 \leq k \leq j$.

For each state in a deterministic FSM, each input sequence corresponds to exactly one possible output sequence. Given a reset state, a deterministic FSM realizes a unique input-output behavior. But given a behavior, there can be (possibly infinitely) many DFSM's that realize the behavior. Thus, the mapping between behaviors and DFSM realizations is a one-to-many relation.

Any other kinds of FSM's, on the other hand, can represent a set of behaviors because by different choices of next states and/or outputs, more than one output sequence can be associated with an input sequence. Moreover, multiple reset states allow alternative trace sets be specified; depending on the choice of the reset state, a behavior within the trace set from the chosen reset state can be implemented. Therefore, while a DFSM represents a single behavior, a non-deterministic FSM (NDFSM) can be viewed as representing a set of behaviors. Each such behavior within its trace set is called a contained behavior of the NDFSM. Then an NDFSM expresses handily flexibilities in sequential synthesis. Using an NDFSM, a user can specified that one out of the set of behaviors is to be implemented. The choice of a particular behavior for implementation is based on some cost function such as the number of states. Other cost functions related to implementability will be investigated in Chapter 7.

2.5 Machine Containment

The fact that an NDFSM represents a set of behaviors leads naturally to the notion of behavioral containment or trace set inclusion between NDFSM's.

Definition 2.23 *An NDFSM's* $M = \langle S, I, O, T, R \rangle$ **behaviorally contains** *another NDFSM* $M' = \langle S', I, O, T', R' \rangle$, *denoted by* $\mathcal{L}(M) \supseteq \mathcal{L}(M')$, *if*[8] *for every* $r' \in R'$, *there exists* $r \in R$ *such that the trace set of* M *from* r *contains the trace set of* M' *from* r'. *i.e.,*

$$\mathcal{L}(M) \supseteq \mathcal{L}(M') \text{ if and only if } \forall r' \in R' \; \exists r \in R \; \mathcal{L}(M|_r) \supseteq \mathcal{L}(M'|_{r'}).$$

A different and more restrictive notion of containment is structural containment, which requires that the state transition graph (STG) of M' be embedded in the STG of M.

Definition 2.24 *An NDFSM's* $M = \langle S, I, O, T, R \rangle$ **structurally contains** *another NDFSM* $M' = \langle S', I, O, T', R' \rangle$, *denoted by* $STG(M) \supseteq STG(M')$, *if there exists a one-to-one mapping* $\varphi : S' \rightarrow S$ *such that*

1. $\forall r \in R' \; \varphi(r) \in R$, *and*
2. $\forall i \in I \; \forall p \in S' \; \forall q \in S' \; \forall o \in O$ *if* $T'(i, p, q, o) = 1$ *then* $T(i, \varphi(p), \varphi(q), o) = 1$.

Theorem 2.11 *If* $STG(M) \supseteq STG(M')$ *then* $\mathcal{L}(M) \supseteq \mathcal{L}(M')$. *The converse is not true in general.*

This theorem implies that in general, we cannot explore all behavior contained in the trace set of an NDFSM by enumerating DFSM's which are structurally contained within the NDFSM. But in Section 2.8, we shall show that all behaviors contained by a PNDFSM can be explored by finding DFSM's that are structurally contained in a derived machine called the power NDFSM.

2.6 Behavioral Exploration in PNDFSM's

In this section, we introduce the notion of a closed cover and establish the fact that by finding all closed covers of a PNDFSM, we can explore all behaviors contained in it.

Definition 2.25 *Given an NDFSM* $M = \langle S, I, O, T, R \rangle$, *a set of state sets,* $\{c_1, c_2, \ldots, c_n\}$, *is a* **cover** *of* M *if*[9] *there exists* $r \in R$ *and* $c_j : 1 \leq j \leq n$ *such that* $r \in c_j$.

Definition 2.26 *Given an NDFSM* $M = \langle S, I, O, T, R \rangle$, *a set of state sets,* $K = \{c_1, c_2, \ldots, c_n\}$, *is* **closed** *in* M *if for every* $i \in I$ *and* $c_j : 1 \leq j \leq n$, *there exists* $o \in O$ *and* $c_k : 1 \leq k \leq n$ *such that for each* $s \in c_j$, *there exists* $s' \in c_k$ *such that* $T(i, s, s', o) = 1$. *i.e.,*

$$\forall i \in I \; \forall c_j \in K \; \exists o \in O \; \exists c_k \in K \; \forall s \in c_j \; \exists s' \in c_k \; T(i, s, s', o) = 1$$

[8] *cf.* classical definition for ISFSM minimization.
[9] *cf.* classical definition for ISFSM minimization.

The above definition applies to all kinds of FSM's. The following lemma restricts the definition of closure condition to the case when M is a PNDFSM. This restriction to the PNDFSM subclass is necessary for some subsequent theorems to hold.

Lemma 2.12 *Given a PNDFSM $M = \langle S, I, O, \delta, \Lambda, R\rangle$, a set of state sets, $K = \{c_1, c_2, \ldots, c_n\}$, is* **closed** *in M if and only if for every $i \in I$ and $c_j : 1 \leq j \leq n$, there exists $o \in O$ such that*

1. *for each $s \in c_j$, $\Lambda(i, s, o) = 1$, and*

2. *there exists $c_k : 1 \leq k \leq n$ such that for every $s \in c_j$, $\delta(i, s, o) \in c_k$.*

i.e., $\forall i \in I\ \forall c_j \in K\ \exists o \in O\ \{\ [\forall s \in c_j\ \Lambda(i, s, o) = 1] \cdot [\exists c_k \in K\ \forall s \in c_j\ \delta(i, s, o) \in c_k]\ \}$.

Proof: By the definition of PNDFSM, $[T(i, s, s', o) = 1] \Leftrightarrow [\Lambda(i, s, o) = 1] \cdot [s' = \delta(i, s, o)]$. The closure condition for PNDFSM can be derived by a series of quantifier moves starting from the formula in Definition 2.26.

$$
\begin{aligned}
& \forall i \in I\ \forall c_j \in K\ \exists o \in O\ \exists c_k \in K\ \forall s \in c_j\ \exists s' \in c_k\ T(i, s, s', o) = 1 \\
\Leftrightarrow\ & \forall i \in I\ \forall c_j \in K\ \exists o \in O\ \exists c_k \in K\ \forall s \in c_j\ \exists s' \in c_k\ \{\ [\Lambda(i, s, o) = 1] \cdot [s' = \delta(i, s, o)]\ \} \\
\Leftrightarrow\ & \forall i \in I\ \forall c_j \in K\ \exists o \in O\ \exists c_k \in K\ \forall s \in c_j\ \{\ [\Lambda(i, s, o) = 1] \cdot [\exists s' \in c_k\ s' = \delta(i, s, o)]\ \} \\
\Leftrightarrow\ & \forall i \in I\ \forall c_j \in K\ \exists o \in O\ \exists c_k \in K\ \{\ [\forall s \in c_j\ \Lambda(i, s, o) = 1] \cdot [\forall s \in c_j\ \delta(i, s, o) \in c_k]\ \} \\
\Leftrightarrow\ & \forall i \in I\ \forall c_j \in K\ \exists o \in O\ \{\ [\forall s \in c_j\ \Lambda(i, s, o) = 1] \cdot [\exists c_k \in K\ \forall s \in c_j\ \delta(i, s, o) \in c_k]\ \}
\end{aligned}
$$

■

Lemma 2.13 *Given an ISFSM $M = \langle S, I, O, \Delta, \Lambda, R\rangle$, a set of state sets, $K = \{c_1, c_2, \ldots, c_n\}$, is* **closed** *in M if and only if for every $i \in I$ and $c_j : 1 \leq j \leq n$,*

1. *there exists $o \in O$ such that for each $s \in c_j$, $\Lambda(i, s, o) = 1$, and*

2. *there exists $c_k : 1 \leq k \leq n$ such that for every $s \in c_j$, there exists $s' \in c_k$ such that $\Delta(i, s, s') = 1$.*

i.e., $\forall i \in I\ \forall c_j \in K\ \{\ [\exists o \in O\ \forall s \in c_j\ \Lambda(i, s, o) = 1] \cdot [\exists c_k \in K\ \forall s \in c_j\ \exists s' \in c_k\ \Delta(i, s, s') = 1]\ \}$.

Proof: By the definition of ISFSM, $[T(i, s, s', o) = 1] \Leftrightarrow [\Lambda(i, s, o) = 1] \cdot [\Delta(i, s, s') = 1]$. The closure condition for ISFSM can be derived by a series of quantifier moves starting from the formula

in Definition 2.26.

$$\begin{aligned}
& \forall i \in I\, \forall c_j \in K\, \exists o \in O\, \exists c_k \in K\, \forall s \in c_j\, \exists s' \in c_k\, T(i,s,s',o) = 1 \\
\Leftrightarrow\ & \forall i \in I\, \forall c_j \in K\, \exists o \in O\, \exists c_k \in K\, \forall s \in c_j\, \exists s' \in c_k\, \{\, [\Lambda(i,s,o) = 1] \cdot [\Delta(i,s,s') = 1]\, \} \\
\Leftrightarrow\ & \forall i \in I\, \forall c_j \in K\, \exists o \in O\, \exists c_k \in K\, \forall s \in c_j\, \{\, [\Lambda(i,s,o) = 1] \cdot [\exists s' \in c_k\, \Delta(i,s,s') = 1]\, \} \\
\Leftrightarrow\ & \forall i \in I\, \forall c_j \in K\, \{\, [\exists o \in O\, \forall s \in c_j\, \Lambda(i,s,o) = 1] \cdot [\exists c_k \in K\, \forall s \in c_j\, \exists s' \in c_k\, \Delta(i,s,s') = 1]\, \}
\end{aligned}$$

■

Definition 2.27 *A set K of state sets is called a* **closed cover** *for $M = \langle S, I, O, T, R\rangle$ if*

1. *K is a cover of M, and*
2. *K is closed in M.*

Definition 2.28 *Let $M = \langle S, I, O, T, R\rangle$, and $K = \{c_1, c_2, \ldots, c_n\}$ be a closed cover for M where $c_j \in 2^S$ for $1 \le j \le n$, and $M' = \langle S', I, O, T', R'\rangle$ where $S' = \{s_1, s_2, \ldots, s_n\}$.*

K **is represented by** *M' (M' represents K) if for every $i \in I$ and $j : 1 \le j \le n$, there exists $k : 1 \le k \le n$ and $o \in O$ such that, if $T'(i, s_j, s_k, o) = 1$ then $\forall s \in c_j\ \exists s' \in c_k\, T(i,s,s',o) = 1$.*

Note that this definition implies a one-to-one mapping of K onto S'; in particular, $c_j \rightarrow s_j$ for $1 \le j \le n$. However, many different FSM's can represent a single closed cover.

Theorem 2.14 *If K is a closed cover for a PNDFSM M, then there exists a DFSM M' representing K. If K is represented by M', then $\mathcal{L}(M') \subseteq \mathcal{L}(M)$.*

Proof: Suppose $K = \{c_1, c_2, \ldots, c_n\}$ is a closed cover for $M = \langle S, I, O, T, R\rangle$. Then a DFSM $M' = \langle S', I, O, T', r'\rangle$ representing K can be constructed as follows. Let $S' = \{s_1, s_2, \ldots, s_n\}$ where there is a one-to-one correspondence between c_j and s_j for $1 \le j \le n$. As K is a cover, there exists a set $c_j : 1 \le j \le n$ and a reset state $r \in R$ such that $r \in c_j$. Pick one such c_j and choose r' to be the corresponding state s_j. As K is closed, for each $i \in I$ and $c_j : 1 \le j \le n$, there exists $c_k : 1 \le k \le n$ and $o \in O$ such that $\forall s \in c_j\ \exists s' \in c_k\, T(i,s,s',o) = 1$. Out of these possible choices of c_k and o, we choose an arbitrary transition (i, s_j, s_k, o) for T' from present state s_j on input i. As each (i, s_j) pair has a unique transition, M' is a DFSM. With this definition of M', it follows directly that M' satisfies Definition 2.28 so that K is represented by M'.

For the second part of the theorem, suppose $K = \{c_1, c_2, \ldots, c_n\}$ is a closed cover for M, and K is represented by $M' = \langle S', I, O, T', r' \rangle$; then $S' = \{s_1, s_2, \ldots, s_n\}$ by Definition 2.28. To prove that $\mathcal{L}(M') \subseteq \mathcal{L}(M)$ we must show that for the unique reset state r' of M', $\exists r \in R\, \mathcal{L}(M|_r) \supseteq \mathcal{L}(M'|_{r'})$. As K is a cover, $\exists r \in R\, r \in c_j$ for some $c_j : 1 \leq j \leq n$. We show $\mathcal{L}(M|_r) \supseteq \mathcal{L}(M'|_{s_j})$ by induction on the length k of an arbitrary input sequence, $i_1, i_2, \ldots, i_k$. As M' is a DFSM, the input sequence produces from state s_j a unique output sequence $o_1, o_2, \ldots, o_k$ and a unique state sequence $s_{j(1)}, s_{j(2)}, \ldots, s_{j(k)}$. We claim that the same input sequence when applied to M at r will produce the same output sequence, via some state sequence $r_1, r_2, \ldots, r_k$, where $r_m \in c_{j(m)}$ for $1 \leq m \leq k$. Remember that $r \in c_j$. For the base case $k = 1$, as K is closed, there exists $c_{j(1)} : 1 \leq j(1) \leq n$ such that for every $r_1 \in c_{j(1)}$, $T(i_1, r, r_1, o_1) = 1$. As M is a PNDFSM, there is a unique $r_1 \in c_{j(1)}$ corresponding to output o_1. Assume the induction hypothesis is true for $k-1$ where $k > 1$. So $r_{k-1} \in c_{j(k-1)}$. Again as K is closed, there exists $c_{j(k)} : 1 \leq j(k) \leq n$ such that for every $r_k \in c_{j(k)}$, $T(i_k, r_{k-1}, r_k, o_k) = 1$. $r_k \in c_{j(k)}$ is unique. Thus the hypothesis is proved.

■

Theorem 2.15 *Given a PNDFSM M and a DFSM M', if $\mathcal{L}(M') \subseteq \mathcal{L}(M)$ then there exists a closed cover for M which is represented by M'.*

Proof: Let $M = \langle S, I, O, T, R \rangle$ and $M' = \langle S', I, O, T', R' \rangle$ where $S' = \{s_1, s_2, \ldots, s_n\}$. We assume that $\mathcal{L}(M') \subseteq \mathcal{L}(M)$. Let $K = \{c_1, c_2, \ldots, c_n\}$ be the set of state sets defined as follows: $s \in c_j$ if and only if the trace set of M from s contains the trace set of M' from s_j, i.e., $\mathcal{L}(M|_s) \supseteq \mathcal{L}(M'|_{s_j})$.

By the definition of $\mathcal{L}(M') \subseteq \mathcal{L}(M)$, $\forall s_j \in R'$ $(1 \leq j \leq n)$ $\exists r \in R$ such that $\mathcal{L}(M|_r) \supseteq \mathcal{L}(M'|_{s_j})$. By construction of K, $r \in c_j$. Therefore K is a cover of M.

We now show K is closed, i.e., $\forall i \in I\ \forall c_j \in K\ \exists o \in O\ \{\ [(\forall s \in c_j\ \Lambda(i, s, o) = 1] \cdot [\exists c_k \in K\ \forall s \in c_j\ \delta(i, s, o) \in c_k)]\ \}$ according to Lemma 2.12. Consider an arbitrary input $i \in I$ and an arbitrary element $c_j \in K$. As M' is a DFSM, there exists a unique $o \in O$ such that $T'(i, s_j, s'_j, o) = 1$ for some s'_j. Now let us concentrate on this particular o. By the definition of K, we know for every $s \in c_j$, $\mathcal{L}(M|_s) \subseteq \mathcal{L}(M'|_{s_j})$. The trace (i, o) is in $\mathcal{L}(M'|_{s_j})$, and therefore is also in $\mathcal{L}(M|_s)$. So $\forall s \in c_j\ \Lambda(i, s, o) = 1$. We now prove the last term in the conjunction. Assume $\exists c_k \in K\ \forall s \in c_j\ \delta(i, s, o) \in c_k$ is false. For each $c_k \in K$, there exists a pair of states $\{\delta(i, p_k, o), \delta(i, q_k, o)\} \not\subseteq c_k$, where p_k and q_k are elements of c_j. Now $\mathcal{L}(M'|_{s_k})$ is not contained in both $\mathcal{L}(M|_{\delta(i,p_k,o)})$ and $\mathcal{L}(M|_{\delta(i,q_k,o)})$, since $\{\delta(i, p_k, o), \delta(i, q_k, o)\} \not\subseteq c_k$; thus on some input sequence $i_1, i_2, \ldots, i_p$, $\delta(i, p_k, o)$ and $\delta(i, q_k, o)$ cannot produce the same output by M.

Then on input sequence $i, i_1, i_2, \ldots, i_p$, p_k and q_k cannot produce the same output by M. This implies that p_k and q_k are not included in any $c_j : 1 \leq j \leq n$, which contradicts the assumption and proves $\exists c_k \in K \; \forall s \in c_j \; \delta(i,s,o) \in c_k$. In summary, for arbitrary i and c_j, the formula $\exists o \in O \; \{ \; [(\forall s \in c_j \; \Lambda(i,s,o) = 1] \cdot [\exists c_k \in K \; \forall s \in c_j \; \delta(i,s,o) \in c_k)] \; \}$ is true. Therefore K is closed. ■

Note in Theorem 2.15, M cannot be easily generalized to an NDFSM, because the above proof would not go through.

The following is the main theorem proving the correctness of exact state minimization algorithms which are based on closed covers. It shows that one can explore all behaviors contained within a PNDFSM by find all closed covers for the PNDFSM.

Theorem 2.16 *Let M be a PNDFSM and M' be a DFSM. $\mathcal{L}(M') \subseteq \mathcal{L}(M)$ if and only if there exists a closed cover for M which is represented by M'.*

Proof: The *only if* part immediately follows from Theorem 2.14. The *if* part immediately follows from Theorem 2.15. ■

2.7 State Minimization Problems

Definition 2.29 *Given an NDFSM $M = \langle S, I, O, T, R \rangle$, the* **state minimization problem** *is to find a DFSM $M' = \langle S', I, O, T', R' \rangle$ such that*

1. *$\mathcal{L}(M') \subseteq \mathcal{L}(M)$, and*
2. *$\forall M''$ such that $\mathcal{L}(M'') \subseteq \mathcal{L}(M)$, $|S'| \leq |S''|$.* [10]

Such a case is denoted by $\mathcal{L}(M') \overset{min}{\subseteq} \mathcal{L}(M)$.

M' is not required to be deterministic. On the contrary, to preserve flexibility for other sequential synthesis tools, one may extend the definition and choose the M' with maximal non-determinism in specification, out of all state minimum machines.

Definition 2.23 of behavioral containment and Definition 2.29 of the state minimization problem apply also to other kinds of FSM's, which are subclasses of PNDFSM's.

Definition 2.30 *A closed cover K for M is called a* **minimum closed cover** *for M if every closed cover K' of M has $|K'| \geq |K|$.*

[10] Given a set S, $|S|$ denotes the cardinality of the set.

The following theorem is a companion and an extension of Theorem 2.16. It proves the optimality of exact state minimization algorithms which find minimum closed covers.

Theorem 2.17 *Let M be a PNDFSM and M' be a DFSM. $\mathcal{L}(M') \overset{min}{\subseteq} \mathcal{L}(M)$ if and only if there exists a minimum closed cover for M which is represented by M'.*

Proof: Assume $\mathcal{L}(M') \overset{min}{\subseteq} \mathcal{L}(M)$. By Theorem 2.15, there exists a closed cover K for M which is represented by M'. Suppose there exists a closed cover K'' for M such that $|K''| < |K|$. By Theorem 2.14, there exists a DFSM M'' representing K'' such that $\mathcal{L}(M'') \subseteq \mathcal{L}(M)$. This contradicts with our initial assumption so K must be a minimum closed cover for M.

Assume K is a minimum closed cover for M. By Theorem 2.14, there exists a DFSM M' representing K and $\mathcal{L}(M') \subseteq \mathcal{L}(M)$. If $\mathcal{L}(M'') \overset{min}{\subseteq} \mathcal{L}(M)$, there exists a minimum closed cover K'' for M such that K'' represents M''. However K must have the same number of sets as K'' since K is also a minimum closed cover for M. Thus M' has the same number of states as M'' so that $\mathcal{L}(M') \overset{min}{\subseteq} \mathcal{L}(M)$. ■

The state minimization problem defined above is very different from the NDFA minimization problem described in classical automata textbooks. Here we want a minimal state implementation which is contained in the specification, while the classical problem is defined as:

Definition 2.31 *Given an NDFSM M, the* **behavior-preserving state minimization problem** *is to find an NDFSM M' which represents the same set of behavior as M but has the fewest number of states.*

The above notion of behavior-preserving state minimization may be useful in formal verification because all behaviors specified must be verified and thus must be *preserved* during minimization.

2.8 Power NDFSM and Structural Containment

Now let us revisit the question: Can we explore behaviorally contained DFSM's by structural containment? This is an interesting question because intuitively, structural containment is easier to compute than behavioral containment. In this section, we shall show that given a PNDFSM M, we can derive a new NDFSM called M^{power} such that M' is behaviorally contained in M if and only if M' is structurally contained in M^{power}.

Definition 2.32 *Given a PNDFSM* $M = \langle S, I, O, T, R \rangle$, *the* **power NDFSM** *is* $M^{power} = \langle 2^S, I, O, T^{power}, R^{power} \rangle$. T^{power} *is a transition relation in* $I \times 2^S \times 2^S \times O$. $T^{power}(i, c_j, c_k, o) = 1$ *if and only if* $\forall s \in c_j \; \exists s' \in c_k \; T(i, s, s', o) = 1$. $R^{power} \subseteq 2^S$ *includes every state set which contains a reset state* $r \in R$.

Note that M^{power} is not pseudo non-deterministic because given any transition $(i, c_j, c_k, o) \in T^{power}$, for every $\hat{c_k} \supset c_k$, there is another transition such that $T^{power}(i, c_j, \hat{c_k}, o) = 1$ and $R^{power}(c) = 1$.

Definition 2.33 *Given a power NDFSM* $M^{power} = \langle 2^S, I, O, T^{power}, R^{power} \rangle$, *the power NDFSM with its state space restricted to a set* K *of state sets, denoted by* $M^{power}|_K$, *is the NDFSM* $\langle K, I, O, T^{power}|_K, R^{power}|_K \rangle$. $T^{power}|_K(i, c, c', o) = 1$ *if and only if* $c, c' \in K$ *and* $T^{power}(i, c, c', o) = 1$. $R^{power}|_K(c) = 1$ *if and only if* $c \in K$ *and* $R^{power}(c) = 1$.

Lemma 2.18 *K is a closed cover if*

1. $K \cap R^{power} \neq \emptyset$, *and*
2. *for every* $i \in I$ *and* $c_j \in K$, *there exists* $c_k \in K$ *and* $o \in O$ *such that* $T^{power}(i, c_j, c_k, o) = 1$.

Proof: Condition 1 of Lemma 2.18 is true if and only if there exists c_j such that $c_j \in R^{power}$ and $c_j \in K$. By Definition 2.32, $c_j \in R^{power}$ implies that there exists a reset state $r \in R$ such that $r \in c_j$. As $c_j \in K$, K is a cover according to Definition 2.25. By substituting the definition of T^{power} into condition 2 of Lemma 2.18, the closure condition as expressed by Definition 2.26 is satisfied. ■

This lemma suggests an alternative way of finding a closed cover, K. Looking to the power PNDFSM M^{power}, K corresponds to a set of states in 2^S such that (1) at least one state in K is a reset state in R^{power}, and (2) for each input i, each state in K can have a next state also in K under transitions in T^{power}. This selection procedure is analogous to finding a DFSM which is structurally contained within the power PNDFSM, and covers at least one reset state.

Lemma 2.19 *K is represented by M' if the power NDFSM with its state space restricted to K structurally contains M'. i.e., $STG(M^{power}|_K) \supseteq STG(M')$.*

Proof: By Definition 2.28, K is represented by M' if and only if for every $i \in I$ and $j : 1 \leq j \leq n$, there exists $k : 1 \leq k \leq n$ and $o \in O$ such that, if $T'(i, s_j, s_k, o) = 1$ then $T^{power}(i, c_j, c_k, o) = 1$. Let $M^{power}|_K$ be the power NDFSM of M with its state space restricted to the set K. By Definition 2.24, K is represented by M' if and only if $STG(M^{power}|_K) \supseteq STG(M')$. ■

Now we are ready to extend Theorem 2.16 and provide another way of exploring behaviors contained within a PNDFSM, beside enumerating closed covers.

Theorem 2.20 *Let M be a PNDFSM and M' be a DFSM. The following statements are equivalent:*

1. $\mathcal{L}(M') \subseteq \mathcal{L}(M)$.
2. *There exists a closed cover for M which is represented by M'.*
3. *There exists a subset $K \subseteq 2^S$ such that*
   - $K \cap R^{power} \neq \emptyset$, *and*
   - $\forall i \in I \; \forall c_j \in K \; \exists c_k \in K \; \exists o \in O \; T^{power}(i, c_j, c_k, o) = 1$, *and*
   - $STG(M') \subseteq STG(M^{power}|_K)$.

Proof: Equivalence of statements 1 and 2 follows from Theorem 2.16. By applying Lemmas 2.18 and 2.19 and gathering their conditions, statement 2 becomes statement 3. ■

Definition 2.34 *A DFSM M' is a minimum structurally contained DFSM in a PNDFSM M, denoted by $STG(M') \overset{min}{\subseteq} STG(M)$, if*

1. $STG(M') \subseteq STG(M)$, *and*
2. $\forall M''$ *such that* $STG(M'') \subseteq STG(M)$, $|S'| \leq |S''|$.

Theorem 2.21 *Let M be a PNDFSM and M' be a DFSM. The following statements are equivalent:*

1. $\mathcal{L}(M') \overset{min}{\subseteq} \mathcal{L}(M)$.
2. *There exists a minimum closed cover for M which is represented by M'.*
3. *There exists a subset $K \subseteq 2^S$ such that*
   - $K \cap R^{power} \neq \emptyset$, *and*
   - $\forall i \in I \; \forall c_j \in K \; \exists c_k \in K \; \exists o \in O \; T^{power}(i, c_j, c_k, o) = 1$, *and*
   - $STG(M') \overset{min}{\subseteq} STG(M^{power}|_K)$.

Theorem 2.21 summarizes succinctly the theory we know about state minimization.

Although we have the theory, we do not yet have an exact algorithm for state minimization which explores structurally contained DFSM's within the power NDFSM. However, this is not due to an inability to represent M^{power} compactly as it can be represented implicitly as shown in Chapter 6. In Chapter 7, we will describe a heuristic procedure to find a minimal DFSM structurally contained in the power NDFSM. In the next section, we'll outline the common steps used in state minimization algorithms which are based on closed cover, for different subclasses of NDFSM's.

2.9 Exact Algorithm for State Minimization

By Theorem 2.17, the state minimization problem of PNDFSM can be reduced to the problem of finding a minimum closed cover for the PNDFSM. From what we know so far, a closed cover may contain any arbitrary set of states. Once such a minimum closed cover is found, a minimum state DFSM which represents the closed cover can be obtained easily, and this final step is traditionally called **mapping**. A brute force approach to find a minimum closed cover would be to enumerate sets of state sets, and test each one of them to see if it represents a closed cover according to Definition 2.25 and Lemma 2.12. Out of all the closed covers, one would pick one of minimum cardinality. This section will discuss ways to improve on this brute force algorithm.

In the exact method described above, each candidate closed cover includes subsets of 2^S where S is the state space of the PNDFSM. The number of such state sets to be generated is exponential in the number of states, $|S|$. Actually we do not have to consider all subsets, but only those that will constitute a closed cover. The definition of a closed cover requires that it contains subsets only of the following types.

Definition 2.35 *A set of states is an* **output compatible** *if for every input, there is a corresponding output which can be produced by each state in the set.*

Lemma 2.22 *Every element of a closed cover* [11] *is an output compatible.*

Proof: By Definition 2.26, for each set $c_j : 1 \leq j \leq n$ on every input $i \in I$, there is an output $o \in O$ that can be produced by each state in the set. Therefore c_j is an output compatible. ■

Definition 2.36 *A set of states is a* **compatible** *if for each input sequence, there is a corresponding output sequence which can be produced by each state in the compatible.*

[11] This more restrictive lemma is also true: Every element of a closed set is an output compatible.

Lemma 2.23 *Every element of a closed cover is a compatible.*

Proof: The lemma can be proved by showing that if K is a closed set of output compatibles then K is a closed set of compatibles. (The converse is trivially true because a compatible is also an output compatible.) Assume K is a closed set of output compatibles but there exists $c_j \in K$ such that $p_1 \in c_j$ and $q_1 \in c_j$ are not compatible. Thus there exists some input sequence $i_1, i_2, \ldots, i_k$ to M which produces state sequences $p_1, p_2, \ldots, p_k$ and $q_1, q_2, \ldots, q_k$, respectively. And there is no output $o \in O$ such that $T(i, p_k, p_{k+1}, o) = 1$ and $T(i, q_k, q_{k+1}, o) = 1$, for any p_{k+1} and q_{k+1}. As K is closed, for any $m : 1 \leq m \leq n$, $\{p_m, q_m\}$ is contained in some set in K. However $\{p_k, q_k\}$ contradicts our assumption that each set in K is output compatible and proves that each $c_j \in K$ is a compatible. ■

Because of Lemma 2.23, an exact state minimization algorithm only needs to generate compatibles. The next step of an exact algorithm after compatible generation is to select a subset of compatibles that corresponds to a minimized machine. To satisfy behavioral containment, the selection of compatibles should be such that appropriate covering and closure conditions are met. The covering conditions guarantee that some selected compatible (i.e., some state in the minimized machine) corresponds to a reset state of the original machine. The closure conditions require that for each selected compatible, the compatibles implied by state transitions should also be selected. The state minimization problem reduces to one that selects a minimum closed cover of compatibles. Instead of enumerating and testing all subset of compatibles, the selection is usually solved as a **binate covering problem.** Covering and closure conditions are expressed as binate clauses, and the minimum solution can be searched in a systematic manner.

The set of compatibles is still very large. For the purpose of state minimization, it is sufficient to identify a minimum subset called the prime compatibles such that a minimum closed cover of prime compatibles still yields a minimum behaviorally contained machine. Compatibles that are not *dominated* by other compatibles are called prime compatibles.

Definition 2.37 *A compatible c'* **prime dominates** *a compatible c if for each minimum closed cover containing c, the selection with c replaced by c' also corresponds to a minimum closed cover.*

Definition 2.38 *A compatible c is a* **prime compatible** *if there does not exists another compatible c' such that c' prime dominates c.*

Theorem 2.24 *There exists a minimum closed cover made up entirely of prime compatibles.*

Proof: Suppose C is a minimum closed cover which contains a compatible c that is not prime, i.e., there exists a prime compatible $\hat{c}$ which dominates c. By Definition 2.38, the set $(C - \{c\}) \cup \{\hat{c}\}$ must be another minimum closed cover with one more prime compatible. One can continue this substitution process until the minimum closed cover is made up entirely of prime compatibles. ■

The actual computations of compatibles and primeness differ for different types of FSM's. Also the related covering problems vary slightly.

Chapter 3

Implicit Techniques

3.1 Introduction

In this chapter, we shall review and introduce techniques for efficient representation and manipulation of objects. We say that a representation is *explicit* if the objects it represents are listed one by one internally. And objects are manipulated explicitly, if they are processed one after another. *Implicit* representation means a shared representation of the objects, such that the size of the representation is not proportional to the number of objects in it. By implicit manipulation, we mean that in one step, many objects are processed simultaneously.

The objects we need to represent include functions, relations, sets, and sets of sets. In Section 3.2, we introduce a new data structure called the multi-valued decision diagram (MDD) which can represent multi-valued input multi-valued output functions. MDD is an generalization of binary decision diagram (BDD) [11] which will be reviewed in Section 3.3, and the latter represents binary input binary output functions. Relations and sets can be expressed in terms of characteristic functions [13] as shown in Section 3.3. We also describe a variant of BDD called the zero-suppressed BDD (ZBDD) in Section 3.4. The remaining of the chapter concerns with efficient representation of multi-valued input binary output functions. In such a case, Section 3.5 shows that an MDD can be mapped to an equivalent BDD after choosing an encoding for each multi-valued variable. In Section 3.6, the logarithmic encoded MDD is described. And in Section 3.7, the 1-hot encoded MDD is introduced. The latter is particular useful for efficient representation of set of sets, so an extensive suite of operators will be introduced for efficient manipulations of such sets of sets. In Section 3.8, MDD's are used to represent FSM's. Issues on MDD variable ordering will be discussed in Section 3.9.

First we give definitions pertaining to MDD's.

Definition 3.1 *Let $\mathcal{F}$ be a multiple-valued input, multiple-valued output function of n variables - $x_1, x_2, \ldots, x_n$.*

$$\mathcal{F} : P_1 \times P_2 \times \ldots \times P_n \rightarrow Y$$

Each variable, x_i, may take any one of the p_i values from a finite set $P_i = \{0, 1, \ldots, p_i - 1\}$. The output of $\mathcal{F}$ may take m values from the set $Y = \{0, 1, \ldots, m-1\}$. Without loss of generality, we may assume that the domain and range of $\mathcal{F}$ are integers. In particular, $\mathcal{F}$ is a binary-valued output function if $m = 2$, and $\mathcal{F}$ is a binary-valued input function if $p_i = 2$ for every $i : 1 \leq i \leq n$.

Definition 3.2 *Let T_i be a subset of P_i. $x_i^{T_i}$ represents a* **literal** *of variable x_i which is defined as the Boolean function:*

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

Definition 3.3 *The* **cofactor** *of $\mathcal{F}$ with respect to a variable x_i taking a constant* **value** *j is denoted by $\mathcal{F}_{x_i^j}$ and is the function resulting when x_i is replaced by j:*

$$\mathcal{F}_{x_i^j}(x_1, \ldots, x_n) = \mathcal{F}(x_1, \ldots, x_{i-1}, j, x_{i+1}, \ldots, x_n)$$

Definition 3.4 *The* **cofactor** *of $\mathcal{F}$ with respect to a* **literal** *$x_i^{T_i}$ is denoted by $\mathcal{F}_{x_i^{T_i}}$ and is the union of the cofactors of $\mathcal{F}$ with respect to each value the literal represents:*

$$\mathcal{F}_{x_i^{T_i}} = \bigcup_{j \in T_i} \mathcal{F}_{x^j}$$

The cofactor of $\mathcal{F}$ is a simpler function than $\mathcal{F}$ itself because the cofactor no longer depends on the variable x_i.

Definition 3.5 *The* **Shannon decomposition** *of a function $\mathcal{F}$ with respect to a variable x_i is:*

$$\mathcal{F} = \sum_{j=0}^{p_i - 1} x_i^j \cdot \mathcal{F}_{x_i^j}$$

The Shannon decomposition expresses function $\mathcal{F}$ as a sum of simpler functions, i.e. its cofactors $\mathcal{F}_{x_i^j}$. This allows us to construct a function by recursive decomposition.

3.2 Multi-valued Decision Diagrams

This section describes a new data structure - the **multi-valued decision diagram** [78, 36] that is used to solve discrete variable problems [36, 45, 84, 44, 3]. Our definition of multi-valued decision diagrams closely follows that of Bryant, [11], with two exceptions: we do not restrict ourselves to the Boolean domain, and the range of our functions is multi-valued. From Section 3.3 onwards, we use only binary-valued output functions, however the theory is valid for the more general multi-valued functions.

Definition 3.6 *An* **multi-valued decision diagram** *(MDD) is a rooted, directed acyclic graph. Each nonterminal vertex v is labeled by a multi-valued variable $var(v)$ which can take values from a range $range(v)$. Vertex v has arcs directed towards $|range(v)|$ children vertices, denoted by $child_k(v)$ for each $k \in range(v)$. Each terminal vertex u is labeled a value $value(u) \in Y = \{0, 1, \ldots, m-1\}$.*

Each vertex in an MDD represents a multi-valued input multi-valued output function and all used-visible vertices are roots. The terminal vertex u represent the constant (function) $value(u)$. For each nonterminal vertex v representing a function F, its child vertex $child_k(v)$ represents the function F_{v^k} for each $k \in range(v)$. i.e., $F = \sum_{k \in range(v)} v^k \cdot F_{v^k}$.

For a given assignment to the variables, the value yielded by the function is determined by tracing a decision path from the root to a terminal vertex, following the branches indicated by the values assigned to the variables. The function value is then given by the terminal vertex label.

Example The MDD in Figure 3.1 represents the discrete function $F = max(0, x - y)$ where x and y are 3-valued variables.

3.2.1 Reduced Ordered MDD's

Definition 3.7 *An MDD is* **ordered** *if there is a total order $\prec$ over the set of variables such that for every nonterminal vertex v, $var(v) \prec var(child_k(v))$ if $child_k(v)$ is also nonterminal.*

Definition 3.8 *An MDD is* **reduced** *if*

1. *it contains no vertex v such that all outgoing arcs from v point to a same vertex, and*
2. *it does not contain two distinct vertices v and v' such that the subgraphs rooted at v and v' are isomorphic.*

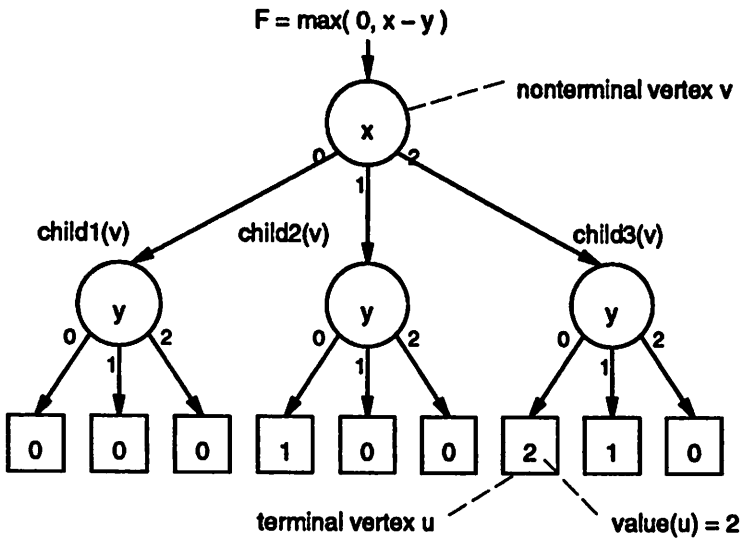

Figure 3.1: Example of an MDD for a discrete function.

Definition 3.9 *A* **reduced ordered multi-valued decision diagram** *(ROMDD) is an MDD which is both reduced and ordered.*

Henceforth, we consider only ROMDD's and the name MDD will be used to mean ROMDD.

Variable ordering must be decided before the construction of any MDD. We assume that this has been decided and that the naming of input variables have been permuted so that $x_i \prec x_{i+1}$. MDD's are guaranteed to be reduced at any time during the constructions and operations on MDD's. Each operation returns a resultant MDD in a reduced ordered form.

Example The ROMDD for the MDD in Figure 3.1 is shown in Figure 3.2. The variable ordering is $x \prec y$. Note that one redundant nonterminal vertex and six terminal vertices have been eliminated.

A very desirable property of an ROMDD is that it is a canonical representation.

Theorem 3.1 *For any multi-valued function $\mathcal{F}$, there is a unique reduced ordered (up to isomorphism) MDD denoting $\mathcal{F}$. Any other MDD denoting $\mathcal{F}$ contains more vertices.*

Proof: The complete proof has been given in [78]. ■

Corollary 3.2 *Two functions are equivalent if and only if the ROMDD's for each function are isomorphic.*

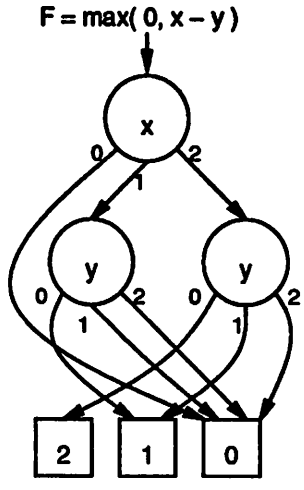

Figure 3.2: Reduced ordered MDD for the same function.

3.2.2 CASE Operator

The CASE operator forms the basis for manipulating MDD's. Most operations on discrete functions can be expressed in terms of the CASE operator on MDD's.

Definition 3.10 *The CASE operator selects and returns a function G_i according to the value of the function F:*

$$CASE(F, G_0, G_1, \ldots, G_{m-1}) = G_i \; if \; (F = i)$$

The operator is defined only if range(F) = {0, 1, ..., m-1}. The function returned from the CASE operation has a range of range(G_i). In particular, if the G_i are binary-valued, the resultant function will also be a binary-valued output function.

The input parameters to the CASE operator are, in general, multi-valued functions given in the form of MDD's. The task is to generate the resultant function $H = CASE(F, G_0, G_1, \ldots, G_{m-1})$. Since the selector F can be a function instead of a variable, we need a recursive algorithm to compute the CASE operator.

If the selector is a variable x, the following function returned by the CASE operator corresponds to a vertex with a top variable x and with child functions $G_0, G_1, \ldots, G_{p_i-1}$. The vertex is denoted by a $(p_i + 1)$-tuples on the right.

$$CASE(x, G_0, G_1, \ldots, G_{p_i-1}) = (x, G_0, G_1, \ldots, G_{p_i-1}) \tag{3.1}$$

Equation 3.1 and 3.2 will form the terminal cases for our recursive algorithm.

$$CASE(F, 0, 1, \ldots, m-1) = F \tag{3.2}$$

Notice that the Shannon decomposition of H with respect to x can be realized by:

$$\begin{aligned} H &= \sum_{j=0}^{p-1} x^j \cdot H_{x^j} \\ &= CASE(x, H_{x^0}, H_{x^1}, \ldots, H_{x^{p-1}}) \\ &= (x, H_{x^0}, H_{x^1}, \ldots, H_{x^{p-1}}) \end{aligned} \tag{3.3}$$

Recursion is based on the following reasoning. Remember that we can express a complex function in terms of its cofactors using Shannon decomposition. The cofactors of a function are simpler to compute than the original function. So to compute the CASE of complex functions, we first compute the CASE of their cofactors and then compose them together using Shannon decomposition. More rigorously,

$$\begin{aligned} CASE(F, G_0, G_1, \ldots, G_{m-1}) &= \sum_{i=0}^{p-1} x^i \cdot CASE(F, G_0, G_1, \ldots, G_{m-1})_{x^i} \\ &= \sum_{i=0}^{p-1} x^i \cdot (G_j \; if \; (F = j))_{x^i} \\ &= \sum_{i=0}^{p-1} x^i \cdot (G_{j\,x^i} \; if \; (F = j)_{x^i}) \\ &= \sum_{i=0}^{p-1} x^i \cdot (G_{j\,x^i} \; if \; (F_{x^i} = j)) \\ &= CASE(x, \\ &\quad CASE(F_{x^0}, G_{0\,x^0}, G_{1\,x^0}, \ldots, G_{m-1\,x^0}), \\ &\quad CASE(F_{x^1}, G_{0\,x^1}, G_{1\,x^1}, \ldots, G_{m-1\,x^1}), \\ &\quad \ldots, \\ &\quad CASE(F_{x^{p-1}}, G_{0\,x^{p-1}}, G_{1\,x^{p-1}}, \ldots, G_{m-1\,x^{p-1}})) \end{aligned} \tag{3.4}$$

The pseudo-code for the recursive $CASE$ algorithm is given in Figure 3.3. First, the algorithm checks for terminal cases. Then if the function needed has already been computed and stored in the unique table, it will be returned. It not, the cofactors H_{x^j} of the function H are computed by calling $CASE$ recursively with the cofactors $F_{x^j}, G_{0\,x^j}, \ldots, G_{m-1\,x^j}$ as its arguments. These

```
CASE(F, G_0, ..., G_{m-1}) {
    if terminal case return result
    if CASE(F, G_0, ..., G_{m-1}) in computed-table return result
    let x be the top-variable of F, G_0, ..., G_{m-1}
    let p be the number of values x takes
    for j = 0 to (p - 1) do
        H_{xj} = CASE(F_{xj}, G_{0 xj}, ..., G_{m-1 xj});
    result = (x, H_{x^0}, ..., H_{x^{p-1}})
    insert result in computed-table for CASE(F, G_0, ..., G_{m-1})
    return result
}
```

Figure 3.3: Pseudo-code for the $CASE$ algorithm.

are composed together using Shannon decomposition. By Equation 3.3, Shannon decomposition with respect to x is equivalent to the $(p+1)$-tuple $(x, H_{x^0}, \ldots, H_{x^{p-1}})$.

It is shown in [78] that the worst-case time complexity of the $CASE$ algorithm is $O(p_{max} \cdot \mid F \mid \cdot \mid G_0 \mid \ldots \mid G_{m-1} \mid)$.

3.3 Binary Decision Diagrams

The literal x_i denotes that variable x_i has the value 1 and the literal $\overline{x_i}$ denotes that variable x_i has the value 0. Cofactors with respect to literals are similar to the ones in the previous section, and are formally defined in Section 3.3.1.

Binary decision diagrams were first proposed by Akers in [1] and popularized by Bryant in [11].

Definition 3.11 *A* **binary decision diagram** *(BDD) is a rooted, directed acyclic graph. Each nonterminal vertex v is labeled by a Boolean variable $var(v)$. Vertex v has two outgoing arcs, $child_0(v)$ and $child_1(v)$. Each terminal vertex u is labeled 0 or 1.*

Each vertex in a BDD represents a binary input binary output function and all used-visible vertices

are roots. The terminal vertices represent the constants (functions) 0 and 1. For each nonterminal vertex v representing a function F, its child vertex $child_0(v)$ represents the function $F_{\overline{v}}$ and its other child vertex $child_1(v)$ represents the function F_v. i.e., $F = \overline{v} \cdot F_{\overline{v}} + v \cdot F_v$.

For a given assignment to the variables, the value yielded by the function is determined by tracing a decision path from the root to a terminal vertex, following the branches indicated by the values assigned to the variables. The function value is then given by the terminal vertex label.

Definition 3.12 *A BDD is* **ordered** *if there is a total order $\prec$ over the set of variables such that for every nonterminal vertex v, $var(v) \prec var(child_0(v))$ if $child_0(v)$ is nonterminal, and $var(v) \prec var(child_1(v))$ if $child_1(v)$ is nonterminal.*

Definition 3.13 *A BDD is* **reduced** *if*

1. *it contains no vertex v such that $child_0(v) = child_1(v)$, and*
2. *it does not contain two distinct vertices v and v' such that the subgraphs rooted at v and v' are isomorphic.*

Definition 3.14 *A* **reduced ordered binary decision diagram** *(ROBDD) is a BDD which is both reduced and ordered.*

Any **subset** S in a Boolean space B^n can be represented by a unique Boolean function $\chi_S : B^n \rightarrow B$, which is called its **characteristic function** [13], such that:

$$\chi_S(x) = 1 \text{ if and only if } x \text{ in } S$$

In the sequel, we'll not distinguish the subset S from its characteristic function χ_S, and will use S to denote both.

Any **relation** $\mathcal{R}$ between a pair of Boolean variables can also be represented by a characteristic function $\mathcal{R} : B^2 \rightarrow B$ as:

$$\mathcal{R}(x, y) = 1 \text{ if and only if } x \text{ is in relation } \mathcal{R} \text{ to } y$$

$\mathcal{R}$ can be a one-to-many relation over the two sets in B.

These definitions can be extended to any relation $\mathcal{R}$ between n Boolean variables, and can be represented by a characteristic function $\mathcal{R} : B^n \rightarrow B$ as:

$$\mathcal{R}(x_1, x_2, \ldots, x_n) = 1 \text{ if and only if the } n\text{-tuple } (x_1, x_2, \ldots, x_n) \text{ is in relation } \mathcal{R}$$

3.3.1 BDD Operators

A rich set of BDD operators has been developed and published in the literature [11, 6]. The following is the subset of operators useful in our work.

The ITE operator forms the basis for the construction and manipulation of BDD's. The use of the ITE operator also guarantees that the resulting BDD is in strong canonical form [6]. The CASE operator is the multi-valued analog of the ITE operator.

Definition 3.15 *The ITE operator returns function G_1 if function F evaluates true, else it returns function G_2:*

$$ITE(F, G_1, G_0) = \begin{cases} G_1 & \text{if } F = 1 \\ G_0 & \text{otherwise} \end{cases}$$

where range(F)={0,1}.

Definition 3.16 *The* **substitution** *in the function $\mathcal{F}$ of variable x_i with variable y_i is denoted by:*

$$[x_i \rightarrow y_i]\mathcal{F} = \mathcal{F}(x_1, \ldots, x_{i-1}, y_i, x_{i+1}, \ldots, x_n)$$

and the substitution in the function $\mathcal{F}$ of a set of variables $x = x_1 x_2 \ldots x_n$ with another set of variables $y = y_1 y_2 \ldots y_n$ is obtained simply by:

$$[x \rightarrow y]\mathcal{F} = [x_1 \rightarrow y_1][x_2 \rightarrow y_2] \ldots [x_n \rightarrow y_n]\mathcal{F}$$

In the description of subsequent computations, some obvious substitutions will be omitted for clarity in formulas.

Definition 3.17 *The* **cofactor** *of $\mathcal{F}$ with respect to the literal x_i ($\overline{x_i}$ resp.) is denoted by $\mathcal{F}_{x_i}$ ($\mathcal{F}_{\overline{x_i}}$ resp.) and is the function resulting when x_i is replaced by 1 (0 resp.):*

$$\mathcal{F}_{x_i}(x_1, \ldots, x_n) = \mathcal{F}(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$$
$$\mathcal{F}_{\overline{x_i}}(x_1, \ldots, x_n) = \mathcal{F}(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$$

The cofactor of $\mathcal{F}$ is a simpler function than $\mathcal{F}$ itself because the cofactor no longer depends on the variable x_i.

Definition 3.18 *The* **existential quantification** *(also called* **smoothing***) of a function $\mathcal{F}$ over a variable x_i is denoted by $\exists x_i(\mathcal{F})$ and is defined as:*

$$\exists x_i\,(\mathcal{F}) = \mathcal{F}_{\overline{x_i}} + \mathcal{F}_{x_i}$$

and the existential quantification over a set of variables $x = x_1, x_2, \ldots, x_n$ *is defined as:*

$$\exists x\ (\mathcal{F}) = \exists x_1\ (\exists x_2\ (\ldots(\exists x_n\ (\mathcal{F}))))$$

Definition 3.19 *The* **universal quantification** *(also called* **consensus***) of a function* $\mathcal{F}$ *over a variable* x_i *is denoted by* $\forall x_i(\mathcal{F})$ *and is defined as:*

$$\forall x_i\ (\mathcal{F}) = \mathcal{F}_{\overline{x_i}} \cdot \mathcal{F}_{x_i}$$

and the universal quantification over a set of variables $x = x_1, x_2, \ldots, x_n$ *is defined as:*

$$\forall x\ (\mathcal{F}) = \forall x_1\ (\forall x_2\ (\ldots(\forall x_n\ (\mathcal{F}))))$$

3.3.2 Unique Quantifier

Now we introduce a new BDD operator called the unique quantifier, which is in the same class as the existential and universal quantifiers.

Definition 3.20 *The* **unique quantification** *of a function* $\mathcal{F}$ *over a variable* x_i *is denoted by* $!x_i\ (\mathcal{F})$ *and is defined as:*

$$!x_i\ (\mathcal{F}) = \mathcal{F}_{\overline{x_i}} \oplus \mathcal{F}_{x_i}$$

and the unique quantification over a set of variables $x = x_1, x_2, \ldots, x_n$ *is defined as:*

$$!x\ (\mathcal{F}) = !x_1\ (!x_2\ (\ldots(!x_n\ (\mathcal{F}))))$$

Suppose $\mathcal{F}$ is a relation on x, y and z. $!x\ \mathcal{F}(x, y, z) = \{(y, z) | \text{a pair } (y, z) \text{ is related to a unique } x\}$.

Some properties of the unique quantifier will be presented:

Lemma 3.3 $!x\ !y\ F \Leftrightarrow !y\ !x\ F$

Proof: $!x\ !y\ F \Leftrightarrow (F_{\overline{x}\,\overline{y}} \oplus F_{\overline{x}y} \oplus F_{x\overline{y}} \oplus F_{xy}) \Leftrightarrow !y\ !x\ F$
using the distributive property of cofactor over XOR, $(F \oplus G)_x \Leftrightarrow F_x \oplus G_x$. ■

It is well known that $\exists x\ \forall y\ F \Rightarrow \forall y\ \exists x\ F$. Let us investigate if similar properties hold for the unique quantifier.

Lemma 3.4 $!x\ \exists y\ F \Rightarrow \exists y\ !x\ F$

Proof:

$$
\begin{aligned}
!x\,\exists y\,F &\Leftrightarrow (F_{\overline{y}} + F_y)_{\overline{x}} \oplus (F_{\overline{y}} + F_y)_x \\
&\Leftrightarrow F_{\overline{x}\,\overline{y}} \cdot \overline{F_{x\overline{y}}} \cdot \overline{F_{xy}} + F_{\overline{x}y} \cdot \overline{F_{x\overline{y}}} \cdot \overline{F_{xy}} + F_{x\overline{y}} \cdot \overline{F_{\overline{x}\,\overline{y}}} \cdot \overline{F_{\overline{x}y}} + F_{xy} \cdot \overline{F_{\overline{x}\,\overline{y}}} \cdot \overline{F_{\overline{x}y}} \\
\exists y\,!x\,F &\Leftrightarrow (F_{\overline{x}} \oplus F_x)_{\overline{y}} + (F_{\overline{x}} \oplus F_x)_y \\
&\Leftrightarrow F_{\overline{x}\,\overline{y}} \cdot \overline{F_{x\overline{y}}} + \overline{F_{\overline{x}\,\overline{y}}} \cdot F_{x\overline{y}} + F_{\overline{x}y} \cdot \overline{F_{xy}} + \overline{F_{\overline{x}y}} \cdot F_{xy} \\
!x\,\exists y\,F &\Rightarrow \exists y\,!x\,F
\end{aligned}
$$

■

The converse, $\exists y\,!x\,F \Rightarrow !x\,\exists y\,F$, is not true in general. Consider the relation $F(x,y) = \{(0,0),(0,1),(1,1)\}$, $\exists y\,!x\,F$ is true but $!x\,\exists y\,F$ is false. For the same F, $!x\,\forall y\,F$ is true but $\forall y\,!x\,F$ is not; therefore $!x\,\forall y\,F \Rightarrow \forall y\,!x\,F$ is not true in general. Also, $!x\,\forall y\,F \Leftarrow \forall y\,!x\,F$ is not true in general because of the counter example $F(x,y) = \{(0,0),(1,1)\}$, where $\forall y\,!x\,F$ is true but $!x\,\forall y\,F$ is false.

The pseudo-code in Figure 3.4 outlines the BDD *unique* algorithm. The BDD function for $!x\,F$ is returned by calling $unique(F,x,1)$ where $x = \{x_1, x_2, \ldots, x_n\}$.

First, the algorithm checks for terminal cases, and checks if the result has already been computed before. Otherwise if the top variable v of F is the same as x_i then the following recursive formula is applied:

$$!x_i, x_{i+1}, \ldots, x_n\,(F) = !x_{i+1}, x_{i+2}, \ldots, x_n\,(F_{\overline{x_i}}) \oplus !x_{i+1}, x_{i+2}, \ldots, x_n\,(F_{x_i})$$

If v is below x_i, F is independent of variable x_i and therefore the result from the next recursion can be simply returned. If v is above x_i, we need to compute $!x\,F_{\overline{v}}$ and $!x\,F_v$ and merge the results by the ITE operator. Finally the result is stored in the computed-table, and returned.

3.4 Zero-suppressed BDD's

Definition 3.21 *A* **zero-suppressed BDD** *(ZBDD) is defined identically as a BDD.*

The functional interpretation of ZBDD's is the same as that for BDD's.

Definition 3.22 *A ZBDD is* **ordered** *if it is ordered when viewed as a BDD.*

Definition 3.23 *A ZBDD is* **reduced** *if*

```
unique(F, x, i) {
    if (i > |x|) or (top_index(F) > bottom_index(x)) return F
    if unique(F, x, i) in computed-table return result
    let v be the top variable of F
    if (index(v) = index(x_i)) {
        T = unique(F_{x_i}, x, i + 1)
        E = unique(F_{x̄_i}, x, i + 1)
        result = ITE(T, Ē, E)
    } else if (top_index(F) > index(x_i)) {
        result = unique(F, x, i + 1)
    } else {
        T = unique(F_v, x, i)
        E = unique(F_v̄, x, i)
        result = ITE(v, T, E)
    }
    insert result in computed-table for unique(F, x, i)
    return result
}
```

Figure 3.4: Pseudo-code for the *unique* quantifier.

1. *it contains no vertex v such that $child_1(v)$ is a terminal vertex labeled 0, and*

2. *it does not contain two distinct vertices v and v' such that the subgraphs rooted at v and v' are isomorphic.*

Definition 3.24 *A* **reduced ordered zero-suppressed binary decision diagram** *(ROZBDD) is a ZBDD which is both reduced and ordered.*

The difference between ROBDD and ROZBDD is in one reduction rule. ROBDD eliminates all vertex whose two outgoing arc points to a same vertex. ROZBDD eliminates all vertex whose 1-edge points to a terminal vertex 0. Once a redundant vertex is removed from a ZBDD, the incoming edges of the vertex is directly connected to the vertex pointed to by the corresponding terminal vertex 0.

3.4.1 ZBDD Operators

ROZBDD is designed for representing combination sets so its operators are for set operations, which turn out to be quite different from the common BDD operations on characteristic functions. The following outlines some of the differences between BDD's and ZBDD's, and the difficulties in using the latter.

Unlike ROBDD, the ROZBDD for a universal set is not unique but it has a chain structure on the variable space of interest. A constant-time complement operation is not available in ZBDD. Complementation on a set can be performed as set difference against a universal set. But even this is not recommended because if the original set is a sparse set having a compact ROZBDD representation, its complement will be dense and cannot be represented compactly.

The semantics of BDD's and ZBDD's are quite different. BDD is based on Shannon decomposition but ZBDD is not. As a result, BDD operators such as cofactor, smooth, consensus, generalized cofactor, etc., do not have direct counterparts in ZBDD.

As a result of these differences, one who has an implicit BDD algorithm cannot readily convert it into an implicit ZBDD algorithm and expects the latter to run efficiently.

3.5 Mapping MDD's into BDD's

The first-generation MDD package is a direct implementation of the theory presented in Section 3.2. For efficiency, our current MDD package uses BDD's as its internal representation. By

hiding the mapped-BDD and its encoded variables from the users, users can construct functions, manipulate them and output results in terms of multi-valued variables only. But only Boolean output (multi-valued input) functions can be represented by mapped-BDD's.

The mapping of MDD's into BDD's involves two distinct steps: *encoding* and *variable ordering*. **Encoding** is the process of associating a number of binary-valued variables to each multi-valued variable, and assigning codes to represent the values the multi-valued variables can take. **Variable Ordering** is the process of finding an ordering of the encoded binary-valued variables such that the size of the final BDD is minimized. Section 3.9 will be devote to variable ordering techniques used in our work, while we shall first describe the encoding process here. To encode each m-valued MDD vertex, we must decide on:

1. the number of binary variables used: n encoding variables result in 2^n code points, and each code corresponds to a decision path in the full BDD subgraph of these variables.

2. the assignments of codes to values: At least one distinct code point must be assigned to each value. Therefore $2^n \geq m$ must be true.

3. the treatment of unused code points: If $2^n > m$, there is one or more unused code points. Each unused code can either be left unassigned, or be associated a value which another code has already been assigned to. For the former case, its corresponding path always points to the terminal vertex 0. For the latter case, the corresponding path can point to the same place as another path in the BDD subgraph.

In the next two sections, we shall investigate two encoding schemes. Logarithmic encoding in Section 3.6 offers compact representation of functions, e.g., characteristic functions, where as 1-hot encoding in Section 3.7 is useful for set representation. Once an encoding and an ordering are chosen, there is a unique way to map each MDD vertex into a BDD subgraph.

Example Figure 3.5a shows an MDD representing the following function:

$$F = \begin{cases} 1 & \text{if } x > y \\ 0 & \text{otherwise} \end{cases}$$

x and y are 3-valued variables which can take values from $P_x = P_y = \{0, 1, 2\}$. To represent the MDD using BDD's, each MDD nonterminal vertex must be mapped into a number of BDD vertices interconnected in a subgraph. For example in Figure 3.5, the MDD vertex labeled by variable x is mapped into the BDD vertices labeled by x_0 and x_1. In addition, different indices have to be

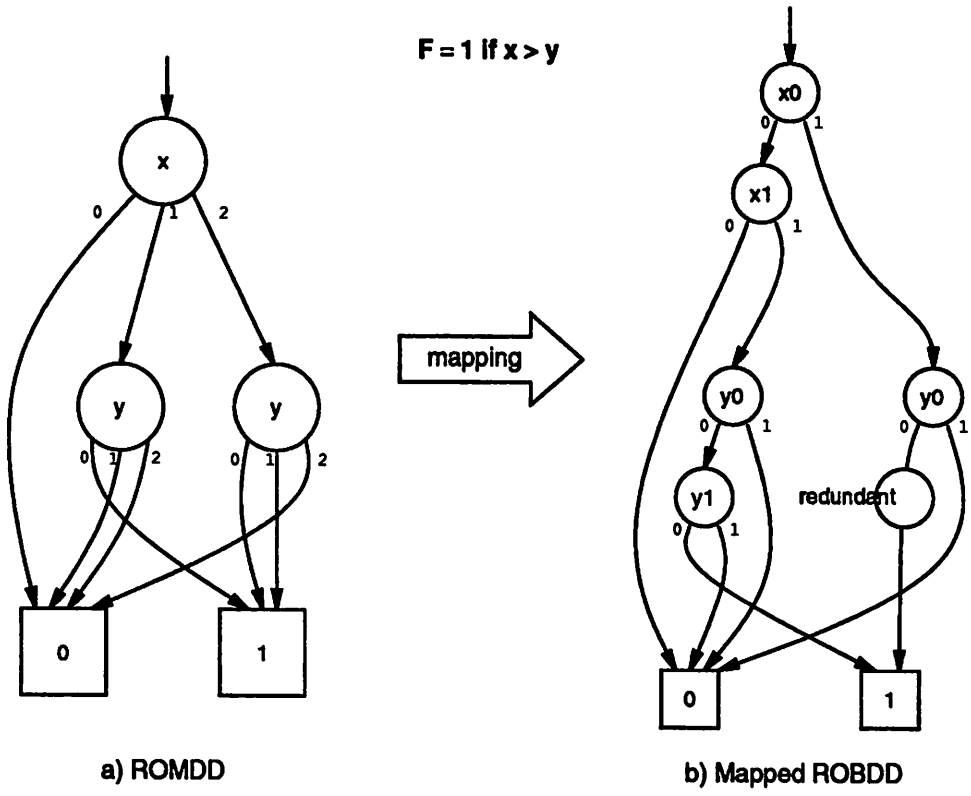

Figure 3.5: MDD and mapped-BDD representing the relation $x > y$.

assigned to these binary variables. In this case, since $x \prec y$ for the MDD, this ordering is respected for the associated binary variables; $x_0 \prec x_1 \prec y_0 \prec y_1$. The mapping process dictates the encoding used. The same encoding, as well as ordering, must be used consistently throughout all function manipulations.

3.6 Logarithmic Encoded MDD's

In this section, logarithmic encoding (i.e., integer encoding) is used to map MDD's into BDD's. Our prime concern here is to use the least number of variables and BDD vertices. An m-valued MDD vertex is represented with $\lceil log_2 m \rceil$ number of Boolean variables, and is mapped to a BDD subgraph of $m-1$ vertices. These numbers are provably minimum in graph theory.

Each value is assigned an integer code. As discrete variables in CAD problems usually take values from the ordered set of integers and the operations between them are sometimes integer-arithmetic in nature, integer encoding results in efficient representation and manipulation.

Of course, not all 2^k code points will be used since typically $m < 2^k$. On the other hand the mapped-BDD will have a path for each binary code point. The decision path for each unused code point is chosen so as to minimize the BDD size. In fact, an unused point is assigned to the same path as the used point whose encoding is closest to the unused code point. This mapping is related to the *generalized cofactor* operator in [80] which was initially proposed in [17] as the *constraint* operator. Given a function f and a care set c, the generalized cofactor of f with respect to c is the projection of f that maps a don't care point x to the care point $y \in c$ which has the closest distance to x. Generalized cofactoring results in a small and canonical BDD representation of the incompletely specified function.

Example Suppose v is a 6-valued variable taking values from $P_v = \{0,1,2,3,4,5\}$. Three binary-valued variables u_0, u_1 and u_2 can be assigned to encode variable v as shown in Table 3.1. The last column is used in the example in Section 3.6.1.

Note that if the value range is not a power of 2, some codes will not be used, e.g., 110 and 111. These encodings are used as don't cares since the values will never occur. In this case these don't care are mapped into the same nodes as 100 and 101 respectively. The notation $1*0$ is used to represent both encodings 100 and 110 as we don't care about the variable u_1.

Value of v	Binary Encoding $u_0u_1u_2$	F =
0	000	G_0
1	001	G_1
2	010	G_2
3	011	G_3
4	1*0	G_4
5	1*1	G_5

Table 3.1: Logarithmic/integer encoding with don't cares.

3.6.1 Relationships between CASE and ITE Operators

As the CASE and ITE operators form the basis for manipulation of MDD's and BDD's respectively, mapping can be conveniently performed by replacing each CASE operation by a set of ITE operations. The recursion step in Equation 3.4 is our starting point. It gives an outer CASE operator in terms of a top-variable v, and enables conversion to a hierarchy of ITE operators. The conversion can be summarized by the following recursive formulas:

if p is even: $CASE(v, \underbrace{G'_0, G'_1}, \underbrace{G'_2, G'_3}, \ldots, \underbrace{G'_{p-2}, G'_{p-1}})$

$$= CASE(v', ITE(u, G'_1, G'_0), ITE(u, G'_3, G'_2), \ldots, ITE(u, G'_{p-1}, G'_{p-2}))$$

if p is odd: $CASE(v, \underbrace{G'_0, G'_1}, \underbrace{G'_2, G'_3}, \ldots, \underbrace{G'_{p-3}, G'_{p-2}}, G'_{p-1})$

$$= CASE(v', ITE(u, G'_1, G'_0), ITE(u, G'_3, G'_2), \ldots, ITE(u, G'_{p-2}, G'_{p-3}), G'_{p-1})$$

The recursive use of the above two equations continues until this recursion terminates when there are only two child-functions remaining in the outer CASE operator:

$$CASE(v, G'_0, G'_1) = ITE(v, G'_1, G'_0).$$

While pairing up child-functions with the ITE operator, these formulas replace the big MDD vertex labeled with variable v with a smaller one, labeled with a new multi-valued variable v', and a number of BDD vertices labeled with a new binary variable u. This mapping process is best explained by an example.

Example Suppose v is a 6-valued MDD vertex, and $G'_0, \ldots, G'_5$ are the six child-functions connected

to it, the CASE to ITE mapping proceeds as follows:

$$
\begin{aligned}
& CASE(v, \underbrace{G'_0, G'_1}, \underbrace{G'_2, G'_3}, \underbrace{G'_4, G'_5}) \\
&= CASE(v', \underbrace{ITE(u_2, G'_1, G'_0), ITE(u_2, G'_3, G'_2)}, ITE(u_2, G'_5, G'_4)) \\
&= CASE(v'', \underbrace{ITE(u_1, ITE(u_2, G'_3, G'_2), ITE(u_2, G'_1, G'_0))}, ITE(u_2, G'_5, G'_4)) \\
&= ITE(u_0, ITE(u_2, G'_5, G'_4), ITE(u_1, ITE(u_2, G'_3, G'_2), ITE(u_2, G'_1, G'_0)))
\end{aligned}
$$

Note that while pairing up child-functions for ITE operations in the first step, we effectively replace the original 6-valued MDD vertex with a smaller 3-valued MDD vertex. During the assignment of BDD variables, the ordering $u_0 \prec u_1 \prec u_2$ is used. Figure 3.6 shows the bottom-up recursive mapping process. Note that the original MDD node labeled v has been mapped into a BDD subgraph with 5 internal nodes.

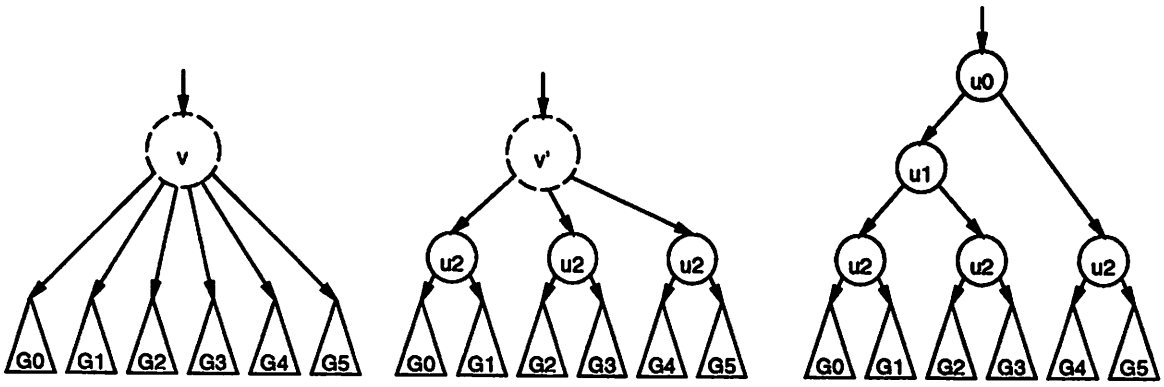

Figure 3.6: Recursive mapping from an MDD vertex to a mapped-BDD subgraph.

3.7 1-hot Encoded MDD's

In this section we describe how to represent and manipulate implicitly sets of objects. This theory is especially useful for applications where sets of sets of objects need to be constructed and manipulated, as it is often the case in logic synthesis and combinatorial optimization. In many applications such as FSM minimization, encoding and partitioning, the number of objects (number of states in these cases) to be handled is usually not large. But their exact optimization algorithms require exploration of many different subsets of such objects. As shown in Section 2.7, exact state minimization requires selection of a minimum closed cover out of a huge number of candidate sets of state sets. Therefore our prime concern here is to have a compact representation for set of sets.

Suppose the elements corresponds to n distinct objects. With 1-hot encoding, a Boolean variable is associated with each object so n Boolean variables are used. Each singleton element is assigned a distinct 1-hot code. Obviously with this 1-hot encoding scheme, there are a lot of unused code points. Unlike the logarithmic encoding where unused code points are reassigned to values, these code points are used for a purpose other than representing elements or values, but to represent sets other than singletons.

3.7.1 Positional-set Notation

Given that there are 2^n possible distinct sets of objects, in order to represent collections of them it is not possible to encode the objects using $log_2 n$ Boolean variables. Instead, each subset of objects is represented in **positional-set** or positional-cube notation form, using a set of n Boolean variables, $x = x_1 x_2 \ldots x_n$. The presence of an element s_k in the set is denoted by the fact that variable x_k takes the value 1 in the positional-set, whereas x_k takes the value 0 if element s_k is not a member of the set. One Boolean variable is needed for each element because the element can either be present or absent in the set [1].

In the above example, $n = 6$, and the set with a single element s_4 is represented by 000100 while the set $s_2 s_3 s_5$ is represented by 011010. The elements s_1, s_4, s_6 which are not present correspond to 0's in the positional-set.

A set of sets of objects is represented as a set S of positional-sets, by a characteristic function $\chi_S : B^n \rightarrow B$ as:

$\chi_S(x) = 1$ if and only if the set represented by the positional-set x is in the set S of sets.

A 1-hot encoded MDD representing $\chi_S(x)$ will contain minterms, each corresponding to a set in S. Operators for manipulating positional-sets and characteristic functions will be introduced in the next two subsections.

A 1-hot encoded MDD can be represented as a BDD where each Boolean variable corresponds to a BDD variable. From now on, we use BDD to refer to 1-hot encoded MDD where there is no ambiguity.

[1] The representation of primes proposed by Coudert *et al.* [19] needs 3 values per variable to distinguish if the present literal is in positive or negative phase or in both phases.

3.7.2 Operations on Positional-sets

With our previous definitions of relations and positional-set notation for representing set of objects, useful relational operators on sets can be derived. We propose a unified notational framework for set manipulation which extends the notation used in [48]. In this section, each operators Op acts on two sets of variables $x = x_1x_2 \ldots x_n$ and $y = y_1y_2 \ldots y_n$ and returns a relation $(x\ Op\ y)$ (as a characteristic function) of pairs of positional-sets. Alternatively, they can also be viewed as constraints imposed on the possible pairs out of two sets of objects, x and y. For example, given two sets of sets X and Y, the set pairs (x, y) where x contains y are given by the product of X and Y and the containment constraint, $X(x) \cdot Y(y) \cdot (x \supseteq y)$.

Lemma 3.5 *The* **equality relation** *evaluates true if the two sets of objects represented by positional-sets x and y are identical, and can be computed as:*

$$(x = y) = \prod_{k=1}^{n} x_k \Leftrightarrow y_k$$

where $x_k \Leftrightarrow y_k = x_k \cdot y_k + \neg x_k \cdot \neg y_k$ designates the Boolean XNOR *operation and $\neg$ designates the Boolean* NOT *operation.*

Proof: $\prod_{k=1}^{n} x_k \Leftrightarrow y_k$ requires that for every element k, either both positional-sets x and y contain it, or it is absent from both. Therefore, x and y contains exactly the same set of elements and thus are equal. ■

Lemma 3.6 *The* **containment relation** *evaluates true if the set of objects represented by x contains the set of objects represented by y, and can be computed as:*

$$(x \supseteq y) = \prod_{k=1}^{n} y_k \Rightarrow x_k$$

where $x_k \Rightarrow y_k = \neg x_k + y_k$ designates the Boolean implication operation.

Proof: $\prod_{k=1}^{n} y_k \Rightarrow x_k$ requires that for all object, if an object k is present in y (i.e., $y_k = 1$), it must also be present in x ($x_k = 1$). Therefore set x contains all the objects in y. ■

Lemma 3.7 *The* **strict containment relation** *evaluates true if the set of objects represented by x strictly contains the set of objects represented by y, and can be computed as:*

$$(x \supset y) = (x \supseteq y) \cdot \neg(x = y) \tag{3.5}$$

Alternatively, $(x \supset y)$ *can be computed by:*

$$(x \supset y) = \prod_{k=1}^{n} [y_k \Rightarrow x_k] \cdot \sum_{k=1}^{n} [x_k \cdot \neg y_k] \tag{3.6}$$

Proof: Equation 3.5 follows directly from the two previous theorems. For Equation 3.6, the first term is simply the containment constraint, while the second term $\sum_{k=1}^{n} [x_k \cdot \neg y_k]$ requires that for at least one object k, it is present in x ($x_k = 1$) but is absent from y ($y_k = 0$), i.e., x and y are not the same. So it is an alternative way of computing $(x \supset y)$. ■

Lemma 3.8 *The* **equal-union relation** *evaluates true if the set of objects represented by* x *is the union of the two sets of objects represented by* y *and* z*, and can be computed as:*

$$(x = y \cup z) = \prod_{k=1}^{n} x_k \Leftrightarrow (y_k + z_k)$$

Proof: For each position k, x_k is set to the value of the OR between x_k and y_k. Effectively, $\prod_{k=1}^{n} x_k \Leftrightarrow (y_k + z_k)$ performs a bitwise OR on y and z to form a single positional-set z, which represents the union of the two individual sets. ■

Lemma 3.9 *The* **equal-intersection relation** *evaluates true if the set of objects represented by* x *is the intersection of the two sets of objects represented by* y *and* z*, and can be computed as:*

$$(x = y \cap z) = \prod_{k=1}^{n} x_k \Leftrightarrow (y_k \cdot z_k)$$

Proof: For each position k, x_k is set to the value of the AND between x_k and y_k. Effectively, $\prod_{k=1}^{n} x_k \Leftrightarrow (y_k \cdot z_k)$ performs a bitwise AND on y and z to form a single positional-set x, which represents the intersection of the two individual sets. ■

Lemma 3.10 *The* **contain-union relation** *evaluates true if the set of objects represented by* x *contains the union of the two sets of objects represented by* y *and* z*, and can be computed as:*

$$(x \supseteq y \cup z) = \prod_{k=1}^{n} (y_k + z_k) \Rightarrow x_k$$

Proof: Note the similarity in the computations of $(x \supseteq y \cup z)$ and $(x = y \cup z)$. $(x \supseteq y \cup z)$ performs bitwise OR on singletons y and z. If either of their k-bit is 1, the corresponding x_k bit is constrained to 1. Otherwise, x_k can take any values (i.e., don't care). The outer product $\prod_{k=1}^{n}$ requires that the above is true for each k. ■

Lemma 3.11 *The* **contain-intersection relation** *evaluates true if the set of objects represented by x contains the intersection of the two sets of objects represented by y and z, and can be computed as:*

$$(x \supseteq y \cap z) = \prod_{k=1}^{n} (y_k \cdot z_k) \Rightarrow x_k$$

Proof: Note the similarity in the computations of $(x \supseteq y \cap z)$ and $(x = y \cap z)$. $(x \supseteq y \cap z)$ performs bitwise AND on singletons y and z. If either of their k-bit is 1, the corresponding x_k bit is constrained to 1. Otherwise, x_k can take any values (i.e., don't care). The outer product $\prod_{k=1}^{n}$ requires that the above is true for each k. ■

3.7.3 Operations on Sets of Positional-sets

The first three lemmas in this section introduces operators that return a set of positional-sets as the result of some implicit set operations on one or two sets of positional-sets.

Lemma 3.12 *Given the characteristic functions χ_A and χ_B representing the sets A and B, set operations on them such as the* **union, intersection, sharp,** *and* **complementation** *can be performed as logical operations on their characteristic functions, as follows:*

$$\begin{aligned} \chi_{A \cup B} &= \chi_A + \chi_B \\ \chi_{A \cap B} &= \chi_A \cdot \chi_B \\ \chi_{A-B} &= \chi_A \cdot \neg \chi_B \\ \chi_{\overline{A}} &= \neg \chi_A \end{aligned}$$

Lemma 3.13 *The* **maximal** *of a set χ of subsets is the set containing subsets in χ not strictly contained by any other subset in χ, and can be computed as:*

$$Maximal_x(\chi) = \chi(x) \cdot \nexists y\, [(y \supset x) \cdot \chi(y)]$$

Proof: The term $\exists y\, [(y \supset x) \cdot \chi(y)]$ is true if and only if there is a positional-set y in χ such that $x \subset y$. In such a case, x cannot be in the maximal set by definition, and can be subtracted out. What remains is exactly the maximal set of subsets in $\chi(x)$. ■

Lemma 3.14 *Given a set of positional-sets $\chi(x)$ and an array of the Boolean variables x, the maximal of positional-sets in χ with respect to x can be computed by the recursive BDD operator $Maximal(\chi, 0, x)$:*

```
Maximal(χ, k, x) {
    if (χ = 0) return 0
    if (χ = 1) return ∏_{i=k}^{n} x_i
    M_0 = Maximal(χ_{x̄_k}, k + 1)
    M_1 = Maximal(χ_{x_k}, k + 1)
    return ITE(x_k, M_1, M_0 · ¬M_1)
}
```

Proof: The operator starts at the top of the BDD and recurses down until a terminal node is reached. At each recursive call, the operator returns the maximal set of positional-sets within χ making up of elements from k to n. If terminal **0** is reached, there is no positional-set within χ so **0** (i.e., nothing) is returned. If terminal **1** is reached, χ contains all possible position-sets with elements from k to n, and the maximum one is $\prod_{i=k}^{n} x_i$. At any intermediate BDD node, we find the maximal positional-sets M_0 on the *else* branch of χ, the maximal positional-sets M_1 on the *then* branch of χ. The resultant maximal set of sets contains (1) positional-sets in M_1 each with element x_k added to it as they cannot be contained by any set in M_1 which has $x_k = 0$, and (2) positional-sets that are in M_0 but not in M_1 because if a set is present in both, it is already accounted for in (1). Thus the ITE operation returns the required maximal set after each call. ■

To guarantee that each node of the BDD χ is processed exactly once, intermediate results should be cached by a computed-table.

Lemma 3.15 *The* **minimal** *of a set χ of subsets is the set containing subsets in χ not strictly containing any other subset in χ, and can be computed as:*

$$Minimal_x(\chi) = \chi(x) \cdot \nexists y\,[(x \supset y) \cdot \chi(y)]$$

Proof: The term $\exists y\,[(x \supset y) \cdot \chi(y)]$ is true if and only if there is a positional-set y in χ such that $x \supset y$. In such a case, x cannot be in the minimal set by definition, and can be subtracted out. What remains is exactly the minimal set of subsets in $\chi(x)$. ■

A recursive BDD operator $Minimal(\chi, k, x)$ can be similarly defined.

The next three operators check set equality, containment and strict containment between two sets of sets, whereas Lemmas 3.5, 3.6 and 3.7 check on a pair of sets only. These following operators returns tautology if the tests are passed.

Lemma 3.16 *Given the characteristic functions $\chi_A(x)$ and $\chi_B(x)$ representing two sets A and B (of positional-sets), the* **set equality test** *is true if and only if sets A and B are identical, and can be computed by:*

$$Equal_x(\chi_A, \chi_B) = \forall x\ [\chi_A(x) \Leftrightarrow \chi_B(x)]$$

Alternatively, Equal can be found by checking if their corresponding ROBDD's are the same by $bdd\_equal(\chi_A, \chi_B)$.

Proof: $\chi_A(x)$ and $\chi_B(x)$ represents the same set if and only if for every x, either $x \in A$ and $x \in B$, or $x \notin A$ and $x \notin B$. As the characteristic function representing a set in positional-set notation is unique, two characteristic functions will represent the same set if and only if their ROBDD's are the same. ■

Lemma 3.17 *Given the characteristic functions $\chi_A(x)$ and $\chi_B(x)$ representing two sets A and B (of positional-sets), the* **set containment test** *is true if and only if set A contains set B, and can be computed by:*

$$Contain_x(\chi_A, \chi_B) = \forall x\ [\chi_B(x) \Rightarrow \chi_A(x)]$$

Lemma 3.18 *Given the characteristic functions χ_A and χ_B representing two sets A and B (of positional-sets), the* **set strict containment test** *is true if and only if set A strictly contains set B, and can be computed by:*

$$Strict\_Contain_x(\chi_A, \chi_B) = Contain_x(\chi_A, \chi_B) \cdot \neg Equal_x(\chi_A, \chi_B)$$

Proof: The proof follows directly from previous two theorems. ■

Beside operating on sets of sets, the above operators can also be used on relations of sets. The effect is best illustrated by an example. Suppose A and B are binary relations on sets. $Contain_x(\chi_A(x, y), \chi_B(x, z))$ will return another relation on pairs (y, z) of sets. Position sets y and z are in the resultant relation if and only if the set of positional-sets x associated with y in relation A *contains* the set of positional-sets x associated with z in B.

The remaining operators in this section take a set of sets and a set of variables as parameters, and returns a singleton positional-set on those variables.

Lemma 3.19 *Given a characteristic function $\chi_A(x)$ representing a set A of positional-sets, the* **set union relation** *tests if positional-set y represents the union of all sets in A, and can be computed by:*

$$Union_{x \to y}(\chi_A) = \prod_{k=1}^{n} y_k \Leftrightarrow \exists x\ [\chi_A(x) \cdot x_k]$$

Proof: For each position k, the right hand expression sets y_k to 1 if and only if there exists an x in χ_A such that its k-th bit is a 1 ($\exists x\ [\chi_A(x) \cdot x_k]$). This implies that the positional-set y will contain the k-th element if and only if there exists a positional-set x in A such that k is a member of x. Effectively, the right hand expression performs a multiple bitwise OR on all positional-sets of χ_A to form a single positional-set y which represents the union of all such positional-sets. ■

Alternatively, we implemented the set *Union* operation as a recursive BDD operator. Bitwise OR is performed at the BDD DAG level, by traversing the BDD and performing OR on BDD vertices with the variables of interest.

Lemma 3.20 *Given a set of positional-sets $\chi(x)$ and an array of the Boolean variables x, the union of positional-sets in χ with respect to x can be computed by the BDD operator $Bitwise\_Or(\chi, 0, x)$, assuming that the variables in x are ordered last:*

```
Bitwise_Or(χ, k, x) {
    if (k ≥ |x|) return χ
    t = top_var(χ)
    if (t < x_k) {
        T = Bitwise_Or(χ_t, k, x)
        E = Bitwise_Or(χ_t̄, k, x)
        return ITE(t, T, E)
    } else {
        if (χ_{x_k} = 0) return x̄_k · Bitwise_Or(χ_{x̄_k}, k + 1, x)
        else return x_k · Bitwise_Or(χ_{x_k} + χ_{x̄_k}, k + 1, x)
    }
}
```

Proof: x_k denotes the k-th variable in the array x. Assuming that the variables in x are ordered last, the above recursion terminates after all of them have been processed ($k \geq |x|$, and a **0** or a **1** is returned as χ). At a BDD vertex where $t < x_k$, the recursion has not reached a variable of interest yet, and we simply recurse down its right and left children and merge the *Bitwise_Or* results by creating a new vertex $ITE(t, T, E)$. If $t \geq x_k$, we have to perform the bitwise OR operation on variable v. If $\chi_{x_k} = \mathbf{0}$, variable x_k never takes a value 1 in any satisfying assignments of χ, so it is set to 0 by $\overline{x_k}$. The bitwise OR of the remaining variables is given by $Bitwise\_Or(\chi_{\overline{x_k}}, k+1, x)$.

Otherwise if $\chi_{x_k} \neq 0$, there exists a satisfying assignment of χ in which $x_k = 1$. So x_k is set to 1, while a bitwise OR is performed over all remaining satisfying assignments of χ, i.e., $\chi_{x_k} + \chi_{\overline{x_k}}$. ■

This recursive BDD operator is very fast, but unfortunately, its operation is valid only if the variables to be bitwise OR are at the bottom of the BDD DAG. So to execute this BDD operator, we need to perform variable substitutions before and after the operation. Experimentally, these substitution steps are too slow to be practical and sometimes cause exponential blowup in the BDD size. As a result, we use the computation in Lemma 3.19 instead.

Lemma 3.21 *Given a characteristic function $\chi_A(x)$ representing a set A of positional-sets, the* **set intersection relation** *tests if positional-set y represents the intersection of all sets in A, and can be computed by:*

$$Intersect_{x \to y}(\chi_A) = \prod_{k=1}^{n} y_k \Leftrightarrow \forall x \, [\chi_A(x) \cdot x_k]$$

Proof: For each position k, the right hand expression sets y_k to 1 if and only if the k-th bit of all x in χ_A is a 1. This implies that the positional-set y will contain the k-th element if and only if all positional-sets x in χ_A have k has a member. Effectively, the right hand expression performs a multiple bitwise AND on all positional-sets of χ_A to form a single positional-set y which represents the intersection of all such positional-sets. ■

3.7.4 k-out-of-n Positional-sets

Let the number of objects be n. In subsequent computations, we will use extensively a suite of sets of sets of objects, $Tuple_{n,k}(x)$, which contains all positional-sets x with exactly k elements in them (i.e., $|x| = k$). In particular, the set of singleton element $Tuple_{n,1}(x)$, the set of pairs $Tuple_{n,2}(x)$, the universal set of all objects $Tuple_{n,n}(x)$, and the set of empty set $Tuple_{n,0}(x)$ [2] are common ones.

An efficient way of constructing and storing such collections of k-tuple sets using BDD will be given next. Figure 3.7 represents a reduced ordered BDD of $Tuple_{5,2}(x)$:

The root of the BDD represents the set $Tuple_{5,2}(x)$, while the internal nodes represent the sets $Tuple_{i,j}(x)$ $(i < 5, j < 2)$. For ease of illustration, the variable ordering is chosen such that the top variable corresponding to $Tuple_{i,j}(x)$ is x_i. At that node, if we choose element i to be in the positional-set, x_i takes the value 1 and we follow the right outgoing arc. In doing so, we still have $i - 1$ elements/variables left to be processed. As we have put element i in the positional-set,

[2] $Tuple_{n,0}(x)$ will be denoted by $\emptyset(x)$.

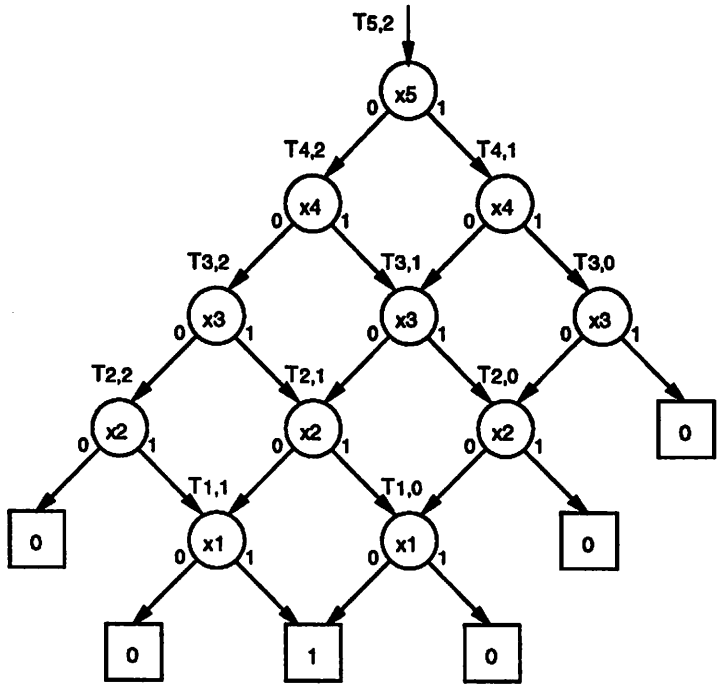

Figure 3.7: BDD representing $Tuple_{5,2}(x)$.

we still have to add exactly $j-1$ elements into the positional-set. That is why the right child of $Tuple_{i,j}(x)$ should be $Tuple_{i-1,j-1}(x)$. Similarly, the left child is $Tuple_{i-1,j}(x)$ because element i has not been put in the positional-set and we have $j-1$ elements/variables left. Thus, the BDD for $Tuple_{i,j}$ can be constructed by the algorithm shown in Figure 3.8.

The total number of nonterminal vertices in the BDD of $Tuple_{n,k}$ is $(n-k+1)\cdot(k+1)-1 = nk-k^2+n = O(nk)$. With the use of the computed table [6], the time complexity of the above algorithm is also $O(nk)$ as the BDD is built from bottom up and each vertex is built once and then re-used. Given any n, the BDD for $Tuple_{n,k}$ is largest when $k = n/2$.

3.8 FSM Representation using Both Encodings

A good representation for a problem is key to the development of efficient algorithms for it, and this is certainly true for problems in sequential synthesis and verification. A state transition graph (STG) (Definition 2.4) is commonly used as the internal representation of FSM's in sequential synthesis systems, such as SIS [73]. Many algorithms for sequential synthesis have been developed to apply to STG's. However, large FSM's cannot be stored and manipulated without memory

```
Tuple(i, j) {
    if (j < 0) or (i < j) return 0
    if (i = j) and (i = 0) return 1
    if Tuple(i, j) in computed-table return result
    T = Tuple(i − 1, j − 1)
    E = Tuple(i − 1, j)
    F = ITE(x_i, T, E)
    insert F in computed-table for Tuple(i, j)
    return F
}
```

Figure 3.8: Pseudo-code for the $Tuple$ operator.

usage and CPU time becoming prohibitively large. A limitation of STG's is the fact that they are a two-level form of representation where state transitions are stored explicitly, one by one. This may degrade the performance of conventional graph algorithms.

Assume that the given FSM has n states. To perform state minimization, one needs to represent and manipulate efficiently sets of states (such as compatibles) and sets of sets of states (such as sets of compatibles). Therefore *1-hot encoding* is used for the states of the FSM. According to the theory in Section 3.7, we can represent any set of sets of states (i.e., set of state sets) implicitly as a single 1-hot encoded MDD, and manipulate such state sets symbolically all at once. Different sets of sets of states can be stored as multiple roots with a single shared 1-hot encoded MDD.

If inputs (outputs respectively) of the FSM are specified symbolically, they can be represented as a multi-valued symbolic variable i (o respectively) where each value of i (o respectively) represents an input (output respectively) combination. For compactness in representation, we choose to use the *logarithmic encoding* for these variables. It is not a problem that different multi-valued variables use different encodings as long as they are used consistently. However if inputs (outputs respectively) of the FSM are already given in encoded form, each encoded bit of inputs (outputs respectively) is represented by a single Boolean variable.

When states and transitions are represented implicitly, the 1-hot encoded MDD represen-

tation is often much smaller than STG. There is no direct correlation between the complexity of the STG and the size of the corresponding 1-hot encoded MDD. Using these MDD relations and the positional-set notation, we propose new implicit algorithms in the coming chapter for generating various subsets of compatibles for solving different state minimization problems.

3.9 Variable Ordering

We can frequently suffer from exponential time and/or space complexities if we neglect the issue of variable ordering. As with most variants of BDD, the space and time complexities for constructing an MDD for any discrete function is in the worst case exponential in the number of variables of the function. Luckily in real life, most discrete functions have reasonable representations provided that a good variable ordering is chosen. Friedman *et al.* in [24] found an $O(n^2 3^n)$ algorithm for finding the optimal variable ordering where n is the number of Boolean variables. Faster variable ordering heuristics for BDD's have been provided by Malik *et al.* in [52] and Fujita *et al.* in [25]. Rudell [69] recently proposed an effective dynamic variable reordering heuristic which offers a tradeoff of runtime for compactness of BDD representation.

The goal in this section is to find a good variable ordering so as to minimize the total number of vertices used. With a mapped-BDD representation of an MDD, the ordering process consists of two steps: order the BDD variables within each MDD variable, and then merge these orderings into a single BDD variable ordering.

Two well-known rule-of-thumbs suggested in [52] and [25] can be used for the ordering of BDD variables within each individual MDD variable:

1. Variables that are closely related should be ordered close to each other.

2. Variables that dominate control of the function should be ordered at the top.

There are two ways of merging these individual orderings. **Cluster ordering** places BDD variables, which correspond to the same multi-valued variable, in consecutive positions in the final ordering. Within each cluster, the binary variable which corresponds to the most significant bit (MSB) is ordered first (i.e., highest). Then the next significant encoding variable is ordered next, and so on. For state minimization, cluster ordering is used to merge different sets of input and output variables. And they are ordered before (i.e., on top of) the state variables because the transition relation depends heavily on inputs and outputs.

Example In Figure 3.9, the relation $(x = y)$ is represented as an MDD on the left and a mapped-BDD (i.e., logarithmic encoded MDD) by cluster ordering in the middle. Variables x and y each can take four values. Note that the multi-valued variable x is encoded into two binary variables x_0 (MSB) and x_1 (LSB) on the right. The circled subgraph before reduction has the same number of outgoing arcs as the MDD vertex and the two representations are equivalent. With the mapped-BDD representation, vertices with equivalent subgraphs can be merged as shown by the two lowest nonterminal vertices.

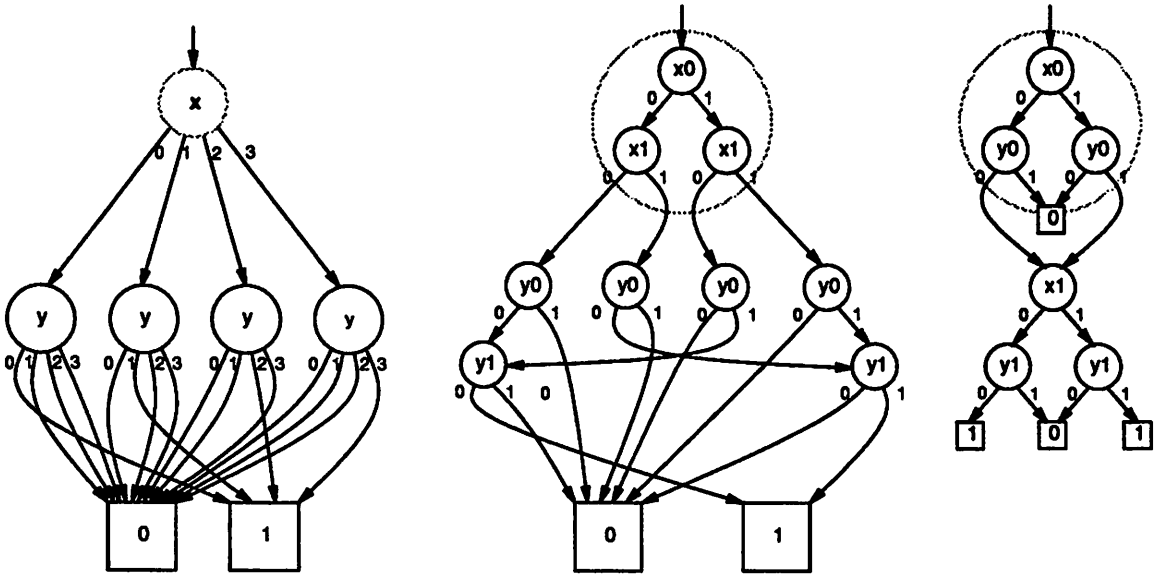

Figure 3.9: Comparison between cluster ordering and interleave ordering.

The problem of using cluster ordering for variables with large value ranges is illustrated by Figure 3.9. Consider the FSM named *squares* in the MCNC benchmark which has 371 states. Using 1-hot encoding on its states, multi-valued variables x and y each consists of 371 BDD variables. As mentioned before, there are about 5×10^{111} outgoing edges from the circled BDD subgraph cluster. In the worst case, it would have 5×10^{111} subgraphs below, each rooted at such an edge. Thus the size of the MDD will grow exponentially in the number of binary encoded variables.

To avoid such exponential growth, we use an **interleave ordering** for the BDD variables instead, as shown on the right of Figure 3.9. The encoded variables x_0 and x_1 for x are interleaved with y_0 and y_1 for y. The more significant bits are compared before the less significant ones as the mapped-BDD is traversed from top to bottom. With interleaving, the width of the mapped-BDD can be kept slim. As a comparison for our example, the mapped-BDD using cluster ordering has

9 nonterminal vertices while interleave ordering results in 6 nonterminal vertices. For the FSM *squares*, the mapped-BDD for $(x = y)$ have only $3 \times 371 = 1113$ nonterminal vertices using interleave ordering.

For state minimization, our implicit algorithms need to operate on multiple set of state variables, each such variable set can represent a positional-set. Interleave ordering must be used for these sets of variables for the reason described above. To avoid exponential complexities, a common wisdom is to use as few BDD variables as possible. In Section 4.6, we allocate only 4 sets of state BDD variables although a total of 10 state vector names are used in the equations. This is possible because we never have to operate on more than 4 sets of state variables simultaneously within a single BDD operation. The actual BDD variables are reused for different purposes, by binding at different times more than one set of variables from the equations onto a single set of BDD variables.

Part II

State Minimization of Incompletely Specified FSM's

Chapter 4

Compatible Generation

4.1 Introduction

State minimization of FSM's is a well-known problem [40]. State minimization of completely specified FSM's (CSFSM's) has a complexity subquadratic in the number of states [31]. This makes it an easy problem when the starting point is a two-level description of an FSM, because the number of states is usually less than a few hundred. The problem becomes difficult to manage when the starting point is an encoded sequential circuit with a large number of latches (in the hundreds). In that case the traditional method would be required to extract a state transition graph from the encoded network and then apply state minimization to it. But when latches are more than a dozen, the number of reachable states may be so huge to make state extraction and/or state minimization unfeasible. Recently it has been shown [49, 47] how to bypass the extraction step and compute equivalence classes of states implicitly. Equivalence classes are basically all that is needed to minimize a completely specified state machine. A compatible projection operator uniquely encodes each equivalence class by selecting a unique representative of the class to which a given state belongs. This implicit technique allows state minimization of sequential networks outside the domain of traditional techniques.

State minimization of incompletely specified FSM's (ISFSM's) instead has been shown to be an NP-hard problem [61]. Therefore even for problems represented with two-level descriptions involving a hundred states, an exact algorithm may consume too much memory and time. Moreover, it has been recently reported [43] that even examples with very few states generated during the synthesis of asynchronous circuits may fail to complete (or require days of CPU time) when run with a state-of-art exact state minimizer as STAMINA [63]. Therefore it is of practical importance

to revisit exact state minimization of ISFSM's and address the issue of representing implicitly the solution space.

We underline that besides the intrinsic interest of state minimization and its variants for sequential synthesis, the implicit techniques reported in this chapter can be applied to other problems of logic synthesis and combinatorial optimization. For instance the implicit computation of maximal compatibles given here can be easily converted into an implicit computation of prime encoding-dichotomies (see [71]). Therefore the computational methods described here contribute to build a body of implicit techniques whose scope goes much beyond a specific application.

In this chapter, we address the problem of computing sets of compatibles for the exact state minimization of ISFSM's [37]. We show how to compute sets of maximal compatibles, compatibles and prime compatibles with implicit techniques and demonstrate that in this way, it is possible to handle examples exhibiting a number of compatibles up to 2^{1500}, an achievement outside the scope of programs based on explicit enumeration [63]. We indicate also where such examples arise in practice. The final step of an implicit exact state minimization procedure, i.e., solving a binate table covering problem [68], will be presented in the next chapter, Chapter 5.

The remainder of this chapter is organized as follows. Section 4.2 gives an introduction to classical exact algorithms for state minimization of ISFSM's. An implicit version of the exact algorithm is presented in Section 4.3. Section 4.4 describes the method to generate the implicit binate table. It will be solved by a binate covering solver described in Chapter 5. Alternative implicit algorithms for prime compatible generation are explored in Section 4.5. A discussion of more subtle aspects of the implementation of the presented algorithms is given in Section 4.6. Results on a variety of benchmarks are reported and discussed in Section 4.7.

4.2 Classical Definitions and Algorithm

Most of the terminology used in this chapter is common parlance of the logic synthesis community [40, 7, 8]. In this section we will revise briefly the basic definitions and procedures for exact state minimization of ISFSM's, as presented in the original papers and standard textbooks [60; 28, 40]. First we restate the definition of incompletely specified FSM's. Some definitions will be restated in Section 4.3 where they will be explained in more detail.

Definition 4.1 *An* **incompletely specified FSM** *(ISFSM) can be defined as a 6-tuple* $M = \langle S, I, O, \Delta, \Lambda, R \rangle$. *$S$ represents the finite state space, I represents the finite input space and O*

represents the finite output space. Δ is the next state relation defined as a characteristic function $\Delta : I \times S \times S \rightarrow B$ where each combination of input and present state is related to a single next state or to all states. Λ is the output relation defined as a characteristic function $\Lambda : I \times S \times O \rightarrow B$ where each combination of input and present state is related to a single output or to all outputs. $R \subseteq S$ represents the set of reset states.

In classical literature, no reset state is specified for an ISFSM, and it is assumed that all states can potentially be picked as a reset state for implementation. The same is assumed in this chapter, and this is reflected in covering conditions defined later.

In addition, unspecified next state is traditionally not represented in the next state relation Δ. i.e., if the next state is not specified for present state s and input i, there is no state s' such that $\Delta(i, s, s') = 1$. This assumption is made in subsequent definitions and computations in this chapter.

Definition 4.2 *A set of states is an* **output compatible** *if for every input, there is a corresponding output which can be produced by each state in the set.*

Two state are an **output incompatible pair** if they are not output compatible.

Lemma 4.1 *Two states are an output incompatible pair if and only if on some input, they cannot produce the same output.*

Definition 4.3 *A set of states is a* **compatible** *if for each input sequence, there is a corresponding output sequence which can be produced by each state in the compatible.*

The set of compatibles can theoretically be computed as follows:

1. Assume that every output compatible is a candidate (for compatible).
2. A candidate is not a compatible if on some input, its states cannot produce a same output and transit to a candidate compatible set.
3. Repeat 2, until no candidate can be deleted from the set of candidate compatibles.

Practically, the number of candidates is too large to be handled by such an explicit algorithm.

Two states are a **incompatible pair** if they are not compatible.

Lemma 4.2 *Two states are an incompatible pair if and only if*

1. *they are output incompatible, or*

2. *on some input, their next states are an incompatible pair.*

The set of incompatible state pairs can be computed as follows:

1. Compute output incompatible pairs, which are incompatible pairs.

2. A pair of states is an incompatible pair if on some input, its pair of next states is a previously determined incompatible pair.

3. Repeat 2, until no new pairs can be added to the set of incompatible pairs.

Theorem 4.3 *Given an ISFSM, a set of states is a compatible if and only if every pair of states in it are compatible.*

Proof: By Definition 4.3, if a set of states is a compatible, then each pair of state contained in it is a compatible.

Suppose each pair of states in a set C of states is compatible. We shall prove that C is a compatible by induction on the length k of an arbitrary input sequence. On an arbitrary input i (as induction basis), each state in C can produce either one output or all outputs, because the machine is an ISFSM. No two states in C produce two *specified* outputs which is different because they are pairwise compatible. As a result, all states in C can produce a common output on input i. Assume that for an arbitrary input sequence σ_i of length k, every states in C can again produce a common output sequence and reach a set of states C'. As C is pairwise compatible, so must C'. By applying an extra input i, each state in C' can produce either one output or all outputs. No two states in C' produce different specified outputs, so all states in C' can produce a common output on input i. We have shown that the state in C, on any input sequence of length $k+1$, can also produce a common output sequence. Thus C is a compatible. ■

Corollary 4.4 *Given an ISFSM, a set of states which does not contain an incompatible pair is a compatible.*

Proof: If a set of states does not contain an incompatible pair, each pair of states in it are compatibles. By Theorem 4.3, the set of states is a compatible. ■

Definition 4.4 *A set of states d_i is an* **implied set** [1] *of a compatible c for input i if* $d_i = \{s' | \Delta(i, s, s') = 1, \forall s \in c\}$.

[1] An implied set d_i of c under i is simply the set of next states from state set c on input i.

Definition 4.5 *A set C of compatibles is a* **cover** *of an ISFSM if each state in the ISFSM is contained in a compatible in C.*

Definition 4.6 *A set C of compatibles is* **closed** *in an ISFSM if for each $c \in C$, all its implied sets c_i are contained in some element of C for each inputs i.*

Theorem 4.5 *The state minimization problem of an ISFSM reduces to finding a closed set C of compatibles, of minimum cardinality, which covers every state of the original machine, i.e., a* **minimum closed cover.**

Definition 4.7 *A compatible set of states is a* **maximal compatible** *if it is not a subset of another compatible.*

A set of states is a **maximal incompatible** if it is not a maximal compatible.
We give as example an elegant procedure to find all maximal compatibles that can be found in [53].

1. Write down the pairs of incompatibles as a product of sums
2. Multiply them out to obtain a sum of products, and minimize it with respect to single-cube containment.
3. For each resultant product, write down missing states to get maximal compatibles.

This is equivalent to compute all prime implicants of a unate function expressed as a product of sums (of pairs of states). For instance:

1. Product of pairs of incompatibles:
   $(s_4 + s_5)(s_4 + s_6)(s_4 + s_9)(s_5 + s_7)(s_6 + s_7)(s_6 + s_8)(s_8 + s_9)$
2. Unate function in sum of products: $s_4s_5s_6s_8 + s_4s_6s_7s_9 + s_4s_7s_8 + s_5s_6s_9$
3. Maximal compatibles: $s_1s_2s_3s_7s_9$, $s_1s_2s_3s_5s_8$, $s_1s_2s_3s_5s_6s_9$, $s_1s_2s_3s_4s_7s_8$

The set of all maximal compatibles of a completely specified FSM is the unique minimum closed cover. For an incompletely specified FSM, a closed cover consisting only of maximal compatibles may contain more sets than a minimum closed cover, in which some or all of the compatibles are proper subsets of maximal compatibles.

Definition 4.8 *An implied set d of a compatible c is in its* **class set** *if*

1. *d has more than one element, and*

2. $d \not\subseteq c$, *and*

3. $d \not\subseteq d'$ *if* $d' \in$ *class set of* c.

Definition 4.9 *A compatible* c' **prime dominates** [2] *a compatible* c *if*

1. $c' \supset c$, *and*

2. *class set of* $c' \subseteq$ *class set of* c.

i.e., c' dominates c if c' covers all states covered by c and the closure conditions of c' are a subset of the closure conditions of c.

Definition 4.10 *A compatible set of states is a* **prime compatible** *if it is not dominated by any other compatible.*

The following procedure (which will be used in Section 4.5.3) generates all prime compatibles from the set of maximal compatibles [28].

1. Initially the set of prime compatibles is empty.

2. Order the maximal compatibles by decreasing size, say n is the size of the largest.

3. Add to the set of prime compatibles the maximal compatibles of size n.

4. For $k = n - 1$ down to 1 do:

   (a) Compatibles of size k (and their implied sets) are generated starting from the maximal compatibles of size n to $k + 1$ (only those having non-void class set).

   (b) Add to the set of prime compatibles the compatibles of size k not dominated by any prime compatible already in the set.

   (c) Add to the set of prime compatibles all maximal compatibles of size k.

The following facts are true about the above algorithm:

- A compatible already added to the set of primes cannot be excluded by a newly generated compatible.

[2] If we take Definition 2.37 as the definition of prime dominance, this Definition 4.9 represents only a sufficient condition for prime dominance.

- The same compatible can be generated more than once by different maximal compatibles. The question arises of finding the most efficient algorithm to generate the compatibles.
- Only the compatibles generated from maximal compatibles with non-void class set need be considered, because a maximal compatible with a void class set dominates any compatible that it generates.
- A single state s can be a prime compatible if every compatible set C with more than one state and containing s implies a set with more than one state.

Definition 4.11 *A prime compatible is an* **essential prime compatible** *if it contains a state not contained in any other prime compatibles.*

The following theorem is proved in [28], and its generalization to PNDFSM has been given in Theorem 2.24.

Theorem 4.6 *For any ISFSM M, there is a reduced ISFSM M_{red} whose states all correspond to prime compatibles of M.*

A minimum closed cover can then be found by setting up a table covering problem [28] whose columns are the prime compatibles and rows corresponds to the covering and closure conditions.
The following facts are useful in the state minimization of FSM's:

- The cardinality of a maximal incompatible is a lower bound on the number of states of the minimized FSM.
- If there is a maximal compatible that contains all states of a given FSM, the FSM reduces to a single state.
- The cardinality of the set of maximal compatibles is an upper bound on the number of states of the minimized FSM.
- If a maximal compatible has a void class set, it must be a prime compatible. As a result, no compatible contained in it can be a prime compatible (result used in Section 4.5.3).
- The minimum number of maximal compatibles covering all states is a lower bound on the number of states of the minimized FSM.

- The minimum number of maximal compatibles covering all states and satisfying the closure conditions is an upper bound on the number of states of the minimized FSM.

4.3 Implicit Generation of Compatibles

An exact algorithm for state minimization consists of two steps: the generation of various sets of compatibles, and the solution of a binate covering problem. The generation step involves identification of sets of states called compatibles which can potentially be merged into a single state in the minimized machine. Unlike the case of CSFSM's, where state equivalence partitions the states, compatibles for incompletely specified FSM may overlap. As a result, the number of compatibles can be exponential in the number of states [66], and the generation of the whole set of compatibles can be a challenging task.

The covering step (which will be described in Sections 5) is to choose a minimum subset of compatibles satisfying covering and closure conditions, i.e., to find a minimum closed cover. The covering conditions require that every state is contained in at least one chosen compatible. The closure conditions guarantee that the states in a chosen compatible are mapped by any input sequence to states contained in a chosen compatible.

In this section, we describe implicit computations to find sets of compatibles required for exact state minimization of ISFSM's. In each of the following subsections, we shall first restate the definition of some theoretical object, and give a logic formula to compute it, and then argue its correctness in a proof.

4.3.1 Output Incompatible Pairs

To generate compatibles, incompatibility relations between pairs of states are derived first from the given output and transition relations of an ISFSM.

Definition 4.12 *Two states are an* **output incompatible pair** *if, for some input, they cannot generate the same output.*

Lemma 4.7 *The set of output incompatible pairs, $\mathcal{OICP}(y,z)$, can be computed as:*

$$\mathcal{OICP}(y,z) = Tuple_1(y) \cdot Tuple_1(z) \cdot \exists i \; \not\exists o \, [\Lambda(i,y,o) \cdot \Lambda(i,z,o)] \tag{4.1}$$

Proof: Although y and z can represent any positional-sets, the conditions $Tuple_1(y) \cdot Tuple_1(z)$ restrict them to represent only pairs of singleton states. The last term is true if and only if for some input i, there is no output pattern that both state y and z can produce (i.e., output incompatible). ■

In the above and subsequent formulas, we will mix notations between relations and their corresponding characteristic functions. Strictly speaking if we used the characteristic function notation, the above formula would have been more clumsy:

$$\mathcal{OICP}(y,z) = 1 \quad \text{if and only if} \quad (Tuple_1(y) = 1) \cdot (Tuple_1(z) = 1) \cdot \exists i \not\exists o \, [(\Lambda(i,y,o) = 1) \cdot (\Lambda(i,z,o) = 1)]$$

4.3.2 Incompatible Pairs

Definition 4.13 *Two states are an* **incompatible pair** *if*

1. *they are output incompatible, or*

2. *on some input, their next states are an incompatible pair.*

Lemma 4.8 *The set of incompatible pairs is the least fixed point of* $\mathcal{ICP}$*:*

$$\mathcal{ICP}(y,z) = \mathcal{OICP}(y,z) + \exists i,u,v \, [\Delta(i,y,u) \cdot \Delta(i,z,v) \cdot \mathcal{ICP}(u,v)]$$

and can be computed by the following iteration:

$$\begin{aligned} \mathcal{ICP}_0(y,z) &= \mathcal{OICP}(y,z) \\ \mathcal{ICP}_{k+1}(y,z) &= \mathcal{ICP}_k(y,z) + \exists i,u,v \, [\Delta(i,y,u) \cdot \Delta(i,z,v) \cdot \mathcal{ICP}_k(u,v)] \end{aligned} \tag{4.2}$$

The iteration can terminate when $\mathcal{ICP}_{k+1} = \mathcal{ICP}_k$ *and the set of incompatible pairs is* $\mathcal{ICP}(y,z) = \mathcal{ICP}_k(y,z)$.

Proof: The fixed point computation starts with the set of output incompatible pairs. After the $(k+1)$-th iteration of Equation 4.2, $\mathcal{ICP}_{k+1}(y,z)$ contains all the incompatible state pairs (y,z) that lead to an output incompatible pair in $k+1$ or less transitions. This set is obtained by adding state pairs (y,z) to the set $\mathcal{ICP}_k(y,z)$, if an input takes states (y,z) into an already known incompatible pair (u,v). ■

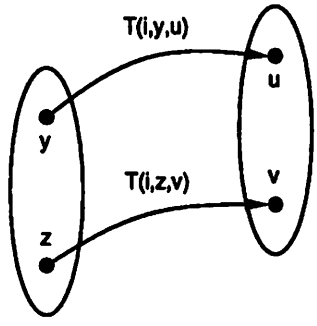

Figure 4.1: Finding incompatible pairs.

4.3.3 Incompatibles

So far we established incompatibility relationships between pairs of states. The following definition introduces sets of states of arbitrary cardinalities.

Definition 4.14 *A set of states is an* **incompatible** *if it contains at least one incompatible pair.*

Lemma 4.9 *The set of incompatibles can be computed as:*

$$\mathcal{IC}(c) = \exists y, z\,[\mathcal{ICP}(y,z) \cdot (c \supseteq y \cup z)] \tag{4.3}$$

Proof: By Lemma 3.10, $(c \supseteq y \cup z) = \prod_{k=1}^{n} y_k + z_k \Rightarrow c_k$ performs bitwise OR on singletons y and z. If either of their k-th bits is 1, the corresponding c_k bit is constrained to 1. Otherwise, c_k can take any values. The outer product $\prod_{k=1}^{n}$ requires that the above is true for each k. Thus, it generates all positional-sets c which contain the union of the positional-sets y and z. The whole computation defines all state sets c each of which contains at least an incompatible pair of singleton states $(y, z) \in \mathcal{ICP}$. ■

4.3.4 Compatibles

Definition 4.15 *A set of states is a* **compatible** *if it is not an incompatible.*

Lemma 4.10 *The set of compatibles, $\mathcal{C}(c)$, can be computed as:*

$$\mathcal{C}(c) = \neg Tuple_0(c) \cdot \neg\mathcal{IC}(c)$$

Proof: $\mathcal{C}(c)$ simply contains all non-empty subsets c of states which are not incompatibles $\mathcal{IC}(c)$. The empty set in positional-set notation is $Tuple_0(c)$ and all subsets which are not incompatible are given by $\neg\mathcal{IC}(c)$. ■

4.3.5 Implied Classes of a Compatible

To set up the covering problem, we also need to compute the closure conditions for each compatible. This is done by finding the class set of a compatible, i.e., the set of next states implied by a compatible.

Definition 4.16 *A set of states d_i is an* **implied set** *of a compatible c for input i if d_i is the set of next states from the states in c on input i.*

Lemma 4.11 *The implied set (in singleton form) of a compatible c for input i can be defined by the relation $\mathcal{F}(c, i, n)$ which evaluates to 1 if and only if on input i, n is a next state from state p in compatible c.*

$$\mathcal{F}(c, i, n) = \exists p \, [\mathcal{C}(c) \cdot (c \supseteq p) \cdot \Delta(i, p, n)] \tag{4.4}$$

Proof: $\mathcal{F}(c, i, n)$ associates a compatible $c \in \mathcal{C}$ and an input i with a singleton next state n. Given c and i, n is in relation $\mathcal{F}(c, i, n)$ (i.e., state n is in the implied set of compatible c under input i) if and only if there is a present state $p \in c$ such that n is the next state of p on input i. ■

Note that the implied next states are represented here as singleton states in $\mathcal{F}(c, i, n)$. All singletons n in relation with a compatible c and an input i can be combined into a single positional-set, for later convenience. This positional-set representation of implied sets associates each compatible c with a set of implied sets d.

Lemma 4.12 *The implied sets d (in positional-set form) of a compatible c for all inputs are computed by the relation $\mathcal{CI}(c, d)$ as:*

$$\mathcal{CI}(c, d) = \exists i \, [\exists n(\mathcal{F}(c, i, n)) \cdot Union_{n \to d}(\mathcal{F}(c, i, n))]$$

Proof: Considering the rightmost term, $\mathcal{F}(c, i, n)$ relates implied next states as singleton positional-sets n to compatible c and input i and $Union_{n \to d}(\mathcal{F}(c, i, n))$ forms the union of these singleton sets by bitwise OR and produces a positional-set d. The term $\exists n(\mathcal{F}(c, i, n))$ is needed, to exclude invalid (compatible, input) combinations. Finally the inputs i are existentially quantified from the implied sets of c of different inputs. ■

4.3.6 Class Set of a Compatible

Definition 4.17 *An implied set d of a compatible c is in its* **class set** *if*

1. *d has more than one element, and*

2. $d \not\subseteq c$, *and*

3. $d \not\subseteq d'$ *if* $d' \in$ *class set of* c.

We can ignore any implied set which contains only a single state, because its closure condition is satisfied if the state is covered by some chosen compatible. Also if $d \subseteq c$, the closure condition is satisfied by the choice of c. Finally, if the closure condition corresponding to d' is stronger than that of d, the implied set d is not necessary.

Lemma 4.13 *The class set of a compatible c is defined by the relation* $CCS(c, d)$ *which evaluates to 1 if and only if the implied set d is in the class set of compatible c.*

$$CCS(c, d) = \neg Tuple_1(d) \cdot (c \not\supseteq d) \cdot Maximal_d(C\mathcal{I}(c, d))$$

Proof: The singleton implied sets $Tuple_1(d)$ are excluded according to condition 1 in Definition 4.17. By condition 2, we prune away implied sets d which are contained in their compatibles c. Finally given a compatible c, $Maximal_d(C\mathcal{I}(c, d))$ gives all its implied sets d which are not strictly contained by any other implied sets in $C\mathcal{I}(c, d)$. ■

4.3.7 Prime Compatibles

To solve exactly the covering problem, it is sufficient to consider a subset of compatibles called prime compatibles. As proved in [28], at least one minimum closed cover consists entirely of prime compatibles.

Definition 4.18 . *A compatible* c' **dominates** *a compatible c if*

1. $c' \supset c$, *and*

2. *class set of* $c' \subseteq$ *class set of c.*

i.e., c' dominates c if c' covers all states covered by c, and the closure conditions of c' are a subset of the closure conditions of c. As a result, compatible c' expresses strictly less stringent conditions than compatible c. Therefore c' is always a better choice for a closed cover than c, thus c can be excluded from further consideration.

Lemma 4.14 *The prime dominance relation is given by:*

$$Dominate(c', c) = (c' \supset c) \cdot Contain_d(CCS(c, d), CCS(c', d))$$

Proof: The two terms on the right express the two dominance conditions by which c' dominates c according to Definition 4.18. Since compatibles c and c' are represented as positional-sets, $(c' \supset c)$ is computed according to Lemma 3.7. On the other hand, class sets are sets of sets of states and are represented by their characteristic functions. Containment between such sets of sets of states is computed by $\forall d\,[CCS(c', d) \Rightarrow CCS(c, d)]$, as described by Lemma 3.17. ■

Definition 4.19 *A* **prime compatible** *is a compatible not dominated by another compatible.*

Lemma 4.15 *The set of prime compatibles is given by:*

$$\mathcal{PC}(c) = C(c) \cdot \nexists c'\,[C(c') \cdot Dominate(c', c)]$$

Proof: Compatibles c that are dominated by some compatible c' are computed by the expression $\exists c'\,[C(c') \cdot Dominate(c', c)]$. By Definition 4.19, the set of prime compatibles is simply given by the set of compatibles $C(c)$ excluding those that are dominated. ■

4.3.8 Essential and Non-essential Prime Compatibles

Definition 4.20 *A prime compatible is an* **essential prime compatible** *if it contains a state not contained in any other prime compatibles.*

Because any solution must correspond to a closed *cover*, each state mush be contained in a selected compatible, and thus every essential prime compatible must be selected.

Lemma 4.16 *The set of essential prime compatibles can be computed as:*

$$\mathcal{EPC}(c) = \mathcal{PC}(c) \cdot \sum_{k=1}^{n} \{c_k \cdot \nexists c'\,[c'_k \cdot \mathcal{PC}(c') \cdot (c \neq c')]\}$$

Proof: For a set c of states to be an essential prime compatible $\mathcal{EPC}(c)$, it must be a prime compatible $\mathcal{PC}(c)$. In addition, there must be a state s_k such that $s_k \in c$ and there is no $c' \in \mathcal{PC}$ different than c such that $s_k \in c'$. The positive literal c_k denotes the fact $s_k \in c$, and similarly for c'_k. ■

Definition 4.21 . *A prime compatible is a* **non-essential prime compatible** *if it is not an essential prime compatible.*

Lemma 4.17 *The set of non-essential prime compatibles can be computed as:*

$$\mathcal{NEPC}(c) = \mathcal{PC}(c) \cdot \neg \mathcal{EPC}(c)$$

Proof: By Definition 4.21. ■

4.4 Implicit Generation of Binate Covering Table

Once the set of (non-essential) prime compatibles is generated, the problem of exact state minimization can be solved as a binate table covering problem. Algorithms for binate covering will be described in detail in Chapter 5. In this section, we shall describe how such a binate table can be generated. To keep with our stated objective, the binate table is also represented implicitly. We describe an implicit representation of the covering table, that adroitly exploits how row and columns were implicitly computed.

We do not represent (even implicitly) the elements of the table, but we make use only of a set of row labels and a set of column labels, each represented implicitly as a BDD. They are chosen so that the existence and value of any table entry can be readily inferred by examining its corresponding row and column labels. This choice allows us to define all table manipulations needed by the reduction algorithms in terms of operations on row and column labels and to exploit all the special features of the binate covering problem induced by state minimization (for instance, each row has at most one 0, etc).

Definition 4.22 *A column is labeled by a positional-set p. The set of* **column labels** *C is obtained by prime generation as $C(p) = \mathcal{PC}(p)$.*

Beside distinguishing one row from another, each row label must also contain information regarding the positions of 0 and 1's in the row. Each row label r consists of a pair of positional-sets (c, d). Since there is at most one 0 in the row, the label of the column p intersecting it in a 0 is recorded in the row label by setting its c part to p. If there is no 0 in the row, c is set to the empty set, $Tuple_0(c)$. Because of Definition 4.24 for row labels, the columns intersecting a row labeled $r = (c, d)$ in a 1 are labeled by the prime compatibles p that contain d. i.e.,

Definition 4.23 *The table entry at the intersection of a row labeled by $r = (c, d) \in R$ and a column labeled by $p \in C$ can be inferred by:*

the table entry is a 0 iff relation $0(r, p) \stackrel{\text{def}}{=} (p = c)$ *is true,*

the table entry is a 1 iff relation $1(r, p) \stackrel{\text{def}}{=} (p \supseteq d)$ *is true.*

Definition 4.24 *The set of* **row labels** *R is given by:*

$$R(r) = \mathcal{PC}(c) \cdot \mathcal{CCS}(c,d) + Tuple_0(c) \cdot Tuple_1(d)$$

The closure conditions associated with a prime compatible p are that if p is included in a solution, each implied set d in its class set must be contained in at least one chosen prime compatible. A binate clause of the form $(\overline{p} + p_1 + p_2 + \cdots + p_k)$ has to be satisfied for each implied set of p, where p_i is a prime compatible containing the implied set d. The labels for binate rows are given succinctly by $\mathcal{PC}(c) \cdot \mathcal{CCS}(c,d)$. There is a row label for each (c,d) pair such that $c \in \mathcal{PC}$ is a prime compatible and d is one of its implied sets in $\mathcal{CCS}(c,d)$. This row label consistently represents the binate clause because the 0 entry in the row is given by the column labeled by the prime compatible $p = c$, and the row has 1's in the columns labeled by p_i wherever $(p_i \supseteq d)$.

The covering conditions require that each state be contained by some prime compatible in the solution. For each state $d \in S$, a unate clause has to be satisfied which is of the form $(p_1 + p_2 + \cdots + p_j)$ where the p_i's are the prime compatibles that contain the state d. By specifying the unate row labels to be $Tuple_0(c) \cdot Tuple_1(d)$, we define a row label for each state in $Tuple_1(d)$. Since the row has no 0, its c part must be set to $Tuple_0(c)$. The 1 entries are correctly positioned at the intersection with all columns labeled by prime compatibles p_i which contain the singleton state d.

4.5 Improvements on Implicit Algorithm

The experiments, which will be reported in Section 4.7, identifies two bottlenecks in the implicit computations described in Section 4.3:

1. the fixed point computation of incompatible pairs;

2. the handling of closure information, i.e., implied classes and class sets.

Sections 4.5.1 and 4.5.2 describes alternative methods to perform those computations. Section 4.5.3 shows how maximal compatibles can be used with advantage in the computation of prime compatibles. The reader may choose to skip this and the next section (Sections 4.5 and 4.6) without losing continuity in reading.

4.5.1 Incompatible Pairs Generation using Generalized Cofactor

This subsection describes a variation on the fixed point computation of incompatible pairs $\mathcal{ICP}(y,z)$, presented in Section 4.3.2. Each iteration of the computation of Equation 4.2 can be viewed as an inverse image projection from a set of state pairs in $\mathcal{ICP}_k(u,v)$ to a set of states pairs in $\mathcal{ICP}_{k+1}(y,z)$ via the product transition relation $\Delta(i,y,u)\cdot\Delta(i,z,v)$. In the original method, all state pairs in $\mathcal{ICP}_{k+1}(u,v)$ are projected during the $(k+2)$-th iteration. Some are not necessary because if the projected pair (y',z') of $\mathcal{ICP}_{k+1}$ is actually in $\mathcal{ICP}_k$ as shown in Figure 4.2 [3], its projection (y'',z'') must have already been calculated in a previous iteration. Thus at the $(k+2)$-th iteration, we need only project the *new* incompatible state pairs discovered at the $(k+1)$-th iteration. This is done in the following modification of the fixed point computation of Section 4.3.2.

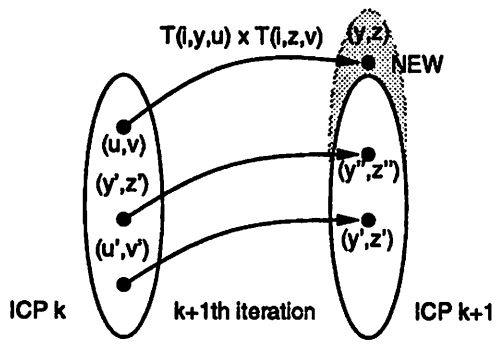

Figure 4.2: An alternative way of finding incompatible pairs.

$$
\begin{aligned}
\mathcal{ICP}_0(y,z) &= \mathcal{OICP}(y,z) \\
&(= NEW(y,z)) \\
TMP(y,z) &= \exists i,u\,\{\,\Delta(i,y,u)\cdot[\,\exists v\,\Delta(i,z,v)\cdot NEW(u,v)\,]\,\} \\
NEW(y,z) &= TMP(y,z)\cdot\neg\mathcal{ICP}_k(y,z) \\
\mathcal{ICP}_{k+1}(y,z) &= \mathcal{ICP}_k(y,z)+NEW(y,z)
\end{aligned}
$$

Instead of finding a minimum cardinality set of state pairs for projection, a minimal set of state pairs with a small BDD representation is more desirable for our implicit BDD formulation. A small BDD for $NEW(y,z)$ can be obtained using the generalized cofactor [80] using $\mathcal{ICP}_k(y,z)$ as the don't care set:

$$NEW(y,z) = NEW(y,z)|_{\neg\mathcal{ICP}_k(y,z)}$$

[3] In Figure 4.2, the direction of the arrows shows the *inverse* projections.

As a result of this modification, the geometric mean of the ratio of CPU run time for computing $\mathcal{ICP}$ with this generalized cofactor method versus the original method is 0.68.

4.5.2 Handling of Closure Information

For FSM's with many compatibles, the most time consuming part of our implicit algorithm is the computation of implied classes and class sets of the compatibles. The complexity arises because these implicit computations deal with two sets of variables (c, d) in each relation, c representing a compatible and d representing its implied class or class set. Since each compatible may have a different class set, the size of the corresponding BDD's may blowup during the computation.

A way to cope with this problem is to represent the class sets by means of singletons n (by Lemma 4.11) instead of sets d of arbitrary cardinalities (by Lemma 4.12). This avoids the computation of the BDD objects $\mathcal{CI}(c, d)$ and $\mathcal{CCS}(c, d)$ which are usually large and time consuming. The following series of computations is equivalent to and replaces the computations in Sections 4.3.5, 4.3.6 and 4.3.7.

Theorem 4.18 *The class sets of compatibles can be obtained by pruning the implied set relation $\mathcal{F}(c, i, n)$ in singleton form, by the following computations:*

$$
\begin{aligned}
\mathcal{F}(c,i,n) &= \exists p\,[\Delta(i,p,n)\cdot C(c)\cdot(c \supseteq p)] \\
I(c,i) &= \exists n\, n'\,[\mathcal{F}(c,i,n)\cdot\mathcal{F}(c,i,n')\cdot(n \neq n')] \qquad (4.5)\\
\mathcal{F}(c,i,n) &= \mathcal{F}(c,i,n)\cdot I(c,i) \\
J(c,i) &= \exists n\,[\mathcal{F}(c,i,n)\cdot(c \not\supseteq n)] \\
\mathcal{F}(c,i,n) &= \mathcal{F}(c,i,n)\cdot J(c,i) \\
K(c,i) &= \exists\, i'\,\{\,[Contain_n(\mathcal{F}(c,i,n),\mathcal{F}(c,i',n)) + \neg J(c,i')] \\
&\quad \cdot\neg[Contain_n(\mathcal{F}(c,i',n),\mathcal{F}(c,i,n)) + \neg J(c,i')]\,\} \qquad (4.6)\\
\mathcal{F}(c,i,n) &= \mathcal{F}(c,i,n)\cdot\neg K(c,i)
\end{aligned}
$$

Proof: For each input i and each compatible c, we must compute a unique implied set denoted by c_i. Each implied class c_i is represented as a set of singleton-states n, i.e., $c_i = \{n|\mathcal{F}(c,i,n)\}$. Definition 4.17, an implied class c_i of a compatible c is in a class set if

1. c_i has more than one element,

2. $c_i \not\subseteq c$,

3. $c_i \not\subseteq c_{i'}$ if $c_{i'} \in$ class set.

The relations I, J and K capture the implied sets that satisfy the first, second and third class set conditions respectively. The relation $I(c,i)$ finds all implied classes c_i which contains at least two distinct next states n and n', i.e., it computes all implied classes with more than one element. Equation 4.5 prunes the set $\mathcal{F}$ accordingly. Relation $J(c,i)$ contains all remaining implied classes not contained in c, thus satisfying the second condition.

To compute the third condition implicitly, its form needs to be transformed. From the set of implied classes, we delete an implied class c_i if $c_i \subset c_{i'}$. We know that, $c_i \subset c_{i'}$ if and only if $c_i \subseteq c_{i'}$ and $c_{i'} \not\subseteq c_i$. The $Contain_n(\mathcal{F}(c,i,n), \mathcal{F}(c,i',n))$ operation is used to test if $c_i \subset c_{i'}$, but since its result may include invalid (c,i') pairs (i.e., implied classes) the terms $\neg J(c,i')$ are needed in the equation. In the last equation, the implied sets in $K(c,i)$ are subtracted away from $\mathcal{F}$ because they violate the third condition. ■

Theorem 4.19 *The condition that compatible c' dominates compatible c can be computed as:*

$$Dominate(c',c) = (c' \supset c) \cdot \forall i' \, \exists i \, Contain_n(\mathcal{F}(c,i,n), \mathcal{F}(c',i',n))$$

Proof: By Definition 4.18, c' dominates c if c' covers all states covered by c and the conditions on the closure of c' are a subset of the conditions on the closure of c. ■

After computing the dominance relation $Dominate(c',c)$, the prime compatibles can be generated using Lemma 4.15.

4.5.3 Prime Compatible Generation using Maximals

The prime compatible generation algorithm given in Section 4.3 does not rely on the computation of maximal compatibles, whereas the classical method in [28] does. We are going to present alternative implicit generation algorithm that does make use of maximal compatibles. The motivation is to expedite the prime generation step, as it is usually the most time consuming step.

Theorem 4.20 *The set of maximal compatibles can be computed as:*

$$\mathcal{MC}(c) = Maximal_c(\mathcal{C}(c))$$

Proof: By Definition 4.7, the set of maximal compatibles $\mathcal{MC}(c)$ is simply the *maximal* set of positional-sets in $\mathcal{C}(c)$ with respect to c (Lemma 3.13). ■

Theorem 4.21 *The set of singleton next states implied by a maximal compatible c under input i, $\mathcal{F}(c, i, y)$, can be computed by:*

$$\mathcal{F}(c, i, n) = \exists p \, [\mathcal{MC}(c) \cdot (c \supseteq p) \cdot \Delta(i, p, n)]$$

The class set information for the maximal compatibles can then be obtained using the class set computation procedure as described in Theorem 4.18.

Proof: Obvious from definition. ■

Compatible Pruning by Maximal Compatibles with Void Class Set

When generating prime compatibles from compatibles, only the compatibles generated from maximal compatibles with non-void class set need be considered, because a maximal compatible with a void class set dominates any compatible that it generates.

Theorem 4.22 *The maximal compatibles with void class set, $\mathcal{MCV}(c)$, can be obtained by:*

$$\mathcal{MCV}(c) = \mathcal{MC}(c) \cdot \nexists i \; K(c, i)$$

where $K(c, i)$ is given by Equation 4.6.

The set of compatibles can then be pruned by $\mathcal{MCV}(c)$:

$$C(c) = C(c) \cdot \nexists c' \, [\mathcal{MCV}(c) \cdot (c' \supseteq c)]$$

Slicing Procedure for Prime Compatible Generation

The following slicing procedure is an implicit version of the procedure outlined near the end of Section 4.2.

$\mathcal{PC}(c) = \emptyset$
for $k = n$ down to 1 do{
$\mathcal{MC}_k(c) = \mathcal{MC}(c) \cdot Tuple_k(c)$
$C_k(c) = C(c) \cdot Tuple_k(c) \cdot \neg \mathcal{MC}_k(c)$
$\mathcal{F}_{C_k}(c, i, n) = Prune(C_k(c), \Delta(i, p, n))$
$\mathcal{F}_{\mathcal{PC}}(c, i, n) = Prune(\mathcal{PC}(c), \Delta(i, p, n))$
$Dominate(c', c) = (c' \supset c) \cdot \forall i' \exists i\, Contain_n(\mathcal{F}_{\mathcal{PC}}(c, i, n), \mathcal{F}_{C_k}(c', i', n))$
$\mathcal{PC}_k(c) = C(c) \cdot \nexists c' [Dominate(c', c) \cdot C(c')]$
$\mathcal{PC}(c) = \mathcal{PC}(c) + \mathcal{PC}_k(c) + \mathcal{MC}_k(c)$
}

$\mathcal{PC}(c)$ is a set of prime compatibles accumulated during each iteration, and is originally empty. $\mathcal{MC}_k(c)$ contains maximal compatibles with cardinality k. $C_k(c)$ contains compatibles c of cardinality k, excluding those in $\mathcal{MC}_k(c)$. $Prune(C_k(c), \Delta(i, p, n))$ is the class set pruning procedure described in Theorem 4.18, by substituting $C_k(c)$ for $C(c)$, and $\mathcal{F}_C(c, i, n)$ for $\mathcal{F}(c, i, n)$ in the equations. $Prune(\mathcal{PC}(c), \Delta(i, p, n))$ is similarly defined. So $\mathcal{F}_{C_k}(c, i, n)$ and $\mathcal{F}_{\mathcal{PC}}(c, i, n)$ contains the class sets of $C_k(c)$ and $\mathcal{PC}(c)$ respectively. To test for $Dominate(c', c)$, we only need to know if a compatible $c' \in C_k$ is dominated by an already discovered prime compatible $c \in \mathcal{PC}$, because (1) for any other $c' \in C_k$, $c \not\subset c'$, and (2) c can be dominated only by prime compatibles with cardinalities greater than k. $\mathcal{PC}_k(c)$ contains the newly discovered prime compatibles with cardinality k, and this set is added to $\mathcal{MC}_k$ and $\mathcal{PC}$ to update the set of prime compatibles found so far.

Experimentally during BDD construction of the prime compatibles, this slicing method uses on average half the memory requirement as compared to the method in Section 4.5.2. This method is particularly efficient if the sizes of the compatibles are localized to a few cardinalities.

4.6 Implementation Details

4.6.1 BDD Variable Assignment

When dealing with BDD's, a common wisdom is to keep the number of BDD variables used to a minimum. The rationale the smaller the number of BDD variables involved, the less probable is that a BDD operation will cause exponential blowup in the BDD size. In our case

1. 10 state variable vectors $(p, n, y, z, u, v, c, c', d, d')$ are used in all previous equations,
2. in positional-set notation, each state variable vector corresponds to n Boolean variables where n is the number of states.

Looking into each equation carefully reveals the fact that we never operate on more than four sets of variables simultaneously in a single BDD operation. For example, 4 sets of variables y, z, u and v are used in Equation 4.2, and 3 sets p, n and c in Equation 4.4. The idea of *BDD variable assignment* is to use a set of BDD variables for more than one purpose, by binding at different times more than one set of variables from the equations onto a single set of BDD variables. The assignments should be made in such a way that no two sets of variables appearing in an equation will be assigned to the same set of BDD variables. Such an assignment for our previous implicit algorithm is shown in Figure 4.3.

BDD variable sets			
0	1	2	3
		p	n
y	u	z	v
c	d	c'	d'
			e

Figure 4.3: Assignments of equation variables to BDD variables.

There is a conflict with the above BDD variable assignment in Equation 4.3. Variable c is assigned the same BDD variables as variable y in these equations. To get around it, an extra variable e is used instead:

$$\mathcal{IC}(c) = [e \rightarrow c]\exists y, z\,[\mathcal{ICP}(y, z) \cdot (y \supseteq z \cup e)]$$

Note that two functions containing different variables being assigned to the same BDD variable, e.g., $\Delta(i, p, n)$ and $\mathcal{CCS}(c', d')$, can co-exist within a multi-rooted BDD *at the same time*, without interfering one another. Conflict will occur only when they become operands to a BDD operation. Actually as a result of overlapping in variable supports, such unrelated functions can be constructed and manipulated more efficiently due to sharing of BDD subgraphs, i.e., possible hits in the unique and computed hash tables [6] in a BDD package.

4.6.2 BDD Variable Ordering

The issue of BDD variable ordering has been introduced in Section 3.9.

The equality, containment, strict containment, maximal and minimal relations described in Section 3.7.2 have exponential BDD's size if the different sets of BDD variables are not interleaved with each other. Both for space and time efficiency, the four sets of BDD variables have to be interleaved.

It is found that the ordering between individual state variables within a set of BDD variables (i.e., a positional-set) is also important, especially when handling the closure information. The heuristics we use is to put the Boolean variables corresponding to states that occur most frequently in the compatibles at the top of the BDD. This should leave the BDD sparse in the lower part of the BDD where most state variables take a value of 0. As the set of compatibles is usually very large, we approximate the count by counting the occurrences of states in maximal compatibles instead.

4.6.3 Don't Cares in the Positional-set Space

The main advantage of our positional-set representation of FSM's is that, with a single multi-rooted BDD, sets of sets of states can be represented. As a result, we can compactly represent and manipulate sets of compatibles ($\mathcal{C}$), prime compatibles ($\mathcal{PC}$), etc. However during the computation of $\mathcal{OCP}$, $\mathcal{OICP}$ and $\mathcal{ICP}$, we are manipulating only sets of singleton states and so we only *care* about a small portion of the encoding space. Since no positional-set of cardinality greater than 1 will appear there, we can make use of these don't care code points in the positional-set space.

For example, the computations involved in Equations 4.1 to 4.2 manipulate a product of two singleton states (y, z). The don't care condition with respect to this pair of singletons is captured by:

$$DC(y,z) = \neg Tuple_0(y) \cdot \neg Tuple_1(y) + \neg Tuple_0(z) \cdot \neg Tuple_1(z)$$

and can be used to simplify the BDD computation of these sets using generalized cofactor.

4.7 Experimental Results

We implemented the algorithms described in previous sections in a program called ISM, an acronym for Implicit State Minimizer. We ran ISM on different suites of FSM's. We report results on the following suites of FSM's. They are:

1. the MCNC benchmark and other examples,

2. FSM's generated by a synthesis procedure for asynchronous logic [46],
3. FSM's from learning I/O sequences [23],
4. FSM's from synthesis of interacting FSM's [82],
5. FSM's with exponentially many prime compatibles,
6. FSM's with many maximal compatibles, and
7. randomly generated FSM's.

Each suite has different features with respect to state minimization. We discuss features of the experiments and results in different subsections. Comparisons are made with STAMINA [64], a program that represents the state-of-art for state minimization based on explicit techniques. The program STAMINA was run with the option -P to compute all prime compatibles.

For each example, we report the number of states in the original ISFSM, the number of maximal compatibles if applicable, the number compatibles, the number of prime compatibles, the number of non-essential prime compatibles if applicable, and the run time for our implicit algorithm ISM and that for the explicit algorithm STAMINA. All run times are reported in CPU seconds on a DECstation 5000/260 with 440 Mb of memory. The CPU run time refers to the computation of the prime compatibles only.

4.7.1 FSM's from MCNC Benchmark and Others

Table 4.1 reports the results from the MCNC benchmark and from other academic and industrial benchmarks available to us. Most examples have a small number of prime compatibles, with the exception of *ex2* and *green*. The running times of ISM are worse than those of STAMINA, especially in those cases where there are very few compatibles in the number of states (*squares* is the most striking example). In those cases an explicit algorithm is sufficient to get a quick answer and it may be faster than an implicit one. The reason is that ISM manipulates relations having a number of variables linearly proportional to the number of states. When there are many states and few compatibles, the purpose of ISM is defeated and its representation becomes inefficient. But when the number of primes is not negligible as in *ex2* and *green*, ISM ran as fast or faster than STAMINA.

The question now arises of how it is realistic to expect such examples in logic design applications. One could object that the examples of Table 1 show that hand-designed FSM's can

be handled very well by an existing state-of-art program like STAMINA. If this can be true for usual hand-designed FSM's, we argue that there are FSM's produced in the process of logic synthesis of real design applications that generate large sets of compatibles exceeding the capabilities of programs based on an explicit enumeration. The examples of Table 2 are such a case. They are FSM's produced as intermediate stages of an asynchronous logic design procedure and their minimization requires computing very large sets of compatibles. Another case is the one reported in Table 3, referring to the synthesis of finite state machines consistent with a collection of I/O learning examples.

machine	# states	# max compat.	# compat.	# prime compat.	$\#\mathcal{NEPC}$	CPU time (sec) ISM	CPU time (sec) STAMINA
arbseq	94	2	96	9	3	12	0
bbsse	16	11	97	13	0	0	0
beecount	7	4	11	7	5	0	0
ex1	20	2	22	19	1	1	0
ex2	19	36	2925	1366	1366	7	13
ex3	10	10	195	91	91	0	0
ex5	9	6	81	38	38	0	0
ex7	10	6	135	57	57	0	0
fsm1	256	47	302	208	0	83	0.6
green	54	524	1234	524	524	90	125
lion9	9	5	20	5	2	0	0
mark1	15	12	41	18	11	0	0
scf	121	12	1201	175	87	22	0
squares	371	45	473	307	0	731	1
tbk	32	16	48	48	48	3	1
tma	20	15	35	20	4	1	0
train11	11	5	85	17	15	0	0
viterbi	68	5	329	57	3	6	0

Table 4.1: The MCNC benchmark and others.

4.7.2 FSM's from Asynchronous Synthesis

Table 4.2 reports the results of a benchmark of FSM's generated as intermediate steps of an asynchronous synthesis procedure [46]. STAMINA ran out of memory on the examples *vmebus.master.m, isend, pe-rcv-ifc.fc, pe-send-ifc.fc*, while ISM was able to complete them. These examples (with the exception of *vbe4a*) have a number of primes below a thousand. To explain

this data reported in Table 4.2, we notice that in order to compute the prime compatibles, every compatible has to be generated by STAMINA too. The compatibles of the FSM's of this benchmark are usually of large cardinality and therefore their enumeration causes a combinatorial explosion. So the huge size of the set of compatibles accounts for the large running times and/or out-of-memory failures. About the behavior of ISM, we underline that the running times track well with the size of the set of compatibles and when both programs complete, they are usually well below those of STAMINA (*pe-rcv-ifc.fc.m, pe-send-ifc.fc.m, vbe4a*). For asynchronous synthesis, a more appropriate formulation of exact state minimization requires the computation of all compatibles or at least of prime compatibles and a different set-up of the covering problem [46].

machine	# states	# max compat.	# compat.	# prime compat.	$\#\mathcal{NEPC}$	CPU time (sec) ISM	CPU time (sec) STAMINA
alex1	42	787	55928	787	787	24	16
future	36	49	7.929e8	49	49	3	0
future.m	28	16	2.621e7	16	16	2	0
intel_edge.dummy	28	120	9432	396	396	37	3
isend	40	128	22207	480	480	13	spaceout
isend.m	20	15	22207	19	19	1	0
mp-forward-pkt	20	1	1.048e6	1	0	0	0
nak-pa	56	8	4.741e15	8	8	9	0
nak-pa.m	18	8	44799	8	8	1	0
pe-rcv-ifc.fc	46	28	1.528e11	148	148	18	spaceout
pe-rcv-ifc.fc.m	27	18	1.793e6	38	38	3	147
pe-send-ifc.fc	70	39	5.071e17	506	506	571	spaceout
pe-send-ifc.fc.m	26	6	8.978e6	23	22	3	312
ram-read-sbuf	36	2	3.006e10	2	0	2	0
sbuf-ram-write	58	24	1.433e6	24	24	14	0
sbuf-ram-write.m	24	12	1.433e6	12	12	2	0
sbuf-send-ctl	20	10	81407	10	10	0	0
sbuf-send-pkt2	21	2	622591	2	0	0	0
vbe4a	58	2072	1.756e12	2072	2072	109	167
vbe4a.m	22	13	73471	13	13	2	0
vbe6a.m	16	8	527	8	4	1	0
vmebus.master.m	32	10	5.049e7	28	28	16	spaceout

Table 4.2: Asynchronous FSM benchmark.

4.7.3 FSM's from Learning I/O Sequences

Table 4.3 and Figure 4.4 show the results of running a parametrized set of FSM's constructed to be compatible with a given collection of examples of input/output traces [23]. These machines exhibit very large number of compatibles.

Here ISM shows all its power compared to STAMINA, both in terms of number of computed prime compatibles and running time. STAMINA runs out of memory on the examples from *threer.35* and *fourr.30* onwards and, when it completes, it takes close to two order of magnitude more time than ISM.

machine	# state	# compat.	# prime compat.	CPU time (sec) ISM	STAMINA
threer.10	11	671	112	0	0
threer.20	21	16829	3936	1	159
threer.30	31	97849	33064	21	1344
threer.40	41	1.456e6	529420	75	spaceout
threer.55	55	3.622e7	1.555e7	1273	spaceout
fourr.10	11	2047	1	0	0
fourr.20	21	42193	12762	2	217
fourr.30	31	1.346e6	542608	20	spaceout
fourr.40	41	5.266e9	2.388e9	105	spaceout
fourr.50	51	3.643e7	1.696e7	198	spaceout
fourr.60	61	1.052e10	5.021e9	*18181	spaceout
fourr.70	71	9.621e10	4.524e10	*22940	spaceout

Table 4.3: FSM's from learning I/O sequences.

4.7.4 FSM's from Synthesis of Interacting FSM's

It has been reported by Rho and Somenzi in [65] that the exact state minimization of the driven machine of a pair of cascaded FSM's is equivalent to the state minimization of an ISFSM that requires the computation of prime compatibles.

The examples *ifsm0*, *ifsm1*, *ifsm2* come from a set of FSM's produced by FSM optimization, using the input don't care sequences induced by a surrounding network of FSM's [82]. They exhibit often large number of compatibles and prime compatibles, as shown in Table 4.4. For such cases, the run times of the implicit algorithm ISM are shorter than those by STAMINA.

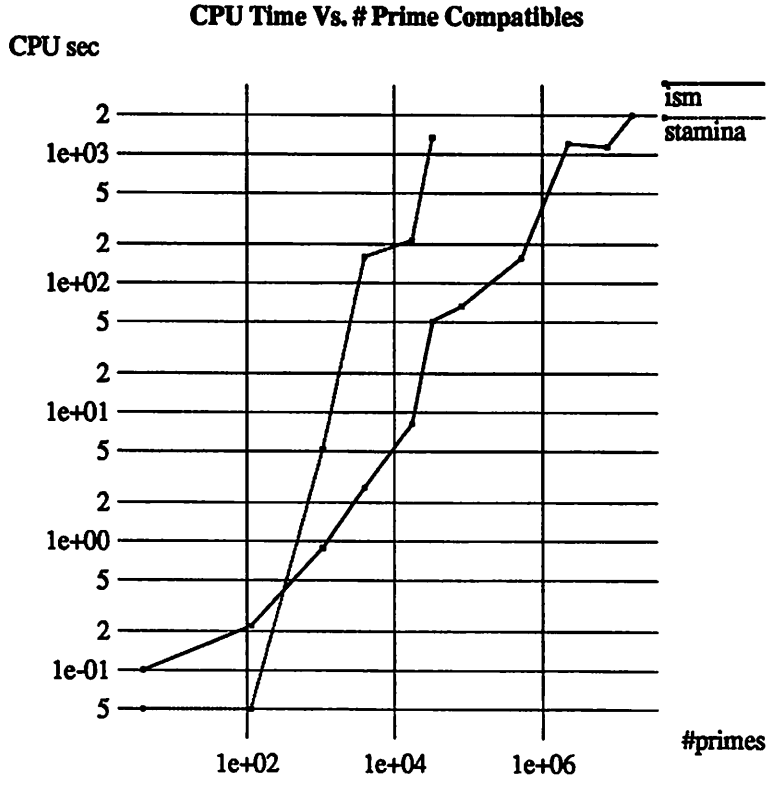

Figure 4.4: Comparison between ISM and STAMINA on learning I/O sequences benchmark.

machine	# state	# compat.	# prime compat.	CPU time (sec) ISM	STAMINA
ifsm0	38	1064973	18686	43	4253
ifsm1	74	43006	8925	25	466
ifsm2	150	497399	774	267	356

Table 4.4: Examples from synthesis of interacting FSM's.

4.7.5 FSM's with Exponentially Many Prime Compatibles

In the previous examples, the number of prime compatibles is not large compared to the number of states. A natural question to ask is whether there are FSM's that generate a large number of prime compatibles with respect to the number of states. We were able to construct a suite of FSM's where the number of prime compatibles is exponential in the number of states.

Rubin gave in [66] a sharp upper bound for the number of maximal compatibles of an ISFSM. He showed that $M(n)$, the maximum number of maximal compatibles over all ISFSM's with $n > 1$ states, is given by $M(n) = i.3^m$, if $n = 3.m + i$. The proof of this counting statement is based on the construction of a family of incompatibility graphs $I(n)$ parametrized in the number of states [4]. Each $I(n)$ is composed canonically of a number of connected components. Each maximal compatible contains exactly one state from each connected component of the graph. The number of such choices is shown to be $M(n)$.

The proof of the theorem does not exhibit an FSM that has a canonical incompatibility graph. Based on the construction of the incompatibility graphs given in the paper, we have built a family $F(n)$ of ISFSM's [5] (parametrized in the number of states n) that have a number of maximal compatibles in the order of $3^{(n/3)}$ and a number of prime compatibles in the order of $2^{(2n/3)}$. $F(n)$ has 1 input and $n/3$ outputs. Each machine F is derived from a non-connected state transition graph whose component subgraphs F_i are defined on the same input and outputs. Each subgraph F_i has 3 states $\{s_{i0}, s_{i1}, s_{i2}\}$ and 3 specified transitions $\{e_{i0} = (s_{i0}, s_{i1}), e_{i1} = (s_{i1}, s_{i2}), e_{i2} = (s_{i2}, s_{i0})\}$. Each transition under the input set to 1 asserts all outputs to $-$, with the exception that e_{i0} and e_{i1} assert the i-th output to 0 and e_{i2} asserts the i-th output to 1. Under the input set to 0, the transitions are left unspecified.

Table 4.5 and Figure 4.5 show the results of running increasingly larger FSM's of the family. While ISM is able to generate sets of prime compatibles of cardinality up to 2^{1500} with reasonable running times, STAMINA, based on an explicit enumeration runs out of memory soon (and where it completes, it takes much longer).

4.7.6 FSM's with Many Maximal Compatibles

Table 4.6 shows the results of running some examples from a set of FSM's constructed to have a large number of maximal compatibles. The examples *jac4, jc43, jc44, jc45, jc46, jc47* are

[4] The incompatibility graph of an ISFSM F is a graph whose nodes are the states of F, with an undirected arc between two nodes s and t iff s and t are incompatible.

[5] Called *rubin* followed by n in the table of results.

machine	# states	# max compat.	# compat.	# prime compat.	$\#\mathcal{NEPC}$	CPU time (sec) ISM	STAMINA
rubin12	12	3^4	2^8-1	2^8-1	2^8-1	0	4
rubin18	18	3^6	$2^{12}-1$	$2^{12}-1$	$2^{12}-1$	1	751
rubin24	24	3^8	$2^{16}-1$	$2^{16}-1$	$2^{16}-1$	1	spaceout
rubin300	300	3^{100}	$2^{200}-1$	$2^{200}-1$	$2^{200}-1$	256	spaceout
rubin600	600	3^{200}	$2^{400}-1$	$2^{400}-1$	$2^{400}-1$	1995	spaceout
rubin900	900	3^{300}	$2^{600}-1$	$2^{600}-1$	$2^{600}-1$	6373	spaceout
rubin1200	1200	3^{400}	$2^{800}-1$	$2^{800}-1$	$2^{800}-1$	17711	spaceout
rubin1500	1500	3^{500}	$2^{1000}-1$	$2^{1000}-1$	$2^{1000}-1$	42674	spaceout
rubin1800	1800	3^{600}	$2^{1200}-1$	$2^{1200}-1$	$2^{1200}-1$	78553	spaceout
rubin2250	2250	3^{750}	$2^{1500}-1$	$2^{1500}-1$	$2^{1500}-1$	271134	spaceout

Table 4.5: Constructed FSM's.

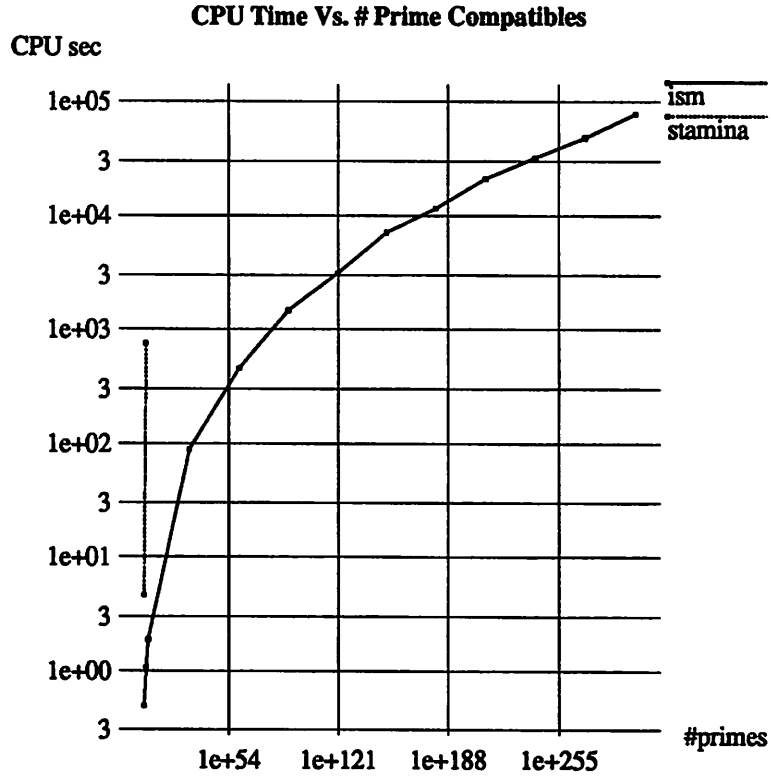

Figure 4.5: Comparison between ISM and STAMINA on constructed FSM's.

due to R. Jacoby and have been kindly provided by J.-K. Rho of University of Colorado, Boulder. The example *lavagno* is from asynchronous synthesis as those reported in Section 4.7.2. For these examples the program STAMINA was run with the option -M to compute all maximal compatibles. While ISM could complete on them in reasonable running times, STAMINA could not complete on *jac4* and completed the other ones with running times exceeding those of ISM by one or two order of magnitudes. Notice that ISM could also compute the set of all compatibles even though the computation of prime compatibles cannot be carried to the end while STAMINA failed on both. The prime compatibles for these examples could not be computed by either program.

machine	# states	# max compat.	# compat.	# prime compat.	CPU time (sec) ISM	STAMINA
jac4	65	3.859e6	4.159e7	?	34	spaceout
jc43	45	82431	1.556e6	?	13	7739
jc44	55	4785	7.584e9	?	20	662
jc45	40	17323	480028	?	10	1211
jc46	42	26086	1.153e6	?	11	2076
jc47	51	397514	1.120e7	?	19	41297
lavagno	65	47971	9.163e6	?	163	40472

Table 4.6: FSM's with many maximals.

4.7.7 Randomly Generated FSM's

We investigated also whether randomly generated FSM's have a large number of prime compatibles. A program was written to generate random FSM's [6]. A small percentage of the randomly generated FSM's were found to exhibit this behavior. Table 4 shows the results of running ISM and STAMINA on some interesting examples with a large number of primes. Again only ISM could complete the examples exhibiting a large number of primes.

4.7.8 Experiments Comparing BDD and ZBDD Sizes

As mentioned before, the objects we manipulate in our implicit algorithms are sets and relations of positional-sets. A characteristic of our representations is that the positional-sets (e.g., compatibles) are usually sparse sets. Minato in [57] showed that Zero-suppressed BDD is a good

[6] Parameters: number of states, number of inputs, number of outputs, don't care output percentage, don't care target state percentage.

machine	# states	# max compat.	# compat.	# prime compat.	$\#\mathcal{NEPC}$	CPU time (sec) ISM	CPU time (sec) STAMINA
fsm15.232	14	4	7679	360	360	2	23
fsm15.304	14	2	12287	954	954	1	85
fsm15.468	13	2	4607	772	772	1	16
fsm15.897	15	2	20479	617	616	0	50
ex2.271	19	2	393215	96383	96382	21	spaceout
ex2.285	19	2	393215	121501	121500	13	spaceout
ex2.304	19	2	393215	264079	264079	93	spaceout
ex2.423	19	4	204799	160494	160494	102	spaceout
ex2.680	19	2	327679	192803	192803	151	spaceout

Table 4.7: Random FSM's.

representation of sets of sparse sets. As a preliminary investigation of the effectiveness of a ZBDD-based algorithm for exact state minimization, we convert some key BDD objects into ZBDD's and observe the change in the number of nodes used.

Experiments are performed on machines which require non-trivial covering steps for state minimization. During state minimization of each machine, ZBDD's are generated from BDD's representing the set of compatibles $\mathcal{C}$, the set of prime compatibles $\mathcal{PC}$ (which is usually also the set of column labels), and the set of row labels R. From the rightmost two columns of the above table, the conversion routines between ZBDD and BDD seem to execute fast enough to allow for algorithms that switches between BDD and ZBDD representations and manipulations.

Out of the 18 examples, 9 of them have smaller ZBDD's than BDD's for representing $\mathcal{C}$, 13 of them have smaller ZBDD's than BDD's for representing $\mathcal{PC}$ (the remaining 5 examples are all randomly generated machines), and 18 of them have smaller ZBDD representations of R. Disregarding the second set of examples, ex2.* * *, which are randomly generated machines, the comparison shows that ZBDD's are usually smaller than BDD's and are always comparably close for the exceptional cases.

When converting BDD's to ZBDD's, the most reduction in sizes occurs for the representation of R, and less for $\mathcal{PC}$, and the least for $\mathcal{C}$. A possible explanation is that $\mathcal{PC}$ is more sparse a set than $\mathcal{C}$ because all state sets that are in $\mathcal{PC}$ are in $\mathcal{C}$ but not vice versa. A similar explanation also applies between R and $\mathcal{PC}$. Each row label in R consist of two parts: a positional-set representing a prime compatible, and another positional-set representing its class set. As we don't expect much

machine	Compatibles # nodes in BDD	Compatibles # nodes in ZBDD	Prime Compat. # nodes in BDD	Prime Compat. # nodes in ZBDD	table→R # nodes in BDD	table→R # nodes in ZBDD	table→R CPU Time (sec) bdd2zbdd	table→R CPU Time (sec) zbdd2bdd
alex1	2562	1203	1243	415	167	43	0.00	0.00
ex2.271	21	51	236	286	5646	4307	0.44	0.27
ex2.285	22	38	461	471	39	2	0.00	0.00
ex2.304	22	38	787	826	56928	34263	5.66	3.19
ex2.423	29	66	675	782	40857	27060	4.66	3.32
ex2.680	23	41	1819	1953	57446	40277	8.06	4.25
ex2	161	108	222	148	4421	1521	0.18	0.16
ex3	28	33	45	41	584	290	0.02	0.01
ex5	34	31	33	26	269	111	0.01	0.00
ex7	26	30	41	39	322	159	0.01	0.01
green	197	78	194	62	211	54	0.01	0.01
keyb_ex2	581	173	1260	324	14539	2053	0.77	0.58
s386_keyb	320	188	404	175	6386	1270	0.25	0.22
room4.16	75	79	201	159	1213	501	0.05	0.04
room4.20	122	134	458	407	2863	1431	0.16	0.11
room3.20	113	110	265	228	1717	790	0.07	0.05
room3.25	220	185	544	402	3922	1567	0.18	0.12
room3.30	637	473	1205	785	9391	3389	0.59	0.38

Table 4.8: Comparison between BDD and ZBDD sizes.

sharing between different class sets, the set of row labels (whose variable support is twice of $\mathcal{PC}$ and $\mathcal{C}$) should be the most sparse set.

Chapter 5

Binate Covering

5.1 Introduction

At the core of the exact solution of various logic synthesis problems lies often a so-called covering step that requires the choice of a set of elements of minimum cost that *cover* a set of ground items, under certain conditions. Prominent among these problems are the covering steps in the Quine-McCluskey procedure for minimizing logic functions, selection of a set of encodeable generalized prime implicants, state minimization of finite state machines, technology mapping and Boolean relations. Let us review first how covering problems are defined formally.

Suppose that a set $S = \{s_1, \ldots, s_n\}$ is given. The cost of selecting s_i is c_i where $c_i \geq 0$. By associating a binary variable x_i to s_i, which is 1 if s_i is selected and 0 otherwise, the binate covering problem (BCP) can be defined as finding $S' \subseteq S$ that minimizes

$$\sum_{i=1}^{n} c_i x_i,$$

subject to the constraint

$$A(x_1, x_2, \ldots, x_n) = 1,$$

where A is a Boolean function, sometimes called the constraint function. The constraint function specifies a set of subsets of S that can be a solution. No structural hypothesis is made on A. Binate refers to the fact that A is in general a binate function (a function is binate if it has at least a binate variable). BCP is the problem of finding an onset minterm of A that minimizes the cost function (i.e., a solution of minimum cost of the Boolean equation $A(x_1, x_2, \ldots, x_n) = 1$).

If A is given in product-of-sums form, finding a satisfying assignment is exactly the problem SAT, the prototypical NP-complete problem [26]. In this case it also possible to write A

as an array of cubes (that form a matrix with coefficients from the set $\{0, 1, 2\}$). Each variable of A is a column and each sum (or clause) is a row and the problem can be interpreted as one of finding a subset C of columns of minimum cost, such that for every row r_i, either

1. $\exists j$ such that $a_{ij} = 1$ and $c_j \in C$, or

2. $\exists j$ such that $a_{ij} = 0$ and $c_j \notin C$.

In other words, each clause must be satisfied by setting to 1 a variable appearing in it in the positive phase or by setting to 0 a variable appearing in it in the negative phase. In a unate covering problem, the coefficients of A are restricted to the values 1 and 2 and only the first condition must hold. In this chapter, we shall consider the minimum binate covering problem where A is given in product-of-sums form. In this case, the term *covering* is fully justified because one can say that the assignment of a variable to 0 or 1 covers some rows that are satisfied by that choice. The product-of-sums A is called covering matrix or covering table.

As an example of binate covering formulation of a well-known logic synthesis problem, consider the problem of finding the minimum number of prime compatibles that are a minimum closed cover of a given FSM. A binate covering problem can be set up as described in Section 5.7, where each column of the table is a prime compatible and each row is one of the covering or closure clauses of the problem [28]. There are as many covering clauses as states of the original machine and each of them requires that a state is covered by selecting any of the prime compatibles in which it is contained. There are as many closure clauses as prime compatibles and each of them states that if a given prime compatible is selected, then for each implied class in its corresponding class set, one of the prime compatibles containing it must be chosen too.

In the matrix representation, table entry (i, j) is 1 or 0 according to the phase of the literal corresponding to prime compatible j in clause i; if such a literal is absent, the entry is 2.

Various techniques have been proposed to solve binate covering problems. A class of them [10, 42] are branch-and-bound techniques that build explicitly the table of the constraints expressed as product-of-sum expressions and explore in the worst-case all possible solutions, but avoid the generation of some of the suboptimal solutions by a clever use of reduction steps and bounding of search space for solutions. We will refer to these methods as explicit.

A second approach [50] formulates the problem with Binary Decision Diagrams (BDD's) and reduces finding a minimum cost assignment to a shortest path computation. In that case the number of variables of the BDD is the number of columns of the binate table.

Recently, a mixed technique has been proposed in [35]. It is a branch-and-bound algorithm, where the clauses are represented as a conjunction of BDD's. The usage of BDD's leads to an effective method to compute a lower bound on the cost of the solution.

Notice that unate covering is a special case of binate covering. Therefore techniques for the latter solve also the former. In the other direction, exact state minimization, a problem naturally formulated as a binate covering problem, can be reduced to a unate covering problem, after the generation of irredundant prime closed sets [72]. But there is a catch here: the cost function is not any more additive, so that the reduction techniques so convenient to solve covering problems, are not any more applicable as they are.

In this chapter, we are interested in exact solutions of binate covering. Existing explicit methods do quite well on small and medium-sized examples, but fail to complete on larger ones. The reason is that either they cannot build the binate table because the number of rows and columns is too large, or that the branch-and-bound procedure would take too long to complete. For the approach of building a BDD of the constraint function and computing the shortest path fails, it fails when the number of variables (i.e., columns) is too large because it is likely that a BDD with many thousands of variables will blow up.

The crux of the matter, when explicit techniques fail, is that we are representing and manipulating sets that are too large to be exhaustively listed and operated upon. Fortunately we know of an alternative way to represent and manipulate sets: it is by defining the set over an appropriate Boolean space (i.e.; encoding the elements of the set), associating to it a Boolean characteristic function and then representing this function by a binary decision diagram (BDD). Since now on, by BDD of a set we will denote the BDD of the characteristic function of the set over an appropriate Boolean space. As defined in Section 3.3, a BDD [11, 6] is a canonical directed acyclic graph data structure that represents logic functions. The items that a BDD can represent are determined by the number of paths of the BDD, while the size of the BDD is determined by the number of nodes of the DAG. There is no monotonic relation between the size of a BDD and the number of elements that it represents. It is an experimental fact that often very large sets, that cannot be represented explicitly, have a compact BDD representation. Set operations are easily turned into Boolean operations on the corresponding BDD's. So we can manipulate sets by a series of BDD operations (Boolean connectives and quantifications) with a complexity depending on the sizes of the manipulated BDD's and not on the cardinality of the sets that are represented. The hope here is that complex set manipulations have as counterparts Boolean propositions that can be represented with compact BDD's. Of course, this is not always the case and it may happen that an intermediate

BDD computation, in a sequence of operations leading to a set, blows up. The name of the game is a careful analysis of how propositional sentences can be transformed into logically equivalent ones, that can be computed more easily with BDD manipulations. Special care must be exercised with quantifications, that bring more danger of BDD blowups. All of this goes often under the name of implicit representations and computations.

The previous insight has already been tested in a series of applications. Research at Bull [16] and U.C. Berkeley [80] produced powerful techniques for implicit enumeration of subsets of states of a Finite State Machine (FSM). Later work at Bull [19, 48] has shown how implicants, primes and essential primes of a two-valued or multi-valued function can also be computed implicitly. Reported experiments show a suite of examples where all primes could be computed, whereas explicit techniques implemented in ESPRESSO [7] failed to do so. Finally, the fixed-point dominance computation in the covering step of the Quine-McCluskey procedure has been made implicit in current work [18, 79]. The experiments reported show that the cyclic core of all logic functions of the ESPRESSO benchmark can be successfully computed. For some of them ESPRESSO failed the task.

Last but not the least, we have shown in Chapter 4 how all prime compatibles of an FSM can be computed implicitly. In some cases, their number is exponential in the number of states (the largest recorded number is 2^{1500}). Once we have them, we must solve a binate covering problem to choose a minimum closed cover. Of course, we cannot build explicitly a table of such dimensions and solve it (this would defeat the purpose of computing implicitly prime compatibles in the first place). So it is necessary to extend implicit techniques to the solution of the binate covering problem. An implicit binate table solver would also be required to solve implicitly other problems in sequential synthesis that we care about. One is the selection of a set of encodeable generalized prime implicants (GPI's), as defined in [22, 50]. It is not feasible to generate GPI's and to set up a related binate covering table by explicit techniques on non-trivial examples. Using techniques as in [48], GPI's can be generated implicitly. An implicit binate table solver is therefore needed there too. Notice that potential applications include unate covering problems, such as selecting the minimum number of encoding-dichotomies that satisfy a set of encoding constraints [71].

This chapter describes an implicit formulation of the binate covering problem and presents an implementation. The implicit binate solver has been tested in the case of state minimization of ISFSM's (Chapter 4) and pseudo NDFSM's (Chapters 6 and 7), and also for optimum selection of an encodable set of generalized prime implicants for state assignment [81]. The reported experiments show that implicit techniques have pushed the frontier of instances where binate covering problems

can be solved exactly, resulting in better optimizations possible in key steps of sequential logic synthesis.

In the following sections, we will review the known algorithms to solve covering problems and then we will describe a new branch-and-bound algorithm based on implicit computations. The remainder of the chapter is organized as follows. We have defined the minimum cost binate covering problem in this section. In Section 5.2, we will compare this problem with 0-1 integer linear programming. The branch-and-bound scheme will be introduced in Section 5.3 which has be used in explicit binate covering algorithms summarized in Section 5.4. In Section 5.5, we survey the classical reduction rules used in explicit algorithms. Our implicit binate covering algorithm is then introduced in Section 5.6 and its program input, an implicit table representation, is described in Section 5.7. Section 5.8 illustrates how reduction techniques can be implicitized. Other kinds of implicit table manipulations are introduced in Section 5.9. Finally, we shall give experimental results in Section 5.10, for solving of the state minimization problem of ISFSM's.

5.2 Relation to 0-1 Integer Linear Programming

There is an intimate relation between 0-1 integer linear programming (ILP) and binate covering problem (BCP). For every instance of ILP, there is an instance of BCP with the same feasible set (i.e., satisfying solutions) and therefore with the same optimum solutions and vice versa. As an example, the integer inequality constraint

$$3x_1 - 2x_2 + 4x_3 \geq 2,$$

with $0 \leq x_1, x_2, x_3 \leq 1$ corresponds to the Boolean equality constraint

$$x_1\overline{x_2} + x_3 = 1,$$

that can be written in product-of-sums form as:

$$(x_1 + x_3)(\overline{x_2} + x_3) = 1.$$

Given a problem instance, it is not clear a-priori which formulation is better. It is an interesting question to characterize the class of problems that can be better formulated and solved with one technique or the other.

As an example of reduction from ILP to BCP, a procedure (taken from [35]) that derives the Boolean expression corresponding to $\sum_{j=1}^{n} w_j.x_j \geq T$ is shown in Figure 5.1.
The idea of the recursion relies on the observation that:

```
LI_to_BDD(I) {
    let I be ∑_{j=1}^{n} w_j · x_j ≥ T
    if (max(I) < T) return 0
    if (min(I) ≥ T) return 1
    i = ChooseSplittingVar(I)
    I^1 = (∑_{j≠i} w_j · x_j ≥ T − w_i)
    I^0 = (∑_{j≠i} w_j · x_j ≥ T)
    f_1 = LI_to_BDD(I^1)
    f_0 = LI_to_BDD(I^0)
    return f = x_i · f_1 + x̄_i · f_0
}
```

Figure 5.1: Transformation from linear inequality to Boolean expression.

1. $f = 0$ if and only if $max(I) = \sum_{w_i>0} w_i < T$;

2. $f = 1$ if and only if $min(I) = \sum_{w_i<0} w_i \geq T$;

When neither case occurs, the two subproblems I^1 and I^0, obtained by setting the splitting variable x_i to 1 and 0 respectively, are solved recursively.

5.3 Branch-and-Bound as a General Technique

Branch-and-bound constructs a solution of a combinatorial optimization problem by successive partitioning of the solution space. The *branch* refers to this partitioning process; the *bound* refers to lower bounds that are used to construct a proof of optimality without exhaustive search. A set of solutions can be represented by a node in a search tree of solutions, and it is partitioned in mutually exclusive sets. Each subset in the partition is represented by a child of the original node. In this way, a computation tree is built. An algorithm that computes a lower bound on the cost of any solution in a given subset allows to stop further searches from a given node, if the best cost found so far is smaller than the cost of the best solution that can be obtained from the

node (lower bound computed at the node). In this case the node is killed and therefore none of its children needs to be searched; otherwise it is alive.

If we can show at any point that the best descendant of a node y is at least as good as the best descendant of node x, then we say that y *dominates* x, and y can kill x.

Figure 5.2 shows the classical algorithm [59]. An *activeset* holds the live nodes at any point. A variable U is an upper bound on the optimal cost (cost of the best complete solution obtained at any given time). The branching process needs not produce only two children of a given node, but any finite number.

We will see in the next section that BCP can be solved by the following recursive equation

$$BCP(M_f) = BestSolution(BCP(M_{f_{x_i}}) \cup \{x_i\}, BCP(M_{f_{\overline{x_i}}}))$$

where M_f is the binate table that corresponds to a function in product-of-sum form f, and $BCP(M_{f_{x_i}})$ (respectively, $BCP(M_{f_{\overline{x_i}}})$) is the subproblem expressed by the function f_{x_i} (respectively, $f_{\overline{x_i}}$). $BCP(M_f)$ returns an onset minterm of f that minimizes the cost function.

The previous equation can potentially generate an exponential number of subproblems, but powerful dominance and bounding techniques as well as good branching heuristics help in keeping the combinatorial explosion under control.

5.4 A Branch-and-Bound Algorithm for Minimum Binate Covering

We will survey in this section a branch-and-bound solution of minimum cost binate covering. This technique has been described in [29, 28, 9, 10], and implemented in successful computer programs [67, 64, 76]. The branch-and-bound solution of minimum binate covering is based on a recursive procedure. A run of the algorithm can be described by its computation tree. The root of the computation tree is the input of the problem, an edge represents a call to *sm_mincov*, an internal node is a reduced input. A leaf is reached when a complete solution is found or the search is bounded away. From the root to any internal node there is a unique path, that is the current path for that node. In the sequel, we will describe in detail the binary recursion procedure. The presentation will refer to the pseudo-code *sm_mincov*, shown at the end of this subsection.

5.4.1 The Binary Recursion Procedure

The inputs to the algorithm are:

```
branch_and_bound() {
    activeset = original problem
    U = ∞
    currentbest = anything
    while (activeset is not empty) {
        choose a branching node k ∈ activeset
        remove node k from activeset
        generate the children of node k: child i = 1, ..., n_k
            and the corresponding lower bounds z_i
        for i = 1 to n_k {
            if (z_i ≥ U) kill child i
            else if (child i is a complete solution) {
                U = z_i
                currentbest = child i
                else add child i to activeset
            }
        }
    }
}
```

Figure 5.2: Structure of branch-and-bound.

- a covering matrix M;
- a current-path partial solution *select* (initially empty);
- a row of non-negative integers *weight*, whose i-th element is the cost or weight of the i-th column of M;
- a lower bound *lbound* (initially set to 0), which is a monotonic increasing quantity along each path of the computation tree equal to the cost of the partial solution on the current path;
- an upper bound *ubound* (initially set to the sum of weights of all columns in M), which is the cost of the best overall complete solution previously obtained (a globally monotonic decreasing quantity);

The output is the best column cover for input M extended from the partial solution *select* along the current path, called best current solution, if this solution costs less than *ubound*. An empty solution is returned if a solution cannot be found which beats *ubound* or an infeasibility is detected. By infeasibility, it is meant the case when no satisfying assignment of the product of clauses exists. Even though the initial problem in a typical logic synthesis application has usually at least a solution, some subproblems in the branch and bound tree may be infeasible. When *sm_mincov* is called with an empty partial solution *select* and initial *lbound* and *ubound*, it returns a best global solution.

The algorithm calls first a procedure *sm_reduce* that applies to M essential column detection and dominance reductions. The type of domination operations and the way in which they are applied are the subject of Section 5.5. Another more complex reduction criterion (Gimpel's rule) can also be applied (see Subsection 5.5.12). These reduction operations delete from M some rows, columns and entries. What is left after reduction is called a cyclic core. The final goal is to get an empty cyclic core. The value of the lower bound is updated using a maximal independent set computation (see Subsection 5.4.3). If no bounding is possible and the reductions do not suffice to solve completely the problem, a partition of the reduced problem into disjoint subproblems is attempted (see Subsection 5.4.2) and each of them is solved recursively. When everything fails, binary recursion is performed by choosing a branch column (see Subsection 5.4.4). Solutions to the subproblems obtained by including the chosen column in the covering set or by excluding it from the covering set are computed recursively and the best solution is kept (the second recursion is skipped if the solution to the first one matches the updated lower bound).

The procedure *sm_mincov* returns when:

```
sm_mincov(M, select, weight, lbound, ubound) {
    /* Apply row dominance, column dominance, and select essentials */                    (1)
    if (!sm_reduce(M, select, weight, ubound)) return empty_solution
    /* See if Gimpel's reduction technique applies */                                     (2)
    if (gimpel_reduce(M, select, weight, lbound, ubound, &best)) return best
    /* Find lower bound from here to final solution by independent set */                 (3)
    indep = sm_maximal_independent_set(M, weight)
    /* Make sure the lower bound is monotonically increasing */                           (4)
    lbound_new = max(cost(select) + cost(indep), lbound)
    /* Bounding based on no better solution possible */                                   (5)
    if (lbound_new ≥ ubound) best = empty_solution
    else if (M is empty) { /* New best solution at current level */                       (6)
        best = solution_dup(select)
    } else if (sm_block_partition(M, &M1, &M2) gives non-trivial bi-partitions) {         (7)
        best1 = sm_mincov(M1, select, weight, 0, ubound − cost(select))                   (8)
        /* Add best solution to the selected set */                                       (9)
        if (best1 = empty_solution) best = empty_solution
        else {                                                                            (10)
            select = select ∪ best1
            best = sm_mincov(M2, select, weight, lbound_new, ubound)
        }
    } else { /* Branch on cyclic core and recur */                                       (11)
        branch = select_column(M, weight, indep)
        select1 = solution_dup(select) ∪ branch
        let M_branch be the reduced table assuming branch column is not in solution       (12)
        best1 = sm_mincov(M_branch, select, weight, lbound_new, ubound)
        /* Update the upper bound if we found a better solution */                        (13)
        if (best1 ≠ empty_solution) and (ubound > cost(best1)) ubound = cost(best1)
        /* Do not branch if lower bound matched */                                        (14)
        if (best1 ≠ empty_solution) and (cost(best1) = lbound_new) return best1
        let M_¬branch be the reduced table assuming branch column not in solution         (15)
        best2 = sm_mincov(M_¬branch, select, weight, lbound_new, ubound)
        best = best_solution(best1, best2)
    }
    return best
}
```

Figure 5.3: Detailed branch-and-bound algorithm.

- The cost of a partial solution, found by adding essential columns to *select*, is more than *ubound* or infeasibility is detected when applying the domination rules (line 1). An empty solution is returned.

- The best current solution is found by applying Gimpel's reduction technique (line 2). Since *gimpel_reduce* calls recursively *sm_mincov*, an empty solution could be returned too.

- The updated lower bound, determined by adding to *lbound* the cost of the essential primes and of the maximal independent set, is not less than *ubound* (line 5). An empty solution is returned.

- There is no cyclic core and we are not in the previous case. The best current solution is found by updating *select* with the new essential and unacceptable columns (line 6).

- The best current solution is found by partitioning the problem (line 7). The procedure *sm_mincov* is called recursively on two smaller covering matrices determined by *sm_block_partition* (line 8 and 10). An empty solution can be returned by either recursive call. If the first call to *sm_mincov* returns an empty solution, the second one is not invoked (line 9). If neither call returns empty, each contributes its returned value to the current solution.

- A branching column is chosen and *sm_mincov* is called recursively with the branch column in the covering set (line 12). If the recursive call of *sm_mincov* returns a non-empty solution that matches the current lower bound (*lbound_new*), that solution is returned as the current solution (line 14). If the cost of the current solution is less than *ubound*, *ubound* is updated, i.e., the current solution is also the best global solution (line 13).

- As in the previous case, but *sm_mincov* is called recursively with the branch column not in the covering set (line 15). The best among the solution found in the previous case and the one computed here is the current solution.

Notice the following facts about the procedure *sm_mincov*:

- The parameter *lbound* is updated once (line 4). The reason is that after the computation of the essential columns (line 1) and of the independent set (line 3), the cost of the previous partial solution summed to the cost of the essential columns and of the independent set is potentially a sharper lower bound on any complete solution obtained from this node of the recursion

tree. The updated value *lbound_new* is used in the rest of the routine. The lower bound is a monotonically increasing quantity along each path of the computation tree.

- The parameter *ubound* is updated once (line 13). At that point a new complete solution has just been returned by the recursive call to *sm_mincov* (line 12) and an updated value of *ubound* must be recomputed for the following recursive call of *sm_mincov* (line 15). The reason is that when a new complete solution is obtained, the current *ubound* is not any more valid and therefore it must be updated before it is used again. To be updated, *ubound* is compared against the cost of the newly found solution, and the minimum of the two is the new *ubound*. The upper bound is a monotonically decreasing quantity throughout the entire computation.

The previous analysis proves that the algorithm finds a minimum cost satisfying assignment to the problem.

5.4.2 N-way Partitioning

If the covering matrix M can be partitioned into two disjoint blocks M_1 and M_2, the covering problem can be reduced to two independent covering subproblems, and the minimum covering for M is the union of the minimum coverings for M_1 and M_2. Such bi-partition can be found by putting in M_1 a row and all columns that have an element in common with the row (i.e., the columns intersecting the row) and recursively all rows and columns intersecting any row or column in M_1. The remaining rows and columns (i.e., not intersecting any row or column in M_1) are put in M_2. This algorithm can be generalized to find partitions made by N blocks, as shown in Figure 5.4.

Theorem 5.1 *If a covering matrix M can be partitioned into n disjoint blocks $M_1, M_2, \ldots, M_n$, the union of the minimum covers of $M_1, M_2, \ldots, M_n$ is the minimum cover of M.*

Bi-partitioning is implemented in [64, 76] as follows. When checking for a partition of the problem (line 7), the routine *sm_mincov* is called recursively on two independents subproblems (lines 8 and 10), if they exist. When solving the smaller of the two subproblems (line 8), the initial solution is empty, the initial lower bound is set to 0, the initial upper bound is set to the difference between the current *ubound* and the cost of the current partial solution. When solving the larger of the two subproblems (line 10), the initial solution is the current solution (to which the solution of the smaller subproblem is added, if it is not empty), the initial lower bound is set to the current lower bound *lbound_new*, the initial upper bound is set to the current *ubound*.

```
n_way_partition(M) {
    while (there is a row r_i not in any partition) {
        put r_i in a new partition M_k
        while (there is a row r_j connected to any row in partition M_k ) {
            put row r_j in partition M_k
        }
    }
}
```

Figure 5.4: N-way partitioning.

Theorem 5.2 *The upper bound set in the smaller subproblem is correct.*

Proof: Let *select* be the partial solution along the current path. It holds that (cost of the final solution along the current path) $\geq$ (cost of solving M_1 + $cost(select)$ + 1). If (cost of solving M_1) $\geq$ $(ubound - cost(select))$, then (cost of the final solution along the current path) $\geq$ $(ubound + 1)$, i.e., (cost of the final solution along the current path) $>$ $ubound$. This is ruled out by setting the upper bound when solving M_1 to $(ubound - cost(select))$, since *sm_mincov* returns a non-empty solution only if it can beat the given upper bound. ■

5.4.3 Maximal Independent Set

The cardinality of a maximum set of pairwise disjoint rows of M (i.e., no 1's in the same column) is a lower bound on the cardinality of the solution to the covering problem, because a different element must be selected for each of the independent rows in order to cover them. If the size of current solution plus the size of the independent set is greater or equal to the best solution seen so far, the search along this branch can be terminated because no solution better than the current one can possibly be found. It is also true that the size of the independent set at the first level of the recursion is a lower bound for the final minimum cover, so that the search can be terminated if a solution is found of size equal to this lower bound. Since finding a maximum independent set is an NP-complete problem, in practice an heuristic is used that provides a weaker lower bound. Notice that even the lower bound provided by solving exactly maximum independent set is not sharp.

In [67, 64, 76], the adjacency matrix B of a graph whose nodes correspond to rows in the cover matrix M is created. In the binate case, only rows are taken into consideration which do not contain any 0 element. An edge is placed between two nodes if the two rows have an element in common. While B is non-empty, a row R_i of B is found that is disjoint from a maximum number of rows (i.e., the row of minimum length in B). The column of minimum weight intersecting R_i is also found. The weight is cumulated in the independent set cost. All rows having elements in common with R_i are then deleted from B. At the end of the *while*-iteration a set of pairwise disjoint rows (independent set) and their minimum covering cost is found. Notice that one could think to the problem in a dual way as finding a maximal clique in a graph with the same rows as before, and edges between two nodes representing two disjoint rows.

5.4.4 Selection of a Branching Column

The selection of a good branching column is essential for the efficiency of the branch and bound algorithm. Since the time taken by the selection is a significant part of the total, a trade-off must be made between quality and efficiency.

In [67, 64, 76], the selection of the branching variable is restricted to columns intersecting the rows of the independent set, because a unique column must eventually be selected from each row of the maximal independent set. Among those rows, the selection strategy favors columns with large number of 1's and intersecting many short rows. Short rows are considered difficult rows and choosing them first favors the creation of essential columns. More precisely, the column of highest merit is chosen. The merit of a given column is computed as the product of the inverse of the weight of the column multiplied by the sum of the contributions of all rows intersected in a 1 by the column. The inverse of the contribution of a row is equal to the number of all non-2 elements (each can contribute in covering the row) minus 1. The inverse is well-defined, because at this stage each row has at least two-elements (it is not essential).

5.5 Reduction Techniques

Three fundamental processes constitute the essence of the reduction rules:

1. Selection of a column: a column must be selected if it is the only column that satisfies a required constraint (Section 5.5.7). A dual statement holds for unacceptable columns (Section 5.5.8). Also related is the case of unnecessary columns (Section 5.5.9).

2. Elimination of a column: a column C_i can be eliminated, if its elimination does not preclude obtaining a minimal cover, i.e., if there exists in M another column C_j that satisfies at least all the constraints satisfied by C_i (Section 5.5.5).

3. Elimination of a row: a row R_i can be eliminated if there exists in M another row R_j that expresses the same or a stronger constraint (Section 5.5.1).

Even though more complex criteria of dominance have been investigated (for instance, Section 5.5.12), the previous ones are basic in any table covering solver. Reduction rules have previously been stated for the binate covering case [28, 29, 10, 9], and also for the unate covering case [54, 68, 9]. The former set of reductions are needed for state minimization. Here we will present the known reduction rules directly for binate covering and indicate how they simplify for unate covering, when applicable. For each of them, we will first define the reduction rule, and then a theorem showing how that rule is applied. Proofs for the correctness of these reduction rules have been given in [28, 29, 10, 9], and they will not be repeated here, except for a few less common ones. We will provide a survey comparing different related reduction rules used in the literature.

The effect of reductions depends on the order of their application. Reductions are usually attempted in a given order, until nothing changes any more (i.e., the covering matrix has been reduced to a cyclic core). Figure 5.5 shows how reductions are applied in [67, 64, 76][1].

5.5.1 Row Dominance

Definition 5.1 *A row R_i dominates another row R_j if R_j has all the 1's and 0's of R_i; i.e., for each column C_k of M, one of the following occurs:*

- $M_{i,k} = 1$ *and* $M_{j,k} = 1$,
- $M_{i,k} = 0$ *and* $M_{j,k} = 0$,
- $M_{i,k} = 2$.

Theorem 5.3 *If a row R_j is dominated by another row R_i, R_j can be eliminated without affecting the solutions to the covering problem.*

This definition of row dominance is

[1]The reductions *β-dominance* and *row_consensus* are only in [64] and the reduction by implication is only in [76].

```
sm_reduce(M, select, weight, ubound) {
    do {
        apply β-dominance or α-dominance
        find essential columns
        find unacceptable columns
        if (a column is both essential and unacceptable)
            return empty_solution
        for each essential column {
            delete each row intersecting the column in a 1
            if (a row of length 1 intersects the column in a 0)
                return empty_solution
            delete column
            add column to select
            if (cost of select ≥ ubound)
                return empty_solution
        }
        for each unacceptable column {
            delete each row intersecting the column in a 0
            if (a row of length 1 intersects the column in a 1)
                return empty_solution
            delete column
        }
        apply row_consensus
        apply row_dominance
    } while (reductions are applicable)
    return select
}
```

Figure 5.5: Flow of reduction rules.

- similar to column dominance (Rule 3) in [28], except that the labels of dominator row, R_i, and dominated row, R_j, are reversed (i.e., reverse definition of dominance),
- similar to column dominance (Rule 3) in [29], except that the labels of dominator row, R_i, and dominated row, R_j, are reversed (i.e., reverse definition of dominance),
- equivalent to row dominance (Definition 10) in [10],
- identical to row dominance (Definition 2.11) in [9].

Row Dominance for a Unate Table

Definition 5.2 *A row R_i dominates another row R_j if for all columns C_k, $M_{i,k} = 1 \Rightarrow M_{j,k} = 1$.*

5.5.2 Row Consensus

Theorem 5.4 *If R_i dominates R_j, except for a (unique) column C_k where R_i and R_j have different values, element $M_{j,k}$ can be eliminated from the matrix M (i.e., the entry in position $M_{j,k}$ becomes a 2) without affecting the solutions of the covering problem.*

Proof: Suppose that entry $M_{j,k}$ is 1 and entry $M_{i,k}$ is 0. The argument is the same if entry $M_{j,k}$ is 0 and entry $M_{i,k}$ is 1. If entry $M_{j,k}$ is removed, the problem arises that we are not able to satisfy row R_j by setting x_k to 1. A problem arises if a minimum-cost solution requires x_k set to 1, because we could miss the fact that setting x_k to 1 satisfies also row R_j. Instead we could obtain an higher-cost solution, by selecting another column in order to satisfy row $R_j - M_{j,k}$. We now show that this is not the case. If a minimum-cost solution requires x_k set to 1, we must still satisfy row R_i that cannot be satisfied by x_k set to 1. Whatever choice will be made to satisfy R_i, it will satisfy also $R_j - M_{j,k}$ (since $R_j - M_{j,k}$ has all 1's and 0's of R_i) and therefore no more cost will be incurred to satisfy row $R_j - M_{j,k}$. The previous argument fails if $R_j - M_{j,k}$ is empty and there are cases in which an higher-cost solution would be found. One could claim that if $R_j - M_{j,k}$ is empty, then R_j has only entry $M_{j,k}$ and therefore x_k is an essential, that is taken care by the essential column detection. In reality it may happen that by applying row consensus many times to the same row R_j (using different rows R_i) at a certain point R_j is emptied. In that case the last application of row consensus is potentially faulty and should not be done. ■

Row consensus is applied in [64]. This criterion generalizes the one given in [33].

5.5.3 Column α-Dominance

Definition 5.3 *A column C_j α-dominates another column C_k if*

- $c_j \leq c_k$,
- *C_j has all the 1's of C_k,*
- *C_k has all the 0's of C_j;*

i.e., $c_j \leq c_k$, and for each row R_i of M, none of the following can occur:

- $M_{i,j} = 2$ *and* $M_{i,k} = 1$,
- $M_{i,j} = 0$ *and* $M_{i,k} = 1$,
- $M_{i,j} = 0$ *and* $M_{i,k} = 2$.

Alternatively, $c_j \leq c_k$, and for each row R_i of M, one of the following occurs:

- $M_{i,j} = 1$,
- $M_{i,j} = 2$ *and* $M_{i,k} \neq 1$,
- $M_{i,j} = 0$ *and* $M_{i,k} = 0$.

Note that these last 3 cases are exactly the complement of the cases excluded above.

Theorem 5.5 *Let M be satisfiable. If a column C_k is α-dominated by another column C_j, there is at least one minimum cost solution with column C_k eliminated ($x_k = 0$), together with all the rows in which it has 0's.*

This definition of column α-dominance is

- an extension to row α-dominance (Rule 1) in [28], because the latter doesn't include the case $M_{i,j} = 0$ and $M_{i,k} = 0$,
- equivalent to first half of Rule 4 in [29]: (a) C_j has all the 1's of C_k and (b1) C_k has all the 0's of C_j,
- identical to column dominance (Definition 11, Theorem 3) in [10],
- identical to column dominance (Definition 2.12, Theorem 2.4.1) in [9].

Column Dominance for a Unate Table

Definition 5.4 *A column C_i dominates another column C_j if for all rows R_k, $M_{k,j} = 1 \Rightarrow M_{k,i} = 1$.*

5.5.4 Column β-Dominance

Definition 5.5 *A column C_i β-dominates another column C_j if*

- $c_i \leq c_j$,
- *C_i has all the 1's of C_j,*
- *for every row R_p in which C_i has a 0, either C_j has a 0 or there exists a row R_q in which C_j has a 0 and C_i does not have a 0, such that disregarding entries in columns C_i and C_j, R_q dominates R_p.*

Theorem 5.6 *Let M be satisfiable. If C_i β-dominates C_j, there is at least one minimum cost solution with column C_j eliminated ($x_j = 0$), together with all the rows in which it has 0's.*

Proof: We must show that given a solution, one can find another solution, of cost lesser or equal, with column C_j eliminated ($x_j = 0$). There are two cases for the original solution: either $x_i = 1$ and $x_j = 1$ or $x_i = 0$ and $x_j = 1$ (if $x_j = 0$, we are done). The new solution has $x_i = 1$ and $x_j = 0$ and coincides for the rest with the given solution. The case when $x_i = 1$ and $x_j = 1$ is easy, because column C_i has all 1's of column C_i and therefore C_j is useless.

Consider now the case when $x_i = 0$ and $x_j = 1$. The clauses with a 0 in column C_i are satisfied by not choosing C_i and the clauses with a 1 in column C_j are satisfied by choosing C_j. Each clause with a 0 in column C_j (and without a 0 in column C_i) is satisfied by a proper assignment of a column different from C_i and C_j, say C_k. Notice that the hypothesis that column C_i does not have a 0 in the clause is essential here, otherwise this clause would be satisfied already by not choosing C_i, without resorting to a column C_k. Now consider the assignment with column C_i and without column C_j ($x_i = 1$ and $x_j = 0$) and the same remaining assignments as the previous one. It costs no more than the previous one. We show that it is a solution. In order to do that we must make sure that the 0's covered by C_i and the 1's covered by C_j by setting $x_i = 0$ and $x_j = 1$, are still covered in the new assignment where $x_i = 1$ and $x_j = 0$. The clauses with a 1 in C_j are satisfied by C_i, because C_i has all 1's of C_j. Each clause, say R_p, with a 0 in column C_i is satisfied too, because there is a corresponding clause, say R_q, with a 0 in column C_j, and we already noticed

that there exists another column, C_k, that satisfies R_q. But by hypothesis R_q dominates R_p, i.e., R_p has all the 1's and 0's of R_q, hence column C_k satisfies also clause R_p (if entry $M_{q,k} = 1(0)$, then entry $M_{p,k} = 1(0)$ also and $x_k = 1$ ($x_k = 0$) satisfies both clauses). ■

This definition of column β-dominance is

- strictly stronger than column α-dominance given in 5.5.3,
- more general than row β-dominance (Rule 5) in [28], because the latter assumes that the covering table contains only rows with no or one 0,
- equivalent to second half of Rule 4 in [29]: (a) C_i has all the 1's of C_j and (b2) for every row R_p in which C_i has a 0, there exists a row R_q in which C_j has a 0, such that disregarding entries in row C_i and C_j, R_p dominates R_q (with reverse definition of row dominance), noticing that by mistake the condition that C_i does not have a 0 in row R_q was omitted,
- not mentioned in [10] and [9].

5.5.5 Column Dominance

Definition 5.6 *A column C_i dominates another column C_j if either C_i α-dominates C_j or C_i β-dominates C_j.*

Theorem 5.7 *Let M be satisfiable. If C_i dominates C_j, there is at least one minimum cost solution with column C_j eliminated ($x_j = 0$), together with all the rows in which it has 0's.*

5.5.6 Column Mutual Dominance

Definition 5.7 *Two columns C_i and C_j mutually dominate each other if*

- *C_i has a 0 in every row where C_j has a 1,*
- *C_j has a 0 in every row where C_i has a 1.*

Theorem 5.8 *Let M be satisfiable. If C_i and C_j mutually dominate each other, there is at least one minimum cost solution with columns C_i and C_j eliminated ($x_i = x_j = 0$), together with all the rows in which they have 0's.*

This definition of column mutual dominance is

- identical to rule for mutually reducible variables in [74],
- not mentioned in other papers.

5.5.7 Essential Column

Definition 5.8 *A column C_j is an essential column if there exists a row R_i having a* 1 *in column C_j and* 2*'s everywhere else.*

Theorem 5.9 *If C_j is an essential column, it must be selected ($x_j = 1$) in every solutions. Column C_j must then be deleted together with all the rows in which it has* 1*'s.*

This definition of essential column is

- identical to essential row (Rule 2) in [28],
- identical to Rule 1 in [29],
- included in Definition 9 in [10]: the row R_i in the above definition corresponds to a singleton-1 essential row in [10],
- included in Definition 2.10 in [9]: the row R_i in the above definition corresponds to a singleton-1 essential row in [9].

Essential Column for a Unate Table

Definition 5.9 *A column is an essential column if it contains the* 1 *of a singleton row.*

5.5.8 Unacceptable Column

Definition 5.10 *A column C_j is an unacceptable column if there exists a row R_i having a* 0 *in column C_j and* 2*'s everywhere else.*

This reduction rule is a dual of the essential column rule.

Theorem 5.10 *If C_j is an unacceptable column, it must be eliminated ($x_j = 0$) in every solution, together with all the rows in which it has* 0*'s.*

This definition of unacceptable column is

- identical to that of nonselectionable row in [28],
- identical to Rule 2 in [29],

- included in Definition 9 in [10]: the row R_i in the above definition corresponds to a singleton-0 essential row in [10],
- included in Definition 2.10 in [9]: the row R_i in the above definition corresponds to a singleton-0 essential row in [9].

5.5.9 Unnecessary Column

Definition 5.11 *A column of only* 0*'s and* 2*'s is an unnecessary column.*

Notice that there is no symmetric rule for columns of 1's and 2's. The reason is that selecting a column to be in the solution has a cost, while eliminating it has no cost.

Theorem 5.11 *If C_j is an unnecessary column, it may be eliminated ($x_j = 0$), together with all the rows in which it has* 0*'s.*

This definition of unnecessary column is

- identical to Rule 4 in [28],
- identical to Rule 5 in [29],
- not mentioned in [10] and [9].

5.5.10 Trial Rule

Theorem 5.12 *If there exists in a covering table M a row R_i having a* 0 *in column C_j, a* 1 *in column C_k and* 2*'s in the rest, then apply the following test:*

- *eliminate C_k together with the rows in which it has* 0*'s,*
- *eliminate C_j, which is now an unacceptable column, together with the rows in which it has* 0*'s,*
- *continue as long as possible to eliminate the columns which becomes unacceptable columns.*

If at least one row of M has only 2*'s at the end of this test, then column C_k must be selected ($x_k = 1$)*[2]*. Therefore, C_k can be deleted together with all the columns in which it has* 1*'s.*

[2]It is possible that a row is left with only 2's by a sequence of reduction steps.

This reduction rule is

- identical to Rule 6 in [28],
- not mentioned in other papers.

5.5.11 Infeasible Subproblem

Unlike the unate covering problem, the binate covering problem may be infeasible. In particular, an intermediate covering matrix M may found to be unsatisfiable by the following theorem. When an infeasible subproblem is found, that branch of the binary recursion is pruned.

Theorem 5.13 *A covering problem M is infeasible if there exists a column C_j which is both essential and unacceptable (implying $x_j = 1$ and $x_j = 0$).*

This definition of infeasibility is

- not mentioned in [28] and [29],
- briefly mentioned in [10],
- identical to the unfeasible problem in [9].

5.5.12 Gimpel's Reduction Step

Another heuristic for solving the minimum cover problem has been suggested by Gimpel [27]. Gimpel proposed a reduction step which simplifies the covering matrix when it has a special form. This simplification is possible without further branching, and hence is useful at each step of the branch and bound algorithm. In practice, Gimpel's reduction step is applied after reducing the covering matrix to the cyclic core.

Gimpel's reduction can be described in terms of the product-of-sums represented by a covering table. The product-of-sums is examined to see if any clause has only two literals of the same cost. For example, assume the expression has the form:

$$p = R(c_1 + c_2)(c_1 + S_1)\ldots(c_1 + S_n)(c_2 + T_1)\ldots(c_2 + T_m)$$

where c_1 and c_2 are single variables with a cost C, $S_i, i = 1\ldots n$ and $T_j, j = 1\ldots m$ are sums of variables not containing c_1 or c_2, and R is a product of sums of variables not containing c_1 or c_2.

Because the covering table is assumed minimal, if there is a clause $(c_1 + c_2)$, then $m \geq 1$, $n \geq 1$, and none of S_i or T_j is identically zero.

Note that with the expression written in this form, each parenthesized expression corresponds directly to a single row in the covering table. By algebraic manipulations, the expression can be re-written as:

$$p = R(c_1c_2 + c_1T + c_2S)$$

where $S = \prod_{i=1}^{n} S_i$, and $T = \prod_{i=1}^{m} T_i$.

A second covering problem is derived from the original covering problem with the following form:

$$\begin{aligned} p_1 &= R(c_2 + S + T) \\ &= R\prod_{i=1}^{n}\prod_{j=1}^{m}(c_2 + S_i + T_j) \end{aligned}$$

The main theorem of Gimpel is:

Theorem 5.14 *Let M_1 be a minimum cover for p_1. A cover for p can be derived from M_1 according to the rule:* if S is covered by M_1 then add c_2 to M_1 to derive a cover of p; otherwise, add c_1 to M_1 to derive a cover of p. *The resulting cover is a minimum cover for p.*

A proof can be found in [68], where a more extended discussion is presented.

Gimpel's reduction step was originally stated for covering problems where each column had cost 1. Robinson and House [34] showed that the reduction remains valid even for weighted covering problems if the cost of the column c_1 equals the cost of the column c_2, as it has been presented here. Gimpel's rule has been first proposed in [27] and then implemented in [67]. In [64, 76] Gimpel's rule has been extended to handle the binate case. This extension has been described in [77].

5.6 Implicit Binate Covering

The classical branch-and-bound algorithm [28, 29] for minimum-cost binate covering has been described in previous sections, and implemented by means of efficient computer programs (ESPRESSO and STAMINA). These state-of-the-art binate table solvers represent binate tables efficiently using sparse matrix packages. But the fact that each non-empty table entry still has to be explicitly represented put a bound on the size of the tables that can be handled by these binate

```
mincov(R, C, U) {
    (R, C) = Reduce(R, C, U)
    if (Terminal_Case(R, C))
        if (cost(R, C) ≥ U) return empty_solution
        else U = cost(R, C); return solution
    L = Lower_Bound(R, C)
    if (L ≥ U) return _solution
    c_i = Choose_Column(R, C)
    S^1 = mincov(R_{c_i}, C_{c_i}, U)
    S^0 = mincov(R_{c̄_i}, C_{c̄_i}, U)
    return Best_Solution(S^1 ∪ {c_i}, S^0)
}
```

Figure 5.6: Implicit branch-and-bound algorithm.

solvers. For example from Section 4.7, we would not expect these binate solvers to handle many state minimization examples requiring over 10^6 compatibles (up to 2^{1500} prime compatibles), the selection of which would require tables with that many columns. To keep with our stated objective, the binate table has to be represented implicitly. We do not represent (even implicitly) the elements of the table, but we make use only of a set of row labels and a set of column labels, each represented implicitly as a BDD. They are chosen so that the existence and value of any table entry can be readily inferred by examining its corresponding row and column labels. In the sequel, we shall assume that every row has a unit cost.

A binate covering problem instance can be characterized by a 6-tuple $(r, c, R, C, 0, 1)$, defined as follows:

- the group of variables for labeling the rows: r
- the group of variables for labeling the columns: c
- the set of row labels: $R(r)$
- the set of column labels: $C(r)$

- the 0-entries relation at the intersection of row r and column c: $\mathtt{0}(r, c)$
- the 1-entries relation at the intersection of row r and column c: $\mathtt{1}(r, c)$

In other words, the user of our implicit binate solver would first choose an encoding for the rows and columns. Given a binate table, the user will then supply a set of row labels as a BDD $R(r)$ and a set of column labels as a BDD $C(c)$, and also the two inference rules in the form of BDD relations, $\mathtt{0}(r, c)$ and $\mathtt{1}(r, c)$, capturing the 0-entries and 1-entries.

The classical branch-and-bound solution of minimum cost binate covering is based on the recursive procedure as shown in Figure 5.3. In our implicit formulation, we keep the branch-and-bound scheme summarized in Figure 5.6, but we replace the traditional description of the table as a (sparse) matrix with an implicit representation, using BDD's for the characteristic functions of the rows and columns of the table. Moreover, we have implicit versions of the manipulations on the binate table required to implement the branch-and-bound scheme. In the following sections we are going to describe the following:

- implicit representation of the covering table,
- implicit reduction,
- implicit branching column selection,
- implicit computation of the lower bound, and
- implicit table partitioning.

At each call of the binate cover routine *mincov*, the binate table undergoes a reduction step *Reduce* and, if termination conditions are not met, a branching column is selected and *mincov* is called recursively twice, once assuming the selected column c_i in the solution set (on the table R_{c_i}, C_{c_i}) and once out of the solution set (on the table $R_{\overline{c_i}}, C_{\overline{c_i}}$). Some suboptimal solutions are bounded away by computing a lower bound L on the current partial solution and comparing it against an upper bound U (best solution obtained so far). A good lower bound is based on the computation of a maximal independent set.

5.7 Implicit Table Generation for State Minimization

As the first application of our implicit binate solver, the state minimization problem of ISFSM's described in Section 4 is solved implicitly. In this section we shall review how its binate

covering table is generated.

Definition 5.12 *A column is labeled by a positional-set p. The set of* **column labels** *C is obtained by prime generation as $C(p) = \mathcal{PC}(p)$.*

Besides distinguishing a row from another, each row label must also contain information regarding the positions of 0's and 1's in the row. Each row label r consists of a pair of positional-sets (c, d). Since there is at most one 0 in the row, the label of the column p intersecting it in a 0 is recorded in the row label by setting its c part to p. If there is no 0 in the row, c is set to the empty set, $Tuple_0(c)$. Therefore a row label r corresponds to a unate clause if and only if relation $unate\_row(r) = Tuple_0(c)$ is true. Because of Definition 5.14 for row labels, the columns intersecting a row labeled $r = (c, d)$ in a 1 are labeled by the prime compatibles p that contain d. i.e.,

Definition 5.13 *The table entry at the intersection of a row labeled by $r = (c, d) \in R$ and a column labeled by $p \in C$ can be inferred by:*

the table entry is a 0 if and only if relation $0(r, p) \stackrel{\text{def}}{=} (p = c)$ is true,

the table entry is a 1 if and only if relation $1(r, p) \stackrel{\text{def}}{=} (p \supseteq d)$ is true.

Definition 5.14 *The set of* **row labels** *R is given by:*

$$R(r) = \mathcal{PC}(c) \cdot \mathcal{CCS}(c, d) + Tuple_0(c) \cdot Tuple_1(d)$$

The closure conditions associated with a prime compatible p are that if p is included in a solution, each implied set d in its class set must be contained in at least one chosen prime compatible. A binate clause of the form $(\overline{p} + p_1 + p_2 + \cdots + p_k)$ has to be satisfied for each implied set of p, where p_i is a prime compatible containing the implied set d. The labels for binate rows are given succinctly by $\mathcal{PC}(c) \cdot \mathcal{CCS}(c, d)$. There is a row label for each (c, d) pair such that $c \in \mathcal{PC}$ is a prime compatible and d is one of its implied sets in $\mathcal{CCS}(c, d)$. This row label consistently represents the binate clause because the 0 entry in the row is given by the column labeled by the prime compatible $p = c$, and the row has 1's in the columns labeled by p_i wherever $(p_i \supseteq d)$.

The covering conditions require that each state be contained by some prime compatible in the solution. For each state $d \in S$, a unate clause has to be satisfied which is of the form $(p_1 + p_2 + \cdots + p_j)$ where the p_i's are the prime compatibles that contain the state d. By specifying the unate row labels to be $Tuple_0(c) \cdot Tuple_1(d)$, we define a row label for each state in $Tuple_1(d)$.

Since the row has no 0, its c part must be set to $Tuple_0(c)$. The 1 entries are correctly positioned at the intersection with all columns labeled by prime compatibles p_i which contain the singleton state d.

From the next section, we shall describe how a binate covering table can be manipulated implicitly so as to solve the minimum cost binate covering problem. We are going to provide solutions to three variations of the covering problem, and they are listed below in decreasing order of generality in the binate table specification:

1. General binate covering table
   - the group of variables for labeling the rows: r
   - the group of variables for labeling the columns: c
   - the set of row labels: $R(r)$
   - the set of column labels: $C(c)$
   - the 0-entries relation at the intersection of row r and column c: $0(r, c)$
   - the 1-entries relation at the intersection of row r and column c: $1(r, c)$
2. Binate covering table assuming each row has at most one 0:
   - the group of variables for labeling the rows: r
   - the group of variables for labeling the columns: c
   - the set of row labels: $R(r)$
   - the set of column labels: $C(c)$
   - the 0-entries relation at the intersection of row r and column c: $0(r, c)$
   - the 1-entries relation at the intersection of row r and column c: $1(r, c)$
3. Specialized binate covering table for exact state minimization:
   - the group of variables for labeling the rows (each label is a pair): (c, d)
   - the group of variables for labeling the columns: p
   - the set of row labels: $R(c, d)$
   - the set of column labels: $C(p)$
   - the 0-entries relation at the intersection of row (c, d) and column p: $0((c, d), p) = (p = c)$

- the 1-entries relation at the intersection of row (c, d) and column p:
  $1((c, d), p) = (p \supseteq d)$

In the sequel, each implicit table operation will be expressed by three BDD formulas, each representing a realization for a different implicit binate solver. Each equation will be labeled 1, 2, or 3, depending on which of the above set of assumptions are made.

5.8 Implicit Reduction Techniques

Reduction rules aim to the following:

1. Selection of a column. A column must be selected if it is the only column that satisfies a given row. A dual statement holds for columns that must not be part of the solution in order to satisfy a given row.

2. Elimination of a column. A column c_i can be eliminated if its elimination does not preclude obtaining a minimum cover, i.e., if there is another column c_j that satisfies at least all the rows satisfied by c_i.

3. Elimination of a row. A row r_i can be eliminated if there exists another row r_j that expresses the same or a stronger constraint.

The order of the reductions affects the final result. Reductions are usually attempted in a given order, until nothing changes any more (i.e., the covering matrix has been reduced to a cyclic core). The reductions and order implemented in our reduction algorithm are summarized in Figure 5.7.
In the reduction, there are two cases when no solution is generated:

1. The added cardinality of the set of essential columns, and of the partial solution computed so far, Sol, is larger or equal than the upper bound U. In this case, a better solution is known than the one that can be found from now on and so the current computation branch can be bounded away.

2. After having eliminated essential, unacceptable and unnecessary columns and covered rows, it may happen that the rest of the rows cannot be covered by the remaining columns. In this case, the current partial solution cannot be extended to any full solution.

```
Reduce(R, C, U) {
    repeat {
        Collapse_Columns(C)
        Column_Dominance(R, C)
        Sol = Sol ∪ Essential_Columns(R, C)
        if (|Sol| ≥ U) return empty_solution
        Unacceptable_Columns(R, C)
        Unnecessary_Columns(R, C)
        if (C does not cover R) return empty_solution
        Collapse_Rows(R)
        Row_Dominance(R, C)
    } until (both R and C unchanged)
    return (R, C)
}
```

Figure 5.7: Implicit reduction loop.

We are going to describe how the reduction operations are performed implicitly using BDD's on the three table representations described in the previous section.

5.8.1 Duplicated Columns

It is possible that more than one column (row) label is associated with columns (rows) that coincide element by element. We need to identify such duplicated columns (rows) and collapse them into a single column (row). This avoids the problem of columns (rows) dominating each other when performing implicitly column (row) dominance. The following computations can be seen as finding the equivalence relation of duplicated columns (rows) and selecting one representative for each equivalence class.

Definition 5.15 *Two columns are duplicates, if on every row, their corresponding table entries are identical.*

Theorem 5.15 Duplicated columns *can be computed as:*

$$dup\_col(c',c) \;^{1}= \; \forall r\,\{R(r) \Rightarrow [(0(r,c') \Leftrightarrow 0(r,c)) \cdot (1(r,c') \Leftrightarrow 1(r,c))]\}$$

$$dup\_col(c',c) \;^{2}= \; \forall r\,\{R(r) \Rightarrow [\neg 0(r,c') \cdot \neg 0(r,c) \cdot (1(r,c') \Leftrightarrow 1(r,c))]\}$$

$$dup\_col(p',p) \;^{3}= \; \nexists d\, R(p',d) \cdot \nexists d\, R(p,d) \cdot \forall d\,\{[\exists c\, R(c,d)] \Rightarrow [(p' \supseteq d) \Leftrightarrow (p \supseteq d)]\}$$

Proof: As discussed at the end of Section 5.7, the first equation computes the duplicated columns relation for the most general binate table, and the second equation for the binate table with the assumption that there is at most one 0 in each row, and the third equation is for the specialized binate table for state minimization, assuming the columns are prime compatibles p, and the rows are pairs (c,d).

For the column labels c' and c to be in the relation dup_col, the first equation requires the following conditions to be met for every row label $r \in R$: (1) the entry (r,c) is a 0 if and only if the entry (r,c') is a 0, (i.e., $0(r,c') \Leftrightarrow 0(r,c)$), and (2) the entry (r,c) is a 1 if and only if the entry (r,c') is a 1, (i.e., $1(r,c') \Leftrightarrow 1(r,c)$). Assuming each row has at most one 0 for the second equation, condition 2 requires that the row labeled r cannot intersect either column at a 0, (i.e., $\neg 0(r,c') \cdot \neg 0(r,c)$). ■

Theorem 5.16 *Duplicated columns can be collapsed by:*

$$C(c) \;^{1,2}= \; C(c) \cdot \nexists c'\,[C(c') \cdot (c' \prec c) \cdot dup\_col(c',c)]$$

$$C(p) \;^{3}= \; C(p) \cdot \nexists p'\,[C(p') \cdot (p' \prec p) \cdot dup\_col(p',p)]$$

Proof: This computation picks a representative column label out of a set of column labels corresponding to duplicated columns. A column label c is deleted from C if and only if there is another column label c' which has a smaller binary value than c (denoted by $c' \prec c$) and both label the same duplicated column. Here we exploit the fact that any positional-set c can be interpreted as a binary number. Therefore, a unique representative from a set can be selected by picking the one with the smallest binary value. [3] ■

5.8.2 Duplicated Rows

Definition 5.16 *Two rows are duplicates if, on every column, their corresponding table entries are identical.*

Detection of duplicated rows, selection of a representative row, and table updating are performed by the following equations as in the case of duplicated columns.

Theorem 5.17 Duplicated rows *can be computed as:*

$$dup\_row(r', r) \quad {}^{1,2}= \quad \forall c \, \{C(c) \Rightarrow [(0(r', c) \Leftrightarrow 0(r, c)) \cdot (1(r', c) \Leftrightarrow 1(r, c))]\}$$
$$dup\_row(c', d', c, d) \quad {}^{3}= \quad (c' = c) \cdot \nexists p \, [C(p) \cdot ((p \supseteq d') \not\Leftrightarrow (p \supseteq d))]$$

Proof: Similar to the proof for Theorem 5.15. For the row labels r' and r to be in the relation *dup_row*, the first equation requires the following conditions to be met for every column label $c \in C$: (1) the entry (r, c) is a 0 if and only if the entry (r', c) is a 0, (i.e., $0(r', c) \Leftrightarrow 0(r, c)$), and (2) the entry (r, c) is a 1 if and only if the entry (r', c) is a 1, (i.e., $1(r', c) \Leftrightarrow 1(r, c)$). ■

Theorem 5.18 *Duplicated rows can be collapsed by:*

$$R(r) \quad {}^{1,2}= \quad R(r) \cdot \nexists r' \, [R(r') \cdot (r' \prec r) \cdot dup\_row(r', r)]$$
$$R(c, d) \quad {}^{3}= \quad R(c, d) \cdot \nexists c', d' \, [R(c', d') \cdot (d' \prec d) \cdot dup\_row(c', d', c, d)]$$

Proof: The proof is similar to that for Theorem 5.16, except we are delete all duplicating rows here except the representative ones. ■

From now on, sometimes we will blur the distinction between a column (row) label and the column (row) itself, but the context should say clearly which one it is meant.

[3] Alternatively, one could have used the *cproject* BDD operator introduced in [49] to pick a representative column out of each set of duplicated columns.

5.8.3 Column Dominance

Some columns need not be considered in a binate table, if they are dominated by others. Classically, there are two notions of column dominance: α-dominance and β-dominance.

Definition 5.17 *A column c'* **α-dominates** *another column c if c' has all the 1's of c, and c has all the 0's of c'.*

Theorem 5.19 *The α-dominance relation can be computed as:*

$$\alpha\_dom(c', c) \stackrel{1}{=} \not\exists r \{R(r) \cdot [1(r, c) \cdot \neg 1(r, c')] + [0(r, c') \cdot \neg 0(r, c)]\}$$
$$\alpha\_dom(c', c) \stackrel{2}{=} \not\exists r \{R(r) \cdot [1(r, c) \cdot \neg 1(r, c') + 0(r, c')]\}$$
$$\alpha\_dom(p', p) \stackrel{3}{=} \not\exists c, d\, [R(c, d) \cdot (p \supseteq d) \cdot (p' \not\supseteq d)] \cdot \not\exists d\, R(p', d)$$

Proof: For column c' to α-dominate c, the first equation ensures that there doesn't exists a row $r \in R$ such that either (1) the table entry (r, c) is a 1 but the table entry (r, c') is not, *or* (2) the table entry (r, c') is a 0 but the table entry (r, c) is not. Assuming each row has at most one 0, condition 2 can be simplified to the second equation that table entry (r, c') is a 0. ■

Definition 5.18 *A column c'* **β-dominates** *another column c if (1) c' has all the 1's of c, and (2) for every row r' in which c' contains a 0, there exists another row r in which c has a 0 such that disregarding entries in column c', r' has all the 1's of r.*

Theorem 5.20 *The β-dominance relation can be computed by:*

$$\beta\_dom(c', c) \stackrel{1,2}{=} \not\exists r' \{R(r') \cdot [1(r', c) \cdot \neg 1(r', c') + 0(r', c') \cdot \not\exists r\, [R(r) \cdot 0(r, c) \cdot \not\exists c''\, [C(c'') \cdot (c'' \neq c') \cdot 1(r, c'') \cdot \neg 1(r', c'')]\,]]\}$$
$$\beta\_dom(p', p) \stackrel{3}{=} \not\exists d' \{\exists c' (R(c', d')) \cdot (p \supseteq d') \cdot (p' \not\supseteq d')\} \cdot \not\exists d' \{R(p', d') \cdot \not\exists d\, [R(p, d) \cdot \not\exists q\, [C(q) \cdot (q \neq p') \cdot (q \supseteq d) \cdot (q \not\supseteq d')]\,]\}$$

Proof: According to the definition, the table should *not* contain a row $r' \in R$ if either of the following two cases is true at that row: (1) table entry at column c is a 1 while entry at column c' is not a 1 (i.e., $1(r', c) \cdot \neg 1(r', c')$), *or* (2) c' has a 0 in row r' (i.e., $0(r', c')$) but there does not exist a row $r \in R$ such that its column c is a 0 and disregarding entries in column c', row r' has all the 1's of row r. Rephrasing the last part of the condition 2, the expression $\not\exists c''\, [C(c'') \cdot (c'' \neq c') \cdot 1(r, c'') \cdot \neg 1(r', c'')]$ requires that there is no column $c'' \in C$ apart from column c' such that c'' has a 1 in row r, but not in row r'. ■

The conditions for α-dominance are a strict subset of those for β-dominance, but α-dominance is easier to compute implicitly. Either of them can be used as the column dominance relation *col_dom*.

Theorem 5.21 *The set of dominated columns in a table (R, C) can be computed as:*

$$D(c) \stackrel{1,2}{=} C(c) \cdot \exists c' \left[C(c') \cdot (c' \neq c) \cdot col\_dom(c', c) \right]$$
$$D(p) \stackrel{3}{=} C(p) \cdot \exists p' \left[C(p') \cdot (p' \neq p) \cdot col\_dom(p', p) \right]$$

Proof: A column $c \in C$ is dominated if there is another $c' \in C$ different from c (i.e., $c' \neq c$) which column dominates c (i.e., $col\_dom(c', c)$). ■

Theorem 5.22 *The following computations delete a set of columns $D(c)$ from a table (R, C) and all rows intersecting these columns in a 0.*

$$C(c) \stackrel{1,2}{=} C(c) \cdot \neg D(c)$$
$$R(r) \stackrel{1,2}{=} R(r) \cdot \nexists c \left[D(c) \cdot 0(r, c) \right]$$

$$C(p) \stackrel{3}{=} C(p) \cdot \neg D(p)$$
$$R(c, d) \stackrel{3}{=} R(c, d) \cdot \neg D(c)$$

Proof: The first computation removes columns in $D(c)$ from the set of columns $C(c)$. The expression $\exists c \left[D(c) \cdot 0(r, c) \right]$ defines all rows r intersecting the columns in D in a 0. They are deleted from the set of rows R. ■

5.8.4 Row Dominance

Definition 5.19 *A row r'* **dominates** *another row r if r has all the 1's and 0's of r'.*

Theorem 5.23 *The row dominance relation can be computed by:*

$$row\_dom(r', r) \stackrel{1,2}{=} \nexists c \left\{ C(c) \cdot \left[1(r', c) \cdot \neg 1(r, c) + 0(r', c) \cdot \neg 0(r, c) \right] \right\}$$
$$row\_dom(c', d', c, d) \stackrel{3}{=} \nexists p \left[C(p) \cdot (p \supseteq d') \cdot (p \not\supseteq d) \right] \cdot \left[unate\_row(c') + (c' = c) \right]$$

Proof: For r' to dominate r, the equation requires that there is no column $c \in C$ such that either (1) the table entry (r', c) is a 1 but the entry (r, c) is not, *or* (2) the entry (r', c) is a 0 but the entry (r, c) is not. ■

Theorem 5.24 *Given a table $(R(r), C(c))$, the set of unate row labels r can be computed as*

$$unate\_row(r)^{1,2} = \not\exists c\,[C(c) \cdot 0(r,c)].$$

Given a table $(R(c,d), C(p))$, the set of unate row labels c can be computed as

$$unate\_row(c)^{3} = \not\exists p\,[C(p) \cdot (p = c)] = \not\exists c\, C(c).$$

Theorem 5.25 *The set of rows not dominated by other rows can be computed as:*

$$R(r)^{1,2} = R(r) \cdot \not\exists r'\,[R(r') \cdot (r' \neq r) \cdot row\_dom(r', r)]$$
$$R(c,d)^{3} = R(c,d) \cdot \not\exists c', d'\,\{R(c',d') \cdot [(c',d') \neq (c,d)] \cdot row\_dom(c', d', c, d)]\}$$

Proof: The equation expresses that any row $r \in R$, dominated by another *different* row $r' \in R$, is deleted from the set of rows $R(r)$ in the table. ■

5.8.5 Essential Columns

Definition 5.20 *A column c is an* **essential column** *if there is a row having a 1 in column c and 2 everywhere else.*

Theorem 5.26 *The set of essential columns can be computed by:*

$$ess\_col(c)^{1} = C(c) \cdot \exists r\,\{R(r) \cdot 1(r,c) \cdot \not\exists c'[C(c') \cdot (c' \neq c) \cdot (0(r,c') + 1(r,c'))]\}$$
$$ess\_col(c)^{2} = C(c) \cdot \exists r\,\{R(r) \cdot 1(r,c) \cdot unate\_row(r) \cdot \not\exists c'\,[C(c') \cdot (c' \neq c) \cdot 1(r,c')]\}$$
$$ess\_col(p)^{3} = C(p) \cdot \exists c, d\,\{R(c,d) \cdot (p \supseteq d) \cdot unate\_row(c) \cdot \not\exists p'\,[C(p') \cdot (p' \neq p) \cdot (p' \supseteq d)]\}$$

Proof: For a column $c \in C$ to be essential, there must exist a row $r \in R$ which (1) contains a 1 in column c (i.e., $1(r,c)$), and (2) there is not another *different* column intersecting the row in a 1 or 0 (i.e., $\not\exists c'\,[C(c') \cdot (c' \neq c) \cdot (0(r,c') + 1(r,c'))]$).

Assuming that a row can have at most one 0, a column $c \in C$ is essential if and only if there is a row $r \in R$ which (1) contains a 1 in column c (i.e., $1(r,c)$), and (2) does not contain any 0 (i.e., $unate\_row(r)$), and (3) there is not another *different* column intersecting the row in a 1 (i.e., $\not\exists c'\,[C(c') \cdot (c' \neq c) \cdot 1(r,c')]$). ■

Theorem 5.27 *Essential columns must be in the solution. Each essential column must then be deleted from the table together with all rows where it has 1's.*

The following computations add essential columns to the solution, delete them from the set of columns and delete all rows in which they have 1's:

$$\begin{aligned} solution(c) \quad ^{1,2}= \quad & solution(c) + ess\_col(c) \\ C(c) \quad ^{1,2}= \quad & C(c) \cdot \neg ess\_col(c) \\ R(r) \quad ^{1,2}= \quad & R(r) \cdot \not\exists c\,[ess\_col(c) \cdot 1(r,c)] \end{aligned}$$

$$\begin{aligned} solution(p) \quad ^{3}= \quad & solution(p) + ess\_col(p) \\ C(p) \quad ^{3}= \quad & C(p) \cdot \neg ess\_col(p) \\ R(c,d) \quad ^{3}= \quad & R(c,d) \cdot \neg ess\_col(c) \end{aligned}$$

Proof: The first two equations move the essential columns from the column set to the solution set. The third equation deletes from the set of rows R all rows intersecting an essential column c in a 1. ■

5.8.6 Unacceptable Columns

Definition 5.21 *A column c is an* **unacceptable column** *if there is a row having a 0 in column c and 2 everywhere else.*

Theorem 5.28 *The set of unacceptable columns can be computed by:*

$$\begin{aligned} unacceptable\_col(c) \quad ^{1}= \quad & C(c) \cdot \exists r\,\{R(r) \cdot 0(r,c) \cdot \not\exists c'\,[C(c') \cdot (c' \neq c) \cdot 0(r,c')]\} \\ & \cdot \not\exists c'\,[C(c') \cdot 1(r,c')]\} \\ unacceptable\_col(c) \quad ^{2}= \quad & C(c) \cdot \exists r\,\{R(r) \cdot 0(r,c) \cdot \not\exists c'\,[C(c') \cdot 1(r,c')]\} \\ unacceptable\_col(p) \quad ^{3}= \quad & C(p) \cdot \exists d\,\{R(p,d) \cdot \not\exists p'\,[C(p') \cdot (p' \supseteq d)]\} \end{aligned}$$

Proof: For column $c \in C$ to be unacceptable, there must be a row $r \in R$ such that (1) it intersects the column c at a 0, and (2) there does not exists another column c' different from c which intersects that row r at a 0 (i.e., $\not\exists c'\,[C(c') \cdot (c' \neq c) \cdot 0(r,c')]$), and (3) no column c' intersects that row r in a 1 (i.e., $\not\exists c'\,[C(c') \cdot 1(r,c')]$). Condition 2 is not needed if we assume that each row contains at most one 0. ■

5.8.7 Unnecessary Columns

Definition 5.22 *A column is an* **unnecessary column** *if it does not have any 1 in it.*

Theorem 5.29 *The set of unnecessary columns can be computed as:*

$$\begin{aligned} unnecessary\_col(c) \quad &^{1,2}= \quad C(c) \cdot \nexists r\,[R(r) \cdot 1(r,c)] \\ unnecessary\_col(p) \quad &^{3}= \quad C(p) \cdot \nexists c,d\,[R(c,d) \cdot (p \supseteq d)] \end{aligned}$$

Proof: A column $c \in C$ is unnecessary if no row $r \in R$ intersects it in a 1. ■

Theorem 5.30 *Unacceptable and unnecessary columns should be eliminated from the table, together with all the rows in which such columns have 0's.*

The table (R, C) is updated according to Theorem 5.22 by setting

$$\begin{aligned} D(c) \quad &^{1,2}= \quad unacceptable\_col(c) + unnecessary\_col(c) \\ D(p) \quad &^{3}= \quad unacceptable\_col(p) + unnecessary\_col(p) \end{aligned}$$

Proof: Obvious. ■

5.9 Other Implicit Covering Table Manipulations

To have a fully implicit binate covering algorithm as described in Section 5.6, we must also compute implicitly a branching column and a lower bound. These computations as well as table partitioning involve solving a common subproblem of finding columns in a table which have the maximum number of 1's.

5.9.1 Selection of Columns with Maximum Number of 1's

Given a binary relation $F(r, c)$ as a BDD, the abstracted problem is to find a subset of c's each of which relates to the maximum number of r's in $F(r, c)$. An inefficient method is to cofactor F with respect to c taking each possible values c_i, count the number of onset minterms of each $F(r, c)|_{c=c_i}$, and pick the c_i's with the maximum count. Instead our algorithm, $Lmax$, traverses each node of F exactly once as shown by the pseudo-code in Figure 5.8.

$Lmax$ takes a relation $F(r, c)$ and the variables set r as arguments and returns the set G of c's which are related to the maximum number of r's in F, together with the maximum count.

```
Lmax(F, r) {
    v = bdd_top_var(F)
    if (v ∈ r)
        return (1, bdd_count_onset(F))
    else { /* v is a c variable */
        (T, count_T) = Lmax(bdd_then(F), r)
        (E, count_E) = Lmax(bdd_else(F), r)
        count = max(count_T, count_E)
        if (count_T = count_E)
            G = ITE(v, T, E)
        else if (count = count_T)
            G = ITE(v, T, 0)
        else if (count = count_E)
            G = ITE(v, 0, E)
        return (G, count)
    }
}
```

Figure 5.8: Pseudo-code for the $Lmax$ operator.

Variables in c are required to be ordered before variables in r. Starting from the root of BDD F, the algorithm traverses down the graph by recursively calling $Lmax$ on its *then* and *else* subgraphs. This recursion stops when the top variable v of F is within the variable set r. In this case, the BDD rooted at v corresponds to a cofactor $F(r,c)|_{c=c_i}$ for some c_i. The minterms in its onset are counted and returned as *count*, which is the number of r's that are related to c_i.

During the upward traversal of F, we construct a new BDD G in a bottom up fashion, representing the set of c's with maximum count. The two recursive calls of $Lmax$ return the sets $T(c)$ and $E(c)$ with maximum counts *count_T* and *count_E* for the *then* and the *else* subgraphs. The larger of the two counts is returned. If the two counts are the same, the columns in T and E are merged by $ITE(v,T,E)$ and returned. If *count_T* is larger, only T is retained as the updated columns of maximum count. And symmetrically for the other case. To guarantee that each node of BDD $F(r,c)$ is traversed once, the results of $Lmax$ and *bdd_count_onset* are memoized in computed-tables. Note that $Lmax$ returns a set of c's of maximum count. If we need only one c, some heuristic can be used to break the ties.

Example To understand how $Lmax$ works, consider the explicit binate table:

	00	01	10	11
00	1	2	1	1
01	2	1	1	2
10	2	1	2	1
11	2	1	2	1

with four rows and four columns. The columns that maximize the number of 1's are the second and the fourth. If the rows and columns are encoded by two Boolean variables each, using the encodings given on top of each column and to the left of each row, the 1 entries of the table are represented implicitly by the relation $F(c,r)$ [4] whose minterms are:

$$\{0000, 1000, 1100, 0101, 1001, 0110, 1110, 0111, 1111\}.$$

The BDD representing F is shown in Figure 5.9. The result of invoking *Lmax* on $F(r,c)$ is a BDD representing the relation $G(c)$ whose minterms are: $\{01, 11\}$, corresponding to the encodings of the second and fourth column.

5.9.2 Implicit Selection of a Branching Column

The selection of a branching column is a key ingredient of an efficient branch-and-bound covering algorithm. A good choice reduces the number of recursive calls, by helping to discover

[4] r and c are swapped in F so that minterms are listed in the order of the BDD variables.

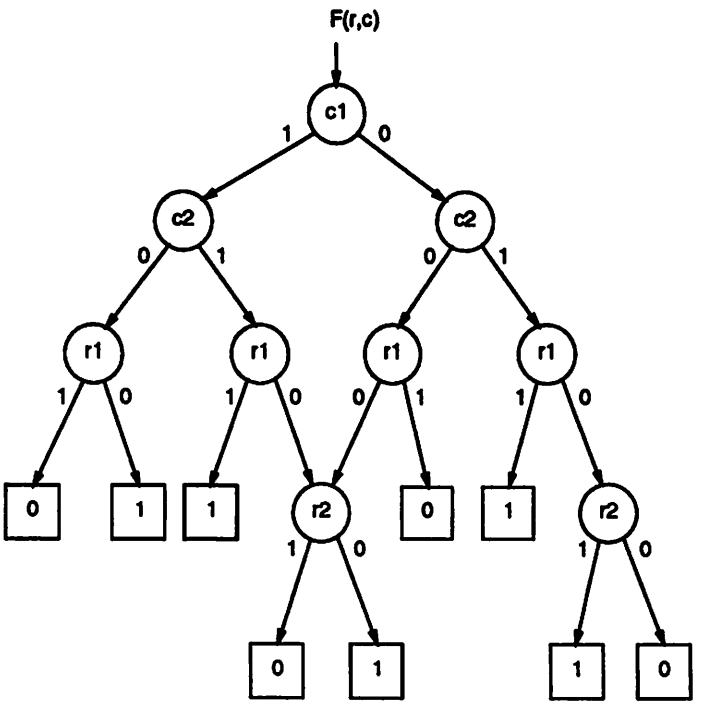

Figure 5.9: BDD of $F(r, c)$ to illustrate the routine $Lmax$.

more quickly a good solution. We adopt a simplified selection criterion: select a column with a maximum number of 1's. By defining $F'(r, c) = R(r) \cdot C(c) \cdot 1(r, c)$ which evaluates true if and only table entry (r, c) is a 1, our column selection problem reduces to one of finding the c related to the maximum number of r's in the relation $F'(r, c)$, and so it can be found implicitly by calling $Lmax(F', r)$. A more refined strategy is to restrict our selection of a branching column to columns intersecting rows of a maximal independent set, because a unique column must eventually be selected from each independent row. A maximal independent set can be computed as follows.

5.9.3 Implicit Selection of a Maximal Independent Set of Rows

Usually a lower bound is obtained by computing a maximum independent set of the unate rows. A maximum independent set of rows is a (maximum) set of rows, no two of which intersect the same column at a 1. Maximum independent set is an NP-hard problem and an approximate one (only maximal) can be computed by a greedy algorithm. The strategy is to select *short unate rows* from the table, so we construct a relation $F''(c, r) = R(r) \cdot unate\_row(r) \cdot C(c) \cdot 1(r, c)$. Variables in r are ordered *before* those in c. The rows with the minimum number of 1's in F'' can be computed by $Lmin(F'', c)$, by replacing in $Lmax$ the expression $max(count\_T, count\_E)$ with

$min(count\_T, count\_E)$. Once a shortest row, $shortest(r)$, is selected, all rows having 1-elements in common with $shortest(r)$ are discarded from $F''(c,r)$ by:

$$F''(c,r) = F''(c,r).\ \not\exists c'\ \{\exists r'\ [shortest(r') \cdot F''(c',r')] \cdot F''(c',r)\}$$

Another shortest row can then be extracted from the remaining table F'' and so on, until F'' becomes empty. The maximum independent set consists of all $shortest(r)$ so selected.

5.9.4 Implicit Covering Table Partitioning

If a covering table can be partitioned into n disjoint blocks, the minimum covering for the original table is the union of the minimum coverings for the n sub-blocks. Let us define the nonempty-entry relation $01(r,c) = 0(r,c) + 1(r,c)$. The implicit algorithm in Figure 5.10 takes a table description in terms of its set of rows $R(r)$, its set of columns $C(c)$ and the nonempty-entry relation $01(r,c)$, partitions it into n disjoint sub-blocks, and return them as n pairs of (R^i, C^i), each corresponding to the rows and columns for the i-th sub-block.

n-way partitioning can be accomplished by successive extraction of disjoint blocks from the table. When the following iteration reaches a fixed point, (R_k, C_k) corresponds to a disjoint sub-block in (R,C).

$$\begin{aligned} R_0(r) &= Lmax(R(r) \cdot C(c) \cdot 01(r,c)\ ,\ c) \\ C_k(c) &= C(c) \cdot \exists r\ \{R_{k-1}(r) \cdot 01(r,c)\} \\ R_k(r) &= R(r) \cdot \exists c\ \{C_k(c) \cdot 01(r,c)\} \end{aligned}$$

This sub-block is extracted from the table (R,C) and the above iteration is applied again to the remaining table, until the table becomes empty.

Given a covering table, a single row $R_0(r)$, which has the maximum number of nonempty entries, is first picked using $Lmax()$. The set of columns $C_1(c)$ intersecting this row at 0 or 1 entries is given by $C(c) \cdot \exists r\ [R_0(r) \cdot 01(r,c)]$ (we want $c \in C$ such that there is a row $r \in R_0$ which intersects c at a 0 or 1). Next we find the set of rows R_1 intersecting the columns in C_1 via nonempty entries, by a similar computation $R(r) \cdot \exists c\ [C_1(c) \cdot 01(r,c)]$. Then we can extract all the rows $R_2(r)$ which intersects $C_1(c)$, and so on. This pair of computations is iteratively applied within the *repeat* loop in Figure 5.10 until no new connected row or column can be found (i.e., $R_k = R_{k-1}$). Effectively, starting from a row, we have extracted a disjoint block (R^1, C^1) from the table, which will later be returned. The remaining table after bi-partition simply contains the rows $R - R^1$ and

```
n_way_partition(R(r), C(c), 01(r, c)) {
    n = 0
    while (R not empty) {
        k = 0
        R_0(r) = Lmax(R(r) · C(c) · 01(r, c))
        repeat {
            k = k + 1
            C_k(c) = C(c) · ∃r {R_{k-1}(r) · 01(r, c)}
            R_k(r) = R(r) · ∃c {C_k(c) · 01(r, c)}
        } until (R_k = R_{k-1})
        R^n = R_k
        C^n = C_k
        R = R − R_k
        C = C − C_k
        n = n + 1
    }
    return {(R^i, C^i) : 0 ≤ i ≤ n − 1}
}
```

Figure 5.10: Implicit n-way partitioning of a covering table.

the columns $C - C^1$. If the remaining table is not empty, we will extract another partition (R^2, C^2) by passing through the outer while loop a second time. If the original table contains n disjoint blocks, the algorithm is guaranteed to return exactly the n sub-blocks by passing through the outer *while* loop n times.

5.10 Experimental Results

In this section we report results of an implementation of the implicit binate covering algorithm, described in the previous sections. We use the benchmarks already introduced in Section 4.7 and concentrate on the examples where prime compatibles are needed to find a minimum solution of the state minimization problem [5]. Here we provide data for a subset of them, sufficient to characterize the capabilities of our prototype program.

Comparisons are made with STAMINA. The binate covering step of STAMINA was run with no row consensus, because row consensus has not yet been implemented in our implicit binate solver. Our implicit binate program currently lacks also routines for Gimpel's reduction rule, that was instead invoked in the version of STAMINA used for comparison. This might sometimes favor STAMINA, but for simplicity we will not elaborate further on this effect. In the near future we will implement beta dominance, row consensus and table partitioning in our package. All run times are reported in CPU seconds on a DECstation 5000/260 with 440 Mb of memory.
The following explanations refer to the tables of results:

- Under table size we provide the dimensions of the original binate table and of its cyclic core, i.e., the dimensions of the table obtained when the first cycle of reductions converges.
- # mincov is the number of recursive calls of the binate cover routine.
- Data are reported with a $*$ in front, when only the first solution was computed.
- Data are reported with a † in front, when only the first table reduction was performed.
- # cover is the cardinality of a minimum cost solution (when only the first solution has been computed, it is the cardinality of the first solution).
- CPU time refers only to the binate covering algorithm. It does not include the time to find the prime compatibles.

[5] Otherwise, a minimum solution of maximal compatibles is closed and therefore is a minimum solution.

5.10.1 Minimizing Small and Medium Examples

With the exception of *ex2*, *ex3*, *ex5*, *ex7*, the examples from the MCNC and asynchronous benchmarks do not require prime compatibles for exact state minimization and yield simple covering problems[6]. Table 5.1 reports those few non-trivial examples. They were all run to full completion, with the exception of *ex2*. In the case of *ex2*, we stopped both programs at the first solution.

FSM	table size (rows x columns) before reduction	table size (rows x columns) after first α reduction	# mincov ISM α	# mincov ISM β	# mincov STAMINA α	# mincov STAMINA β	# cover ISM α	# cover ISM β	# cover STAMINA α	# cover STAMINA β	CPU time (sec) ISM α	CPU time (sec) ISM β	CPU time (sec) STAMINA α	CPU time (sec) STAMINA β
ex2	4418 x 1366	3425 x 1352	*6	*14	*6	*4	*10	*12	*10	*9	*58	*293	*116	*91
ex2	4418 x 1366	3425 x 1352	*6	*14	*6	286	*10	*12	*10	5	*58	*293	*116	2100
ex3	243 x 91	151 x 84	201	37	91	39	4	4	4	4	78	33	0	0
ex5	81 x 38	47 x 31	16	6	10	6	3	3	3	3	4	3	0	0
ex7	137 x 57	62 x 44	38	31	37	6	3	3	3	3	8	12	0	0

Table 5.1: Examples from the MCNC benchmark.

These experiments suggest that

- the number of recursive calls of the binate cover routine (*# mincov*) of ISM and STAMINA is roughly comparable, showing that our implicit branching selection routine is satisfactory. This is an important indication, because selecting a good branching column is a more difficult task in the implicit frame.

- the running times are better for STAMINA except in the largest example, *ex2*, where ISM is slightly faster than STAMINA. This is to be expected because when the size of the table is small the implicit approach has no special advantage, but it starts to pay off scaling up the instances. Moreover, our implicit reduction computations have not yet been fully optimized.

5.10.2 Minimizing Constructed Examples

Table 5.2 presents a few randomly generated FSM's. They generate giant binate tables. The experiments show that ISM is capable of reducing those table and of producing a minimum solution or at least a solution. This is beyond reach of an explicit technique and substantiates the claim that implicit techniques advance decisively the size of instances that can be solved exactly.

[6]Moreover, in the case of the asynchronous benchmark a more appropriate formulation of state minimization requires all compatibles and a different set-up of the covering problem.

FSM	table size (rows x columns) before reduction	table size (rows x columns) after first α reduction	# mincov ISM α	# mincov ISM β	# mincov STAMINA α	# mincov STAMINA β	# cover ISM α	# cover ISM β	# cover STAMINA α	# cover STAMINA β	CPU time (sec) ISM α	CPU time (sec) ISM β	CPU time (sec) STAMINA α	CPU time (sec) STAMINA β
ex2.271	95323 x 96382	0 x 0	1	1	-	-	2	2	-	-	1	55	fails	fails
ex2.285	1 x 121500	0 x 0	1	1	-	-	2	2	-	-	0	0	fails	fails
ex2.304	1053189 x 264079	1052007 x 264079	2	-	-	-	2	-	-	-	463	fails	fails	fails
ex2.423	637916 x 160494	636777 x 160494	*2	-	-	-	*3	-	-	-	*341	fails	fails	fails
ex2.680	757755 x 192803	756940 x 192803	2	-	-	-	2	-	-	-	833	fails	fails	fails

Table 5.2: Random FSM's.

5.10.3 Minimizing FSM's from Learning I/O Sequences

Examples in Table 5.2 demonstrate dramatically the capability of implicit techniques to build and solve huge binate covering problems on suites of contrived examples. Do similar cases arise in real synthesis applications? The examples reported in Table 5.3 answer in the affirmative the question. They are the simplest examples from the suite of FSM's described in Section 4.7.3. It is not possible to build and solve these binate tables with explicit techniques. Instead we can manipulate them with our implicit binate solver and find a solution. In the example *fourr.40*, only the first table reduction was performed.

FSM	table size (rows x columns) before reduction	table size (rows x columns) after first α reduction	# mincov ISM α	# mincov ISM β	# mincov STAMINA α	# mincov STAMINA β	# cover ISM α	# cover ISM β	# cover STAMINA α	# cover STAMINA β	CPU time (sec) ISM α	CPU time (sec) ISM β	CPU time (sec) STAMINA α	CPU time (sec) STAMINA β
threer.20	6977 x 3936	6974 x 3936	*4	*6	*5	*3	*5	*5	*6	*6	*13	*26	*1996	*677
threer.25	35690 x 17372	34707 x 17016	*3	*6	-	-	*5	*6	-	-	*69	*192	fails	fails
threer.30	68007 x 33064	64311 x 32614	*4	*9	-	-	*8	*8	-	-	*526	*770	fails	fails
threer.35	177124 x 82776	165967 x 82038	*8	*9	-	-	*12	*10	-	-	*2296	*2908	fails	fails
threer.40	1.21e6 x 5.29e5	1.15e6 x 5.27e5	*8	-	-	-	*12	-	-	-	*6787	fails	fails	fails
fourr.16	6060 x 3266	5235 x 3162	*2	*3	*3	*3	*3	*3	*4	*4	*6	*23	*1641	*513
fourr.16	6060 x 3266	5235 x 3162	*2	623	*3	377	*3	3	*4	3	*6	9194	*1641	1459
fourr.20	26905 x 12762	26904 x 12762	*2	*4	-	-	*4	*4	-	-	*31	*68	fails	fails
fourr.30	1.40e6 x 5.43e5	1.39e6 x 5.42e5	*2	*5	-	-	*4	*5	-	-	*1230	*1279	fails	fails
fourr.40	6.78e9 x 2.39e9	6.78e9 x 2.39e9	†1	-	-	-	†-	-	-	-	†723	fails	fails	fails

Table 5.3: Learning I/O sequences benchmark.

5.10.4 Minimizing FSM's from Synthesis of Interacting FSM's

Prime compatibles are required only for the state minimization of *ifsm1* and *ifsm2*. For *ifsm1*, ISM can find a first solution faster than STAMINA using α-dominance. But as the table sizes are not very big, the run times ISM take are usually longer than those for STAMINA.

FSM	table size (rows x columns) before reduction	table size (rows x columns) after first α reduction	# mincov ISM α	# mincov ISM β	# mincov STAMINA α	# mincov STAMINA β	# cover ISM α	# cover ISM β	# cover STAMINA α	# cover STAMINA β	CPU time (sec) ISM α	CPU time (sec) ISM β	CPU time (sec) STAMINA α	CPU time (sec) STAMINA β
ifsm1	17663 x 8925	16764 x 8829	*4	2	*10	3	*14	14	*15	14	*388	864	*17582	805
ifsm1	17663 x 8925	16764 x 8829	*4	2	24	3	*14	14	14	14	*388	864	40817	805
ifsm2	1505 x 774	1368 x 672	4	3	41	44	9	9	9	9	136	230	49	3

Table 5.4: Examples from synthesis of interactive FSM's.

Part III

State Minimization of Non-deterministic FSM's

Chapter 6

State Minimization of Non-deterministic FSM's

Non-determinism is a valuable tool in the direct specification of behaviors. In an FSM specification, it captures choices in the behavior to be implemented which a user allows. It can be used both to capture conditions that cannot arise in a certain situation (extending the usage of don't care conditions) and to postpone implementation choices. The most general form of non-determinism can be expressed as an NDFSM. Our goal is to explore different behaviors contained within an FSM specification and choose an optimum one with respect to some cost function, e.g., one with the minimum number of states. In this chapter we address the problem of finding a minimum contained behavior within a stand-alone FSM. In Chapter 7, we consider a feedback composition of FSM's and the problem of finding a minimum well-defined/Moore behavior at an FSM node within a network of FSM's.

The state minimization problem of the general NDFSM will be described in Section 6.1 and some new algorithms related to its minimization are presented in Section 6.2. In Section 6.3, we will describe an implicit algorithm for state minimization of PNDFSM's, a subclass of NDFSM's. This research will appear in [39]. First, we would like to find out if an efficient exact algorithm exists for the state minimization of NDFSM's.

6.1 State Minimization of NDFSM's

Definition 6.1 *A* **non-deterministic FSM** *(NDFSM), or simply an* **FSM,** *is defined as a 5-tuple* $M = \langle S, I, O, T, R \rangle$ *where* S *represents the finite state space,* I *represents the finite input space*

and O represents the finite output space. T is the transition relation defined as a characteristic function $T : I \times S \times S \times O \rightarrow B$. On an input i, the NDFSM at present state p can transit to a next state n and output o if and only if $T(i, p, n, o) = 1$ (i.e., (i, p, n, o) is a transition). There exists one or more transitions for each combination of present state p and input i. $R \subseteq S$ represents the set of reset states.

Is it possible to apply the classical procedure based on computing compatibles to NDFSM's? The answer is: yes, the notions of compatibles and selection of a minimum subset carry through to NDFSM's; but, no, that procedure is not guaranteed to produce a behavior with a minimum number of states. In terms of the theory developed in Chapter 2, some DFSM behaviorally contained in an NDFSM may not corresponds to a closed cover as defined by Definitions 2.25 and 2.26. This fact is illustrated by the counter example in Figure 6.1.

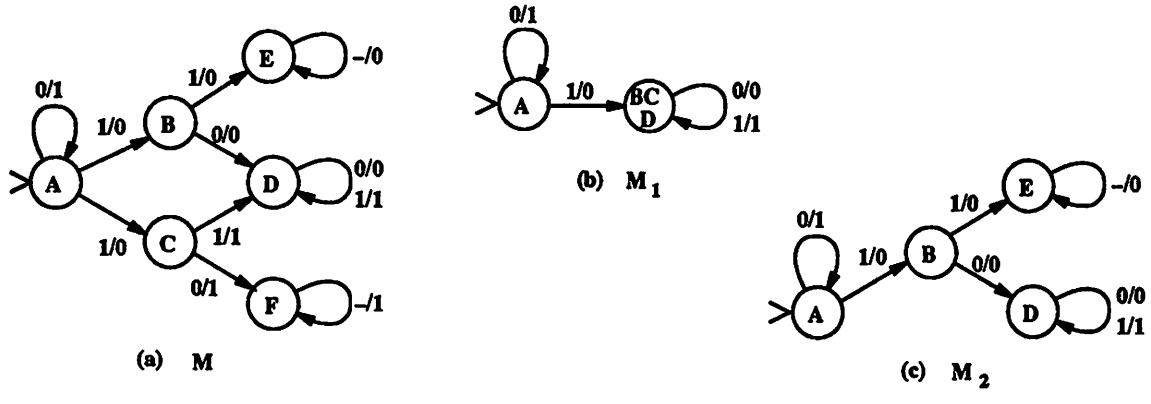

Figure 6.1: A counter example, a) the NDFSM M, b) the minimum state DFSM contained in M, c) one DFSM contained in M found using compatibles.

Example Given the NDFSM M as shown in Figure 6.1a, the minimum state DFSM M_1, as shown in Figure 6.1b, contained in M, cannot be found using compatibles alone. According to Definition 2.36, states B and C are not compatible (and any state set containing B and C cannot form a closed set). As a result, any minimized machine M_2 obtained by compatible-based algorithms will have at least four states, one of the two such minimized machines being shown in Figure 6.1c. However with the non-deterministic transitions into states B and C, we can choose their outgoing transitions in a way as shown in Figure 6.1b such that B and C are merged together as one state. This merged state is compatible to state D. This merging possibility is not explored by compatibility. The minimum state DFSM M_1, whose behavior is contained in the original NDFSM, has only two states.

6.1.1 Generalized Compatibles

By the following definition, we generalize the notion of a compatible (Definition 2.36) which is a set of states, to one that is a set of state sets. Each individual state set contains states that can be merged because they can be non-deterministically reached from a reset state. By merging, we mean that the state set can be considered a new merged state in a reduced machine which on each input, has a transition copied from a state in the set. If the original FSM doesn't have any non-deterministic transition, each such state set will be a singleton. In such a case, each generalized compatible is a set of singleton-states, corresponding to a classical compatible.

In terms of trace sets as defined in Section 2.4, a generalized compatible can be viewed as a compatible set of merged states. Each state set within a generalized compatible corresponds to a merged state. The trace set from a merged state is the union of the trace sets from states in the corresponding state set. The trace set associated with the generalized compatible is the intersection of trace sets from all merged states which correspond to state sets within the generalized compatible.

Definition 6.2 *A set of state sets is a* **generalized compatible** *if for each input sequence, there is a corresponding output sequence which can be produced by at least one state from each state set in the generalized compatible.*

Given an NDFSM, the definition of generalized compatible requires that for all input sequences, there is a common output sequence agreed by at least one state out of each state set in the generalized compatible.

Example {A}, {B}, {C}, {D}, {E}, {F}, {BC}, {BC,D} are generalized compatibles. The enumeration of input sequences for {BC,D} are as follows: $\{B,D\} \xrightarrow{0/0} \{D,D\}$; $\{C,D\} \xrightarrow{1/1} \{D,D\}$; $\{B,D\} \xrightarrow{0/0} \{D,D\} \xrightarrow{0/0} \{D,D\}$; $\{B,D\} \xrightarrow{0/0} \{D,D\} \xrightarrow{1/1} \{D,D\}$; $\{C,D\} \xrightarrow{1/1} \{D,D\} \xrightarrow{0/0} \{D,D\}$; $\{C,D\} \xrightarrow{1/1} \{D,D\} \xrightarrow{1/1} \{D,D\}$; ...

The following theorem shows a recursive characterization of generalized compatibles that expresses the compatibility of a set of state sets in terms of the compatibilities of its sets of next state sets.

Theorem 6.1 *A set K of state sets is a* **generalized compatible** *if and only if for each input i, there exists an output o such that*

1. for each state set in K, its set of transitions under input i and output o is non-empty, and

2. *from the set K of state sets, the set K' of sets of next states under i and o is also a generalized compatible.*

Note the similarity with the generation of classical compatibles (Lemma 2.12), where we require a set of states to be (1) output compatible, and (2) its next state set to be compatible.

Now, the problem of state minimization of NDFSM's can be reduced to one of selecting a minimum subset of generalized compatibles. The selection must satisfy the following covering and closure conditions.

6.1.2 Generalized Covering and Closure Conditions

Definition 6.3 *A set of generalized compatibles* **covers** *the reset state(s) if it contains at least one generalized compatible c such that the set of reset states contains at least one state set in c (i.e., at least one of its state sets is made up entirely of reset states).*

This definition is similar to Definition 2.25 except here each compatible is a generalized compatible. We still require that an element in a selected compatible must behave like a reset state but an element is now a set of states to be merged. To make sure that whatever transition we choose for the merged state, it will still correspond to a transition from a reset state, we require that its corresponding state sets be made up entirely of reset states.

Example Only generalized compatible $\{A\}$ covers the reset state.

The closure condition of a classical compatible requires that the set of next states from the compatible be *contained* in another selected compatible. With our generalized compatibles, we first have to extend this notion of containment to sets of state sets.

Definition 6.4 *A set K of state sets* **contains** *another set K' of state sets if for each state set S' in K', there is state set S in K such that S' contains S.*

Let us define the trace set of a set of state set to be the intersection of the trace sets of the state sets, and the trace set of each state set is in turn the union of the trace sets from the states in the set. If K contain K' according to the Definition 6.4, then the trace set of K contains the trace set of K': For the trace set of K to contain the trace set of K', we require that set containments in the form $\forall S' \in K' \; \exists S \in K \; S' \subseteq S$.

Definition 6.5 *A set $\mathcal{G}$ of generalized compatibles is* **closed** *if for each generalized compatible $K \in \mathcal{G}$, for each input i, there exists an output o such that*

1. *for each state set in K, its set of transitions under input i and output o is non-empty, and*

2. *from the set K of state sets, the set K' of sets of next states under i and o is contained in a generalized compatible of $\mathcal{G}$.*

Example The set of generalized compatible $\mathcal{G} = \{\{A\}, \{BC, D\}\}$ is closed. Closure condition for $\{A\}$ requires the following generalized compatibles to be selected (represented by the clauses): $(\{A\}) \cdot (\{B\} + \{C\} + \{BC\} + \{BC, D\})$. Closure condition for $\{BC, D\}$ requires that $(\{D\} + \{F\}) \cdot (\{D\}) \cdot (\{E\} + \{D\})$. These closure conditions are satisfied by $\mathcal{G}$.

6.1.3 Relationship to Determinization and PNDFSM Minimization

Definition 6.6 *A set of states is a* **mergeable** *if it corresponds to a state label on the determinized state transition graph.*

Example The mergeables of NDFSM M in Figure 6.1a are A, BC, D, E, F, which are state labels on the determinized state graph shown in Figure 6.2.

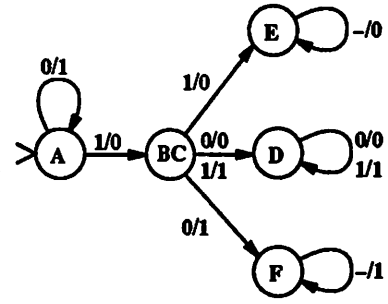

Figure 6.2: Determinized state transition graph of M in Figure 6.1a.

Theorem 6.2 *At least one minimum closed cover consists entirely of generalized compatibles whose state sets are mergeables.*

Example The minimum closed cover {{A},{BC,D}} is made up of mergeables only.

Therefore to form the covering table and solve the covering problem exactly, it is sufficient to generate only the generalized compatibles made up of mergeables. Therefore if one replaces the words "state sets" with "mergeables" in the previous definitions one obtains a more compact representation of the set of generalized compatibles.

6.2 Algorithms for State Minimization of NDFSM's

6.2.1 Determinized Transition Relation and Implicit Subset Construction

Explicit algorithm for subset construction is well-known [32]. Here we outline how subset construction can be performed implicitly.

Given the transition relation $T(i, s, s', o)$ of an NDFSM $M = \langle S, I, O, T, R \rangle$, first we compute the transition relation $T^{det}(i, c, c', o)$ of the determinized PNDFSM M^{det}. A 4-tuple (i, c, c', o) is in relation T^{det} if and only if the set of states c on input i can transit to another set of states c', and simultaneously produce output o. The advantage of using a 1-hot encoding (i.e., positional-set notation) to represent states in the original NDFSM, e.g., s, is that a state in the determinized PNDFSM, e.g., c, (corresponding to a set of states in the NDFSM) can be represented by a minterm in the encoded space. T^{det} can be computed by the following formula:

$$\begin{aligned} T^{det}(i, c, c', o) &= \forall s \{[Tuple_1(s) \cdot (s \subseteq c)] \Rightarrow \exists s' [T(i, s, s', o) \cdot (s' \subseteq c')]\} \\ &\quad \cdot \forall s' \{[Tuple_1(s') \cdot (s' \subseteq c')] \Rightarrow \exists s [T(i, s, s', o) \cdot (s \subseteq c)]\} \\ &\quad \cdot \neg\emptyset(c) \cdot \neg\emptyset(c') \\ &= Union_{s' \rightarrow c'} \{\exists s [(c \supseteq s) \cdot T(i, s, s', o)]\} \cdot \neg\emptyset(c) \cdot \neg\emptyset(c') \end{aligned}$$

Given a 4-tuple (i, c, c', o), the first clause on the right requires that for each singleton state s (i.e., $Tuple_1(s) = 1$; see Section 3.7.4) contained in c, there is a next state s' according to T which is contained in c'. As a result, the next state set of c is a subset of c'. With also the second clause, the 4-tuples in the relation will be such that c' is exactly the next state set of c on input i and output o. Finally, we eliminate all 4-tuples expressing the fact that the empty state set can transit to the empty set under any input/output combination. Alternatively, T^{det} can be computed using the set $Union$ operator (given by Lemma 3.19) as shown in the second formula.

The power of the above computation is that we have effectively determinized the NDFSM M into the PNDFSM $M^{det} = \langle 2^S, I, O, T^{det}, r^{det} \rangle$ where the new reset state is the union of reset states in the NDFSM, $r^{det}(c) = Union_{s \rightarrow c} R(s)$. Compared with explicit subset construction, no iteration nor state graph traversal is needed.

The above relation $T^{det}(i, c, c', o)$, derived from the transition relation T, is useful in the computation of compatibles and closure conditions for PNDFSM's in Section 6.4. T^{det} contains many 4-tuples, as c can be any output compatible set of states. During compatible generation, T^{det} will be restricted to relation τ by forcing c and c' to be compatibles:

$$\tau(i, c, c', o) = T^{det}(i, c, c', o) \cdot C(c) \cdot C(c')$$

If some applications other than state minimization need to know the reachable state space $S^{det} \subseteq 2^S$ of the determinized PNDFSM, it can be computed by the following fixed point computation:

- $S_0^{det}(c) = r^{det}(c)$
- $S_{k+1}^{det}(c) = S_k^{det}(c) + [c' \rightarrow c] \exists c, i, o \{S_k^{det}(c) \cdot T^{det}(i, c, c', o)\}$

The above iteration can terminate when for some j, $S_{j+1}^{det} = S_j^{det}$ and the least fixed point is reached. The set of reachable states of the determinized PNDFSM is given by $S^{det}(c) = S_j^{det}(c)$.

6.2.2 Exact Algorithms for State Minimization

We do not yet know how to implicitize the algorithm described in Section 6.1. The main difficulty is to find a compact representation for sets of generalized compatibles. Given that generalized compatibles are already sets of sets of states, there is one more level of set construction complexity than for state minimization of ISFSM's or PNDFSM's. If such a representation exists, Theorem 6.1 gives a constructive definition of generalized compatible, and the covering and closure conditions in Section 6.1.2 can be solved as a binate covering problem.

As an alternative, suggested in [83], one can convert any NDFSM to a PNDFSM (Theorem 2.9) by an implicit determinization (via subset construction) step as described in Section 6.2.1 and then apply to the PNDFSM our implicit (or any) state minimization algorithm which will be presented in Section 6.3. It goes without saying that subset construction may introduce a blow-up in the number of original states; this can hurt the efficiency of the implicit PNDFSM state minimizer whose computations are on a variable support whose cardinality is linearly proportional to the number of states of the PNDFSM. It is an open problem whether a better procedure can be devised, or instead this exponential blow-up is intrinsic to the problem of minimizing NDFSM's. It must also be stressed that we do not have yet good sources of general NDFSM's in sequential synthesis, while the work in [83] has shown the pivotal importance of PNDFSM's in the synthesis of interconnected FSM's.

6.2.3 Heuristic Algorithms for State Minimization

It has been shown that exact minimization of NDFSM's requires computation of the generalized compatibles, instead of the usual compatibles. If we restrict our attention only to the set of compatibles (against generalized compatibles), the algorithm given in Section 6.3 for exact state minimization of PNDFSM's will still serve as a heuristic algorithm for NDFSM minimization.

The PNDFSM minimization algorithm is a heuristic for the state minimization of NDFSM, because the algorithm chooses only from a subset of the generalized compatibles, namely the classical compatibles. For the NDFSM example in Figure 6.1, such a heuristic algorithm cannot find the 2-state reduced machine M_1 (as this solution contains the generalized compatible $\{BC, D\}$) but will only find the 4-state machine M_2. However, the heuristic algorithm is guaranteed to find at lease one solution made up entirely of classical compatibles, because the original NDFSM is such a candidate solution (made up of compatibles that are singletons). The hope is that most non-determinism expressed in an NDFSM is pseudo non-deterministic in nature, and as a result, the heuristic can find near optimum solutions most of the time.

6.3 State Minimization of PNDFSM's

An NDFSM is a pseudo non-deterministic FSM (PNDFSM) if for each triple $(i, p, o) \in I \times S \times O$, there is a unique state n such that $T(i, p, n, o) = 1$. It is non-deterministic because for a given input and present state there may be more than one output and next state; it is called pseudo non-deterministic because edges carrying different outputs must go to different next states.

Definition 6.7 *A* **pseudo non-deterministic FSM** *(PNDFSM) is a 6-tuple* $M = \langle S, I, O, \delta, \Lambda, R \rangle$. *$\delta$ is the next state function defined as* $\delta : I \times S \times O \rightarrow S$ *where each combination of input, present state and output is mapped to a unique next state.* Λ *is the output relation defined by its characteristic function* $\Lambda : I \times S \times O \rightarrow B$ *where each combination of input and present state is related to one or more outputs.* $R \subseteq S$ *represents the set of reset states.*

Explicit algorithms for exact state minimization for PNDFSM's have been proposed by Watanabe *et al.* in [86] and by Damiani in [20]. In this section, we contribute the first fully implicit algorithms for exact state minimization of PNDFSM's. The theory for PNDFSM state minimization has already been given in Chapter 2. Unlike NDFSM minimization, it has been proved in Theorem 2.17 that the PNDFSM state minimization problem can be reduced to the

problem of finding a minimum closed cover. As a result, the algorithm for state minimization of PNDFSM's consists of two steps: compatible generation and binate covering. It is more complicated than the one for ISFSM minimization [38] because the definition of compatibles and the conditions for a closed cover are more complex.

Example The PNDFSM M_p shown in Figure 6.3 will be used in this section to illustrate various concepts. It is a PNDFSM because on input 1, state D can output 0 and transits to state B, or outputs 1 and transits to state A.

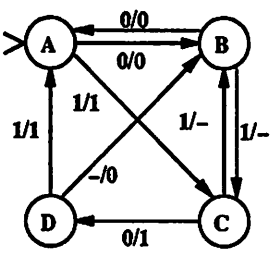

Figure 6.3: A PNDFSM, M_p.

This section is divided into two parts: the first half gives definitions and theorems relating to the generation of compatibles, prime compatibles, covering and closure conditions for state minimization of PNDFSM's. The second half describes in more detail how the prime compatible generation step and the binate covering step can be performed implicitly.

6.3.1 Compatibles

The following theorem serves as an equivalent, constructive definition of compatibles (*cf.* Definition 2.36). The theorem yields an implicit compatible generation procedure in Section 6.4.1.

Theorem 6.3 *A set c of states is a* **compatible** *if and only if for each input i, there exists an output o such that*

1. *each state in c has a transition under input i and output o, and*

2. *from the set c of states, the set c' of next states under i and o is also a compatible.*

Example A, B, C, D, AB are compatibles of M_p of Figure 6.3. AB is a compatible because on input 0, it loops back to itself, and on input 1, it goes to C which is also a compatible.

6.3.2 Covering and Closure Conditions

Definition 6.8 *Given a compatible c, p_c is the positive literal to signify that compatible c is selected, and $\overline{p_c}$ is the negative literal to signify that compatible c is not selected.*

Definition 6.9 *A set of compatibles* **covers** *the reset state(s) if at least one selected compatible contains a reset state.*

Example The covering condition for M_p requires either compatible A or compatible AB be selected. It can be expressed by the simple clause $(p_A + p_{AB})$ where the positive literal p_A evaluates to 1 if and only if compatible A is selected.

Definition 6.10 *A set C of compatibles is* **closed** *if for each compatible $c \in C$, for each input i, there exists an output o such that*

1. *each state in c has a transition under input i and output o, and*
2. *from the set c of states, the set d of next states under i and o is contained in a compatible in C.*

Definition 6.11 *Assuming a compatible c is selected, the* **closure condition** *for c, denoted by $closure(c)$, is a logic formula expressing the requirement that some other compatibles be selected according to Definition 6.10.*

Example Consider the closure condition for compatible D. On input 0, D transits to B. For state B to be in a selected compatible, we must select either compatible B or AB. On input 1, D can (output 1,) transit to A *or* (output 0,) transit to B. We must select a compatible which contains either A or B, i.e., we must select either compatible A or B or AB. Thus, the closure condition for D is the conjunction of disjunctions $(p_B + p_{AB}) \cdot (p_A + p_B + p_{AB})$. Similarly, closure condition for A is $(p_B + p_{AB}) \cdot (p_C)$, for B is $(p_A + p_{AB}) \cdot (p_C)$, for C is $(p_D) \cdot (p_B + p_{AB})$ and for compatible AB is (p_C).

During binate covering, the covering table should guarantee that for each compatible c, either c is not selected (i.e., $\overline{p_c}$ is true), or its closure condition $closure(c)$ is satisfied. In other words, $\overline{p_c} + closure(c)$. This expression can be represented as a conjunction of binate clauses which will be discussed in more detail later. Here we'll introduce the concept through the example.

Example For compatible D, the binate covering table should require that

$$\overline{p_D} + closure(D) = \overline{p_D} + (p_B + p_{AB}) \cdot (p_A + p_B + p_{AB}) = (\overline{p_D} + p_B + p_{AB}) \cdot (\overline{p_D} + p_A + p_B + p_{AB})$$

and actually it can be simplified to $(\overline{p_D} + p_A + p_B + p_{AB})$.

Here we restate a main theorem for state minimization of PNDFSM's, Theorem 2.17.

Theorem 6.4 *The state minimization problem of a PNDFSM reduces to the problem of finding a minimum set of compatibles that covers the reset state(s) and is closed.*

6.3.3 Prime Compatibles

For runtime efficiency, we compute the set of prime compatibles which is a subset of the compatibles. Here we restate its related definitions and theorems from Section 2.9.

Definition 6.12 *A compatible c'* **prime dominates** *a compatible c if for each minimum closed cover containing c, the selection with c replaced by c' also corresponds to a minimum closed cover.*

Definition 6.13 *A compatible c is a* **prime compatible** *if there does not exists another compatible c' such that c' prime dominates c.*

Theorem 6.5 *There exists a minimum closed cover made up entirely of prime compatibles.*

The above theorem justifies our exact state minimization algorithm to consider only prime compatibles.

A sufficient condition for prime dominance (Definition 6.12) is given by the following theorem:

Theorem 6.6 *A compatible c'* **prime dominates** *a compatible c if*

1. *if $(c \cap R) \neq \emptyset$ then $(c' \cap R) \neq \emptyset$, and*
2. *the closure condition for c implies* [1] *the closure condition for c', and*
3. *$c' \supset c$.*

Proof: Assume that c' does not prime dominate c, i.e., there is a minimum closed cover containing c which is not any more a closed cover when c is replaced by c' (Definition 6.13). We show that at least one of the above three conditions is false.

[1] Condition A implies condition B if and only if the satisfaction of condition A automatically guarantees the satisfaction of condition B. In other words, A is *not less restrictive* than B.

Consider any set of compatibles C such that $C \cup \{c\}$ is a minimum closed cover [2]. As $C \cup \{c\}$ and $C \cup \{c'\}$ have the same cardinality, in order that $C \cup \{c'\}$ is not a minimum closed cover, either (1) $C \cup \{c'\}$ does not cover the reset state(s) or (2) $C \cup \{c'\}$ is not closed. For case (1), $C \cup \{c\}$ is a cover but $C \cup \{c'\}$ is not a cover if and only if $(c \cap R) \neq \emptyset$ and $(c' \cap R) = \emptyset$, i.e., condition 1 of the above theorem is false. For case (2), $C \cup \{c\}$ is closed but $C \cup \{c'\}$ is not closed if one of the two situations arises: (2a) C satisfies the closure condition for c but not the closure condition for c'. This happens if and only if condition 2 is false. (2b) c is needed to satisfy the closure condition for some compatible in C, but c' does not satisfy such a condition. This is the case only if $c' \not\supset c$, i.e., condition 3 is false. ■

The converse of the theorem is not true in general, because condition 3 is a sufficient condition, but not a necessary condition, for case (2b) above.

Example Compatible AB prime dominates compatible B because all conditions of Theorem 6.6 are met. In particular, closure condition for B implies closure condition for AB because $[(p_A + p_{AB}) \cdot (p_C)] \Rightarrow [p_C]$. Similarly, AB dominates A. As a result, the prime compatibles are AB, C, D.

6.3.4 Logical Representation of Closure Conditions

We now construct a set of logical clauses expressing the closure requirement that a next state set d is contained in at least one selected compatible, as stated in Definition 6.10. Since we will generate the set of prime compatibles, we express also part 2 of Theorem 6.6 that refers to the implication between closure conditions.

The notion of next state sets d is important for expressing closure conditions and testing prime dominance.

Definition 6.14 *$d_{c,i,o}$ is the set of next states from compatible c under input i and output o.*

Given a triple (c, i, o), the set $d_{c,i,o}$ is unique in a PNDFSM. We associate to each $d_{c,i,o}$ a clause whose positive literals are the prime compatibles that contain $d_{c,i,o}$. This clause will be part of the binate clause representing the closure condition for compatible c. For simplicity in notation, we designate by $d_{c,i,o}$ both the set of next states and the clause associated to it. It will be clear from the context which one it is meant.

Example The next state set from D on input 0 and output 0, $d_{D,0,0}$ is B and it corresponds to the clause $(p_B + p_{AB})$. $d_{D,1,1}$ is A and it corresponds to the clause $(p_A + p_{AB})$. $d_{D,1,0}$ is B and it corresponds to the clause $(p_B + p_{AB})$.

[2] It is possible that no such C exists, and the theorem is trivially true.

The following property is key to evaluating implications between logical clauses by set containments. The latter can be performed implicitly as $d_{c,i,o}$ is represented as a positional-set in Section 6.4.2.

Theorem 6.7 *If the set of next states* $d_{c',i',o'} \supseteq$ *the set of next state set* $d_{c,i,o}$, *then clause* $d_{c',i',o'} \Rightarrow$ *clause* $d_{c,i,o}$.

Proof: If the set of next states $d_{c',i',o'} \supseteq$ the set of next states $d_{c,i,o}$, then each prime compatible that contains $d_{c',i',o'}$ contains also $d_{c,i,o}$. Since each literal in a clause d is a prime compatible that contains the next state set d, it means that the clause $d_{c,i,o}$ has all the literals of the clause $d_{c',i',o'}$ and so each assignment of literals that satisfies the clause $d_{c',i',o'}$ satisfies also the clause $d_{c,i,o}$, i.e., clause $d_{c',i',o'} \Rightarrow$ clause $d_{c,i,o}$. ■

The converse does not hold.

For a PNDFSM, a set of compatible states c under an input i may go to different sets of next states depending on the choice of output o. For at least one choice of o, the corresponding next state set $d_{c,i,o}$ must be contained in some selected compatible. This is expressed by the clause (or disjunctive clause or disjunction), $disjunct(c, i)$, defined as:

$$disjunct(c, i) \quad = \quad \exists o \in \text{outputs at } c \text{ under } i, \quad d_{c,i,o}$$

Example $disjunct(D, 1)$ represents the clause $(d_{D,1,0} + d_{D,1,1}) = (p_A + p_B + p_{AB})$.

For a PNDFSM, the closure condition for a compatible c, denoted by $closure(c)$, has the form of a conjunction of disjunctive clauses. According to Definition 6.10, the conjunction is over all inputs i, while the disjunction is over specified outputs o, such that $\Lambda(i, s, o) = 1$. Given a compatible c, the following product of disjunctions must be satisfied (one disjunction per input):

$$closure(c) \quad = \quad \forall i \in \text{inputs}, \quad disjunct(c, i)$$

In summary, the closure condition for compatible c is fulfilled if and only if for each input i, there is an output o such that the next state set $d_{c,i,o}$ from compatible c under input i and output o is contained in a selected compatible. In logical terms, the closure condition for c is fulfilled if and only if the product-of-sums

$$closure(c) \quad = \quad \forall i \in \text{inputs } \exists o \in \text{outputs at } c \text{ under } i, \quad d_{c,i,o} \tag{6.1}$$

is satisfied. These closure conditions are tested against a certain selection of compatibles.

Example Closure condition for compatible D is

$$closure(D) = (d_{D,0,0} + d_{D,0,1}) \cdot (d_{D,1,0} + d_{D,1,1}) = (p_B + p_{AB}) \cdot (p_A + p_B + p_{AB}).$$

We present now the prime dominance condition of Theorem 6.6. Part 1 and 3 are already expressed as simple logic formulas. Notice that the implication between closure conditions mentioned in part 2 of Theorem 6.6 translates exactly to logical implication ($\Rightarrow$) between clauses, i.e., given compatibles c and c', the closure condition for c implies the closure condition for c' if and only if $closure(c) \Rightarrow closure(c')$. This implication cannot be readily tested by first evaluating each closure condition according to Equation 6.1, because each closure condition is in a product of sums form. We would like to express it in terms of set containment between next state sets using Theorem 6.7. What follows gives such a useful characterization of the formula $closure(c) \Rightarrow closure(c')$.

We first prove two useful lemmas for manipulating logical clauses.

Lemma 6.8 $[\forall x\ F(x)] \Rightarrow [\forall x'\ F'(x')]$ *if and only if* $\forall x'\ \exists x\ [F(x) \Rightarrow F'(x')]$.

Proof: By using some fundamental validities of logic:

$$\begin{aligned} [\forall x\ F(x)] \Rightarrow [\forall x'\ F'(x')] \quad & \text{if and only if} \quad \forall x'\ [\forall x\ F(x) \Rightarrow F'(x')] \\ & \text{if and only if} \quad \forall x'\ \exists x\ [F(x) \Rightarrow F'(x')]. \end{aligned}$$

■

Lemma 6.9 $[\exists x'\ F'(x')] \Rightarrow [\exists x\ F(x)]$ *if and only if* $\forall x'\ \exists x\ [F'(x') \Rightarrow F(x)]$.

Proof: By using some fundamental validities of logic:

$$\begin{aligned} [\exists x'\ F'(x')] \Rightarrow [\exists x\ F(x)] \quad & \text{if and only if} \quad \forall x'\ [F'(x') \Rightarrow \exists x\ F(x)] \\ & \text{if and only if} \quad \forall x'\ \exists x\ [F'(x') \Rightarrow F(x)]. \end{aligned}$$

■

Theorem 6.10 *Given compatibles c and c',*

$$closure(c) \Rightarrow closure(c') \quad \textit{if and only if} \quad \forall i'\ \exists i\ [disjunct(c, i) \Rightarrow disjunct(c', i')].$$

Proof: Substituting $x = i, x' = i', F(x) = disjunct(c, i), F'(x') = disjunct(c', i')$ in Lemma 6.8, one gets that

$$\forall i\ disjunct(c, i) \Rightarrow \forall i'\ disjunct(c', i') \text{ if and only if } \forall i' \exists i\ [disjunct(c, i) \Rightarrow disjunct(c', i')].$$

The former is by definition $closure(c) \Rightarrow closure(c')$. ■

Now that we have expressed implication between closure conditions in terms of implication between disjunctive clauses, the following theorem gives a useful characterization of the formula $disjunct(c', i') \Rightarrow disjunct(c, i)$.

Theorem 6.11 *Given compatibles c', c and inputs i', i,*

$$disjunct(c', i') \Rightarrow disjunct(c, i) \quad \textit{if and only if} \quad \forall o' \exists o\ [d_{c',i',o'} \Rightarrow d_{c,i,o}].$$

Proof: Substituting $x = o, x' = o', F(x) = d_{c,i,o}, F'(x) = d_{c',i',o'}$ into Lemma 6.9, one gets

$$\exists o'\ d_{c',i',o'} \Rightarrow \exists o\ d_{c,i,o} \quad \text{if and only if} \quad \forall o' \exists o\ [d_{c',i',o'} \Rightarrow d_{c,i,o}].$$

The former is by definition $disjunct(c', i') \Rightarrow disjunct(c, i)$. ■

By substituting Theorem 6.11 into Theorem 6.10 and using Theorem 6.7, we have expressed the implication between closure conditions of two compatibles (i.e., part 2 of Theorem 6.6) in terms of a logic formula on the next state sets from the two compatibles.

Theorem 6.12 *If* $\forall i' \exists i \forall o' \exists o\ (d_{c',i',o'} \supseteq d_{c,i,o})$ *then* $closure(c) \Rightarrow closure(c')$.

Proof: By Theorems 6.11, 6.10 and 6.7. ■

The following two theorems simplify the closure conditions. In our implicit algorithm, they are applied before the implication between the conditions is computed.

Theorem 6.13 *Given a compatible c and inputs i' and i, if $disjunct(c, i') \Rightarrow disjunct(c, i)$, then $disjunct(c, i)$ can be omitted from the conjunction $closure(c)$ because of the existence of $disjunct(c, i')$.*

Proof: If $disjunct(c, i') \Rightarrow disjunct(c, i)$, the conjunction of $disjunct(c, i')$ and $disjunct(c, i)$ is simply $disjunct(c, i')$. Therefore $disjunct(c, i)$ can be omitted from the conjunction $closure(c)$. ■

Theorem 6.14 *A set of next states d is not needed to be part of the clause $disjunct(c, i)$, if*

1. *d is a singleton reset state*[3], *or*

2. *$d \subseteq c$, or*

3. *$d \supseteq d'$ if d' is part of $disjunct(c, i)$.*

Proof: (1) If d is a reset state, the covering condition would automatically imply this closure condition expressed by d, and thus the latter is not needed and, even more, the closure condition is vacuously true. (2) d expresses the condition that if c is chosen, a selected compatible must contain the state set d. If $d \supseteq c$, c is such a compatible, and the closure condition is vacuously true. (3) $d \supseteq d'$ by Theorem 6.7 means $d \Rightarrow d'$. The disjunction of d and d' is simply d', so d can be omitted from $disjunct(c, i)$. ■

The order the next state sets are pruned in $disjunct(c, i)$ is important, especially if these pruning rules are executed implicitly (i.e., simultaneously). For proper removal of next state sets, one should find all the d's that satisfy condition 1 or 2 first and remove them from $disjunct(c, i)$. Then on a separate step remove all the d's containing other d' according to condition 3.

6.4 Implicit State Minimization Algorithm for PNDFSM's

In this section, we will show how the state minimization algorithm described in Section 6.3 can be implicitized.

First, we outline the differences between the state minimization algorithm of PNDFSM's and the state minimization algorithm for ISFSM's presented in Chapter 4. Theorem 4.3 does not hold for PNDFSM's. The fact that a set of states may not be a compatible even though it is pairwise compatible is illustrated by the following counter example. As a result, Corollary 4.4 also doesn't hold for PNDFSM's and the set of compatibles cannot be generated from the set of incompatible pairs as in Chapter 4. In addition, the computations for closure conditions as well as prime dominance are more complicated than those for ISFSM's.

Example The following PNDFSM has three states $\{A, B, C\}$, no input and an output with three values $\{x, y, z\}$. All state pairs are compatibles but the set ABC is not a compatible because they cannot agree on an output in one transition.

[3]Condition 1 is valid only if a unique reset state is specified.

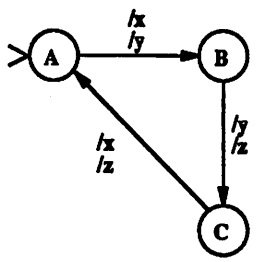

Figure 6.4: A PNDFSM which doesn't have a compatible ABC.

6.4.1 Implicit Generation of Compatibles

As we cannot generate compatibles from incompatible pairs, we have to start with output compatibles (i.e., state sets) of arbitrary cardinalities. First we compute the transition relation T^{det} between sets of states, using the implicit procedure described in Section 6.2.1.

Theorem 6.15 *The set C of compatibles of an NDFSM can be found by the following fixed point computation:*

- $\tau_0(i, c, c') = \exists o\, T^{det}(i, c, c', o)$
- *Initially assume all subsets of states to be compatible:* $C_0(c) = 1$,
- *By Theorem 6.3,*
  - $\tau_{k+1}(i, c, c') = \tau_k(i, c, c') \cdot C_k(c')$
  - $C_{k+1}(c) = \forall i\, \exists c'\, \tau_{k+1}(i, c, c')$

The iteration can terminate when for some j, $C_{j+1} = C_j$, and the greatest fixed point has been reached. The set of compatibles is given by $C(c) = C_j(c)$ and the transition relation on the compatibles is $\tau(i, c, c') = \tau_{j+1}(i, c, c') \cdot C_j(c)$.

6.4.2 Implicit Generation of Prime Compatibles and Closure Conditions

In our implicit framework, we represent each next state set as a positional-set d. The fact that a next state set d is part of $disjunct(c, i)$ can be expressed by the transition relation on compatibles, $\tau(i, c, d)$. The following computation will prune away next state sets d that are not necessary according to Theorem 6.14, and the result is represented by the following relation B.

Theorem 6.16 *The disjunctive conditions can be computed by the following relation B:*

$$A(c,i,d) = ITE(\,\exists d\,\{\tau(i,c,d)\cdot[R(d)+(d\subseteq c)]\},\ \emptyset(d),\ \tau(i,c,d)\,)$$

$$B(c,i,d) = Minimal_d(A(c,i,d))$$

Proof: The first equation corresponds to conditions 1 and 2 of Theorem 6.14. Given a compatible c and an input i, *if* there exists a d which is a next state set from c under i such that $R(d)+(d\subseteq c)$ is true, *then* the $disjunct(c,i)$ is set to the empty set $\emptyset(d)$, *else* we keep the original d in the relation $\tau(i,c,d)$. The second equation tests condition 3 and prunes all the d's that are not minimal (i.e., containing some other d' that is part of $disjunct(c,i)$). ■

In summary $\tau(i,c,d)$ represents the set of disjunctive clauses, while $B(c,i,d)$ represents the pruned set of disjunctive clauses: d is in the relation B with (c,i) if and only if d is part of the disjunctive clause for c under i after pruning.

The following theorem computes the set of disjunctive clauses according to Theorems 6.11 and 6.7, that are used to express the closure conditions. Then the set of prime compatibles is computed according to Theorem 6.6.

Theorem 6.17 *If* $D(c',i',c,i) = \forall d'\,\{B(c',i',d') \Rightarrow \exists d\,[B(c,i,d)\cdot(d'\supseteq d)]\}$,
then $disjunct(c',i') \Rightarrow disjunct(c,i)$.

The set of prime compatibles can be computed by:

$$PC(c) = C(c)\cdot \not\exists c'\{C(c')\cdot[\exists s\,(R(s)\cdot(s\subseteq c)) \Rightarrow \exists s'\,(R(s')\cdot(s'\subseteq c'))]\cdot\forall i'\exists i\,(D(c,i,c',i'))\cdot(c'\supset c)\}$$

Proof: To evaluate $disjunct(c',i') \Rightarrow disjunct(c,i)$, by Theorems 6.11 and 6.7, it is sufficient to check

$$\forall o' \in \text{ outputs at } c' \text{ under } i'\ \exists o \in \text{ outputs at } c \text{ under } i\ [d_{c',i',o'} \supseteq d_{c,i,o}].$$

In other words, we want to check that for all next state set d' from compatible c' on input i' (on some output o'), there exists a next state set d from compatible c on input i (on some output o) such that d' contains d. This corresponds to the condition

$$\forall d'\,\{B(c',i',d') \Rightarrow \exists d\,[B(c,i,d)\cdot(d'\supseteq d)]\}$$

Therefore D is a sufficient condition for $disjunct(c',i') \Rightarrow disjunct(c,i)$.

The second equation defines $PC(c)$ as the set of non-dominated primes. The right subformula within $\{\}$ expresses the three conditions in Theorem 6.6. A positional-set c has a non-empty

intersection with the set of reset states R if and only if there exists a reset state $\exists s\ R(s)$ such that $s \subseteq c$. For condition 1, $\exists s\ (R(s) \cdot (s \subseteq c)) \Rightarrow \exists s'\ (R(s') \cdot (s' \subseteq c'))$ requires that $(c' \cap R) \neq \emptyset$ if $(c \cap R) \neq \emptyset$. By Theorem 6.10, condition 2 of Theorem 6.6 is checked by $\forall i'\ \exists i\ D(c', i', c, i)$ according to the first part of this theorem. Condition (3) is simply $(c' \supset c)$. ■

The following theorem computes the pruned set of disjunctive clauses according to Theorems 6.13 and 6.17. They will be used in the next subsection to set up the binate rows of the covering table.

Theorem 6.18 *Given a compatible c, the inputs i's associated with c which are involved in non-trivial disjunctive clauses are expressed by the following relation E:*

$$E(c,i) = \not\exists i'\ [(i \neq i') \cdot D(c,i',c,i)] + \exists i' cproject_i(D(c,i',c,i) \cdot D(c,i,c,i'))$$

And the corresponding pruned set of disjunctive clauses is given by relation I:

$$I(c,i,d) = B(c,i,d) \cdot PC(c) \cdot E(c,i) \cdot \neg\emptyset(d)$$

Proof: Given a compatible c, Theorem 6.13 states that if $disjunct(c,i') \Rightarrow disjunct(c,i)$ then $disjunct(c,i)$ can be omitted from $closure(c)$. And $disjunct(c,i') \Rightarrow disjunct(c,i)$ if the two pairs are in relation $D(c,i',c,i)$, according to by Theorem 6.17. The first term $\not\exists i'\ [(i \neq i') \cdot D(c,i',c,i)]$ deletes all pairs (c,i) such that there is an input i' where $(i' \neq i)$ such that $disjunct(c,i') \Rightarrow disjunct(c,i)$. But this would eliminate two many (c,i) pairs because it is possible that $(i' \neq i)$, and moreover $disjunct(c,i') \Rightarrow disjunct(c,i)$ and $disjunct(c,i) \Rightarrow disjunct(c,i')$ are both true. Such pairs are defined by $D(c,i',c,i) \cdot D(c,i,c,i')$. In such a case, we must choose and retain exactly one of the two. A unique (c,i) out of each set of "co-implying" pairs is chosen as representative by the BDD *cproject* operator. And the representative is added back to relation E by the last term of the first equation.

For the second equation, the pruned set of disjunctive clauses contains the clauses in $B(c,i,d)$, constrained to have compatibles c that are primes in $PC(c)$, and pairs (c,i) given by relation E. Also, triples with empty set d are vacuously true clauses, and thus are pruned away. ■

6.4.3 Implicit Binate Table Covering

Selection of prime compatibles is performed by the implicit binate covering solver described in Chapter 5. In particular, we use the binate table solver which assumes each row has at most one 0. To use the solver, one has to specify four BDD's: two characteristic functions Col

and *Row* representing a set of column labels and a set of row labels respectively; and two binary relations 1 and 0, one relating columns and rows that intersect at a 1 in the table, and another relating columns and rows that intersect at a 0.

Similar to the case for ISFSM's, each prime compatible corresponds to a single column labeled p in the covering table. So the set of column labels, $Col(p)$, is given by:

$$Col(p) = PC(p)$$

Each row can be labeled by a pair (c, i) because each binate clause originates from the closure condition for a compatible $c \in PC$ under an input i. And the covering condition for a reset state is expressed by a single unate clause, to which we assign a row label $(c, i) = (\emptyset, \emptyset)$. c is chosen to be the empty set to avoid conflicts with the labels of the binate rows, while the choice of $i = \emptyset$ is arbitrary. The set of row labels, $Row(c, i)$, is given by a binate part and a unate part:

$$Row(c, i) = \exists d\, I(c, i, d) + \emptyset(c) \cdot \emptyset(i)$$

Each binate clause associated with a compatible c and an input i expresses the condition that for at least one output o, the next state set must be contained in a selected compatible d. The corresponding next state relation is $I(c, i, d)$.

Next, let us consider the table entries relations $\mathtt{1}(c, i, p)$ and $\mathtt{0}(c, i, p)$. If (c, i) labels a binate row, the expression $\exists d\, [(p \supseteq d) \cdot I(c, i, d)]$ evaluates to true if and only if the table entry is a 1 at the intersection of the row labeled (c, i) and the column labeled p, i.e., the row can be satisfied if next state set d is contained in a selected compatible p. There is an entry 0 at column p if $(p = c)$, i.e., the row can also be satisfied by not selecting a column labeled c.

The row labeled by $(\emptyset, \emptyset)$ represents the disjunction of compatibles p each of which contains at least a reset state $R(s)$. On such a row, a table entry is a 1 if and only if $\exists s\, [\emptyset(c) \cdot \emptyset(i) \cdot R(s) \cdot (s \subseteq p)]$.

As a summary, the inference rules for table entries given a row (c, i) and a column p are:

$$\mathtt{1}(c, i, p) \stackrel{\text{def}}{=} \exists d\, [(p \supseteq d) \cdot I(c, i, d)] + \exists s\, [\emptyset(c) \cdot \emptyset(i) \cdot R(s) \cdot (s \subseteq p)]$$

$$\mathtt{0}(c, i, p) \stackrel{\text{def}}{=} (p = c)$$

6.5 Experimental Results

We have implemented an implicit algorithm for exact state minimization of PNDFSM's in a program called ISM2, a sequel to ISM. Prime compatibles and the binate table are generated according to the algorithm described above; then a minimum cover of the table is found by our implicit binate covering solver presented in Chapter 5. We perform and report experiments on the complete set of examples obtained by Watanabe in [83]. Each PNDFSM is an E-machine derived from an arbitrary connection of two completely specified deterministic FSM's, M_1 and M_2, from the MCNC benchmark. The product machine $M = M_1 \times M_2$ is used as the specification. The E-machine which contains all permissible behaviors at M_1 is derived using the procedure in [85]. Our problem is to find a minimum state machine behaviorally contained in the E-machine.

Watanabe's minimizer, PND_REDUCE, does not compute prime compatibles but finds all compatibles instead. In its exact mode, compatible selection is performed by an explicit binate table solver available in the logic synthesis package SIS. In its heuristic mode, it finds instead a Moore [4] machine in the E-machine, by 'expand' and 'reduce' operations [83] on a closed cover of compatibles. As the heuristic looks only to Moore solutions, it is understandable that it will give a worse solution than the minimum contained machine if the latter is not Moore. Also the run times might not be directly comparable. They are reported in the table for completeness.

Table 6.1 summarizes the results of PNDFSM minimization. For each PNDFSM, we report the number of states in the original PNDFSM, the number of states in a heuristic Moore solution obtained by PND_REDUCE, the number of states in a minimum contained FSM, the size of the binate table for PND_REDUCE and for ISM2, and the overall run time for state minimization for PND_REDUCE (in both heuristic and exact modes) and ISM2. All run times are reported in CPU seconds on a DECstation 5000/260 with 440 Mb of memory. For all experiments, timeout is set at 10000 seconds of CPU time, and spaceout at 440Mb of memory.

Out of the 30 examples, PND_REDUCE in exact mode failed to complete on 8 examples because of timeouts, and failed on 4 examples because of spaceout. It can handle all PNDFSM's with less than 16 states. PND_REDUCE in heuristic mode has a timeout on one example, and spaceout on three. Note that some Moore solutions found heuristically are far from the optimum contained ones, although the heuristic solutions may be close to the minimum Moore solutions. Our program ISM2 can handle more examples than PND_REDUCE and only failed to find an exact solution on 2 examples because of timeouts. In those two cases, ISM2 did succeed in computing the prime compatibles as well as building the binate covering table. Furthermore, for the example pm41, it

[4] State minimization algorithms for finding Moore behaviors will be presented in Chapter 7.

found a solution with 9 states after 4844.7 seconds (whereas PND_REDUCE heuristic took 4011.9 seconds to find a 19-state Moore solution). For pm50, it found a first solution with 4 states in 150.7 seconds, and the minimum one with 3 states in 7309.5 seconds, while optimality was concluded in 49181 seconds after the complete branch-and-bound tree was searched (whereas PND_REDUCE heuristic can only find a Moore solution with 13 states). It is encouraging to note that our implicit exact algorithm has run times at least comparable to the heuristic one in PND_REDUCE, and in some cases, much faster.

Note that many exact minimum solutions had only one state, i.e., the solution is pure combinational logic. Such is a Mealy machine unless the logic is a constant. It is believable that the minimum Moore behavior is much larger in general as indicated possibly by the results of PND_REDUCE.

Note also that each compatible results in a column of the binate table by PND_REDUCE in exact mode whereas ISM2 has one column for each prime compatible. The fact that most examples have very few prime compatibles shows the effectiveness of our prime compatibles computation for PNDFSM minimization. Even in these cases, state minimization may not be trivial because compatible generation and prime dominance may take a long time, e.g., pm04 and s3p1.

	no.	# states in sol.		table size (rows x columns)		CPU time (seconds)		
	of	heur.	exact	PND_REDUCE	ISM2	PND_REDUCE		ISM2
PNDFSM	states	Moore	Mealy	exact	exact	heur.	exact	exact
L3	17	2	2		10 x 4	126.2	timeout	17.4
am9	13	12	1		1 x 1	13.7	timeout	2.3
ax4	11	1	1	26 x 28	1 x 1	2	0.7	0.7
ax7	20	2	2	334 x 308	20 x 6	2.8	15.6	7.6
bx7	23	2	2	254 x 216	20 x 6	3.2	9.9	9.0
damiani	5	5	3	21 x 24	17 x 10	0.1	0.1	1.3
e4at2	14	11	1		1 x 1	5.5	timeout	1.1
e4bp1	11	1	1	1064 x 995	1 x 1	2.4	308.1	0.8
e4t1	6	1	1	103 x 120	1 x 1	1.1	0.7	0.3
e69	8	1	1	551 x 501	1 x 1	0.3	10.5	0.3
e6tm	21	8	1		1 x 1	26.1	timeout	3.1
ex10	13	4	1	23 x 28	1 x 1	0.6	0.5	0.6
ex12	13	1	1	1451 x 1019	1 x 1	1.1	16149.1	0.8
mc9	4	1	1	7 x 11	1 x 1	0.1	0.1	0.1
mt51	16	10	1		1 x 1	8.9	timeout	3.8
mt52	9	4	1	256 x 639	1 x 1	3.7	39.4	0.8
pm03	15	1	1	1203 x 1019	1 x 1	1.2	1751.1	0.8
pm04	79		1		1 x 1	spaceout	spaceout	120.6
pm11	9	1	1	331 x 395	1 x 1	43.5	29.7	1.3
pm12	7	3	1	8 x 19	1 x 1	9.8	0.4	0.4
pm31	22		1		1 x 1	timeout	spaceout	3.6
pm33	21	13	1		1 x 1	3327.3	timeout	6.6
pm41	33	19	$\leq$9		12050 x 4774	4011.9	spaceout	*4844.7
pm50	22	13	3		1249 x 515	32.2	timeout	49181
s3p1	38		1		1 x 2	spaceout	spaceout	915.8
s3t2	36		1		389 x 18	spaceout	timeout	39.9
tm01	10	1	1	476 x 767	1 x 1	1.2	40.9	0.8
tm02	7	1	1	155 x 211	1 x 1	0.4	3.1	0.4
tm31	9	1	1	125 x 113	1 x 1	0.3	1.1	0.3
tm32	9	3	2	106 x 143	37 x 9	0.4	1.2	2.9

* best solution before timeout

Table 6.1: State minimization of PNDFSM's.

Chapter 7

State Minimization of PNDFSM's for FSM Networks

7.1 NDFSM's in Logic Synthesis

NDFSM's are useful in sequential synthesis as a formalism to capture a set of sequential functions. Of all such sequential functions, those that satisfy given criteria of correctness are valid candidates for implementation (design verification stage). Of all valid candidates, one maximizing a given optimization criterion is implemented (synthesis stage). Recent papers [86, 85, 20] have proposed specific applications of NDFSM's, especially PNDFSM's, to sequential synthesis. Some examples are presented in [20].

NDFSM's arise in sequential synthesis when capturing all the flexibility that is in a node of a network of FSM's. We will summarize the related theory, which was first proposed by Watanabe *et al.* in [86, 85]. We review in the sequel these applications that explore permissible behaviors in networks of FSM's. We detail all those parts of the theory that are important for the step of state minimization (i.e., to select an optimal behavior of all permissible ones). We present also a more comprehensive analysis of the conditions of the existence of a related feedback composition.

Given a synchronous system of interacting FSM's and a specification, consider the problem of finding the complete set of permissible behaviors at a particular component of the system [1]. The problem is illustrated in Figure 7.1, where M_1 is the FSM associated with the component to be optimized, M_2 represents the behavior of the rest of the system, and M gives the specification.

[1]This extends to the sequential case the problem of finding the maximum set of permissible Boolean functions at a node of a combinational network.

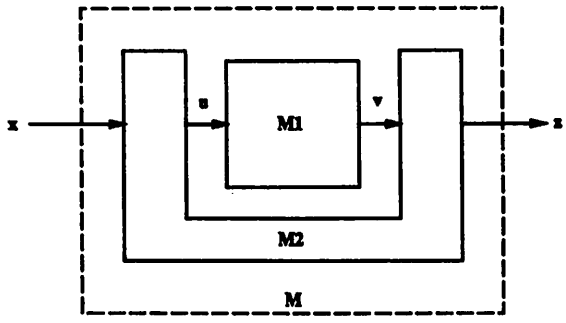

Figure 7.1: Interaction between two machines.

A behavior, as defined in Section 2.4, is a set of input/output strings that can be produced by a DFSM. More than one DFSM may produce the same behavior. We say that such a DFSM represents that behavior. An NDFSM will represent in general more than one behavior. When not confusing, we will refer interchangeably to a behavior and a DFSM that represents it. Loosely speaking, a behavior is permissible if the resulting entire system meets the specification, in the hypothesis that the other components are fixed. In other words, a behavior is permissible at a node of an interconnection, say at M_1, if the system obtained by placing at that node a DFSM that represents the behavior works according to the original specification. Such a replacement is allowed if the composition is *well-defined* and the global behavior is the one expected. In the Section 7.1.1, relevant formal definitions will be given. In particular, a more restricted notion of permissible behavior proposed in [86, 85] will guarantee that the composition is well-defined, i.e., that potentially hazardous implementations are ruled out.

The maximal set of permissible behaviors can be captured by the E-machine and its related theory in [83] will be reviewed in Section 7.1.2. If the set of behaviors permissible at M_1 is found, a 'best' one can then be chosen. Its optimality is usually evaluated by the estimated compactness of a final implementation; for instance, one might choose a behavior with a minimum number of states. Different minimization problems will be described in Section 7.2. Implicit algorithms will be proposed for these problems in Sections 7.3, 7.4 and 7.5.

Similar formulations of this problem have been called *model matching* of FSM's in the context of the *supervisory control problem* for discrete event processes [62, 4, 5].

Given an FSM M, called the *reference model*, that describes some desired input-output behavior and another FSM M_1, called the *plant*, with a given behavior, the problem of *model matching* is to find a *compensator* FSM M_2 that modifies the behavior of M_1 in such a way that the

controlled system, that is the composition of M_1 and M_2, matches the desired behavior. If both the inputs and the outputs of M_1 are allowed to be modified by M_2 as shown in Figure 7.1, the problem is termed *generalized strong model matching* when a unique reset state for each of M, M_1 and M_2 is specified, and *generalized asymptotic model matching* when the reset states of M_1 and M are arbitrary. If M_2 is not allowed to modify the outputs of M_1, then the above problems are termed *strong model matching* and *asymptotic model matching* respectively.

Figure 7.2 shows an example of model matching problem of the latter type, since M_2 is not allowed to modify the outputs of M_1. Notice that the external inputs X drive directly the flexible component (M_2). This simplifies the computation of the permissible behaviors.

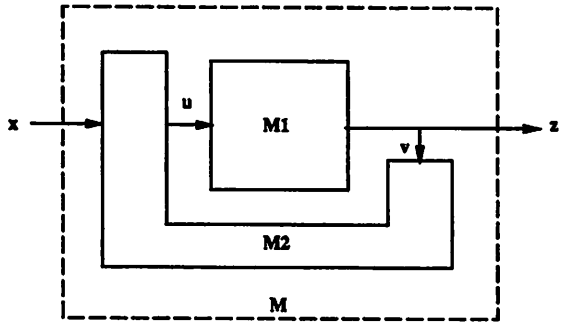

Figure 7.2: Supervisory control problem.

Finally a related problem is the *division* problem for FSM's: given an FSM M and a *divisor* M_1, find the *quotient* Q of the two as shown in Figure 7.3. This is the core problem in the context of factorization and decomposition of FSM's.

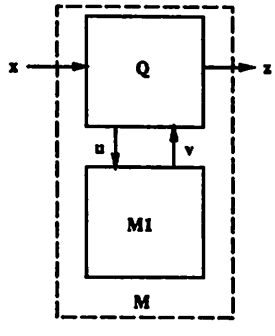

Figure 7.3: FSM Boolean division.

7.1.1 Well-defined Composition of DFSM's

Unless otherwise specified, from now on please refer to the network in Figure 7.1.

Suppose $M_1 = (U, V, S_1, \lambda_1, \delta_1, r_1)$, and $M_2 = (X \cup V, U \cup Z, S_2, \lambda_2, \delta_2, r_2)$ are given as DFSM's. M_1 takes input u and outputs v, and M_2 takes input x and v and outputs u and z. The output function of M_2, λ_2, is often splitted in $\lambda_2^u : S_2 \times B^{|X \cup V|} \rightarrow B^{|U|}$ and $\lambda_2^z : S_2 \times B^{|X \cup V|} \rightarrow B^{|Z|}$ such that $\lambda_2(s_2, xv) = (\lambda_2^u(s_2, xv), \lambda_2^z(s_2, xv))$. It is not always possible to connect M_1 and M_2 according to the topology shown in Figure 7.1 in such a way that the connected machines together represent a deterministic (or output deterministic) FSM with respect to input x and output z. The reason is that the present value of v is a function of the present value of v, as shown by

$$v = \lambda_1(s_1, u) = \lambda_1(s_1, \lambda_2^u(s_2, xv));$$

and, the present value of u is a function of the present value of u, as shown by:

$$u = \lambda_2^u(s_2, xv) = \lambda_2^u(s_2, x\lambda_1(s_1, u)).$$

It is not true a priori that there is a solution of the first equation in v for every choice of s_1, s_2, x. If there is unique solution, one can write explicitly v as a function of the other variables: $v = f(s_1, s_2, x)$ [2]; similarly, for the the second equation. So we ask the question: when is the connection, called product-machine, well-defined?

If M_1 or M_2 can be guaranteed to be Moore machines [3], there is a unique (u, v) pair for each (s_1, s_2, x) combination. A sufficient condition for well-definedness, called implementability, has been proposed in [83]. We shall review it later. Here we introduce a more general condition under which a product is well-defined.

Well-defined product DFSM

Definition 7.1 *Given DFSM* $M_1 = (U, V, S_1, \lambda_1, \delta_1, r_1)$, *and DFSM* $M_2 = (X \cup V, U \cup Z, S_2, \lambda_2, \delta_2, r_2)$, *the* **reachable product DFSM** *of* M_1 *and* M_2, *denoted by* $M_1 \times M_2$, *is the DFSM* $(X, Z, S_p, \lambda_p, \delta_p, r_p)$ *where* $r_p \stackrel{\text{def}}{=} (r_1, r_2)$, S_p *is defined as the set of reachable states* [4] *in* $S_1 \times S_2$, *and there are functions* $\lambda_p : S_p \times X \rightarrow Z$ *and* $\delta_p : S_p \times X \rightarrow S_p$ *such that for each state pair* $(s_1, s_2) \in S_p$ *and an* $x \in X$, *there exists* u *and* v *such that* $\lambda_p((s_1, s_2), x) \stackrel{\text{def}}{=} \lambda_2^z(s_2, xv)$ *and* $\delta_p((s_1, s_2), x) \stackrel{\text{def}}{=} (\delta_1(s_1, u), \delta_2(s_2, xv))$.

[2] There is an analogy here with the implicit function theorem in real analysis.

[3] As observed in [2], we only need M_2 to be Moore in v, but it can be Mealy in x.

[4] A detailed constructive definition of S_p will be given in Theorem 7.3.

Note that the reachable product DFSM is *well-defined* only if λ_p and δ_p are well-defined functions. Functions λ_p and δ_p might not exist because of the dependence on u and v.

The above definition was conceived independently from a similar notion of well-defined composition presented earlier in [4, 5], when characterizing by means of a pre-post dynamic state compensator all solutions to the model matching problem in a closed plant-compensator configuration. Analysis of combinational cycles in circuits has been reported in [51, 12, 75].

Example Consider $\lambda_p((s_1, s_2), x) = z$. We have that $z = \lambda_2^z(s_2, xv)$ depends on v, which in turn is given by $v = \lambda_1(s_1, u)$, where $u = \lambda_2^u(s_2, xv)$. Given (s_1, s_2) and x, it is not guaranteed that there are u and v such that the last two equations are satisfied. If they exist, v can be plugged into the first equation to generate z. A similar analysis holds for δ_p.

Theorem 7.1 *Given DFSM's M_1 and M_2 with their corresponding state transitions represented by $T_1(u, s_1, s_1', v)$ and $T_2(xv, s_2, s_2', uz)$, the product machine $M_1 \times M_2$ has state transitions represented by $T_p(x, (s_1, s_2), (s_1', s_2'), z) = \exists u, v\, [T_1(u, s_1, s_1', v) \cdot T_2(xv, s_2, s_2', uz)]$. The product machine $M_1 \times M_2$ is a* **well-defined DFSM** *if and only if for each reachable product state pair $(s_1, s_2) \in S_p$ and for each input x, there exists a unique next state pair $(s_1', s_2') \in S_p$ and a unique output z such that $T_p(x, (s_1, s_2), (s_1', s_2'), z) = 1$.*

To test the above condition implicitly, we check that the following predicate is true (for all s_1, s_2 and x):

$$S_p(s_1, s_2) \Rightarrow !s_1', s_2', z\, \exists u, v\, [T_1(u, s_1, s_1', v) \cdot T_2(xv, s_2, s_2', uz)]$$

where ! denotes the unique *operator: $!x\, F(x, y) \stackrel{\text{def}}{=} \{y | y$ is related to a unique x in $F\}$.*

As shown in Section 3.3.2, the *unique* operator can be implemented as a primitive BDD operator. As a result, the above well-definedness test can be performed with only a few BDD operations.

Definition 7.2 *Given DFSM $M_1 = (U, V, S_1, \lambda_1, \delta_1, r_1)$, and DFSM $M_2 = (X \cup V, U \cup Z, S_2, \lambda_2, \delta_2, r_2)$, the* **reachable product ODFSM** *(output-deterministic FSM) of M_1 and M_2, denoted by $M_1 \times M_2$, is the NDFSM $(X, Z, S_p, \lambda_p, \Delta_p, r_p)$ where $r_p \stackrel{\text{def}}{=} (r_1, r_2)$, S_p is defined as the set of reachable states in $S_1 \times S_2$, and there is a functions $\lambda_p : S_p \times X \rightarrow Z$ and a relation $\Delta_p : S_p \times X \times S_p \rightarrow B$ such that for each state pair $(s_1, s_2) \in S_p$ and an $x \in X$, there exists u and v such that $\lambda_p((s_1, s_2), x) \stackrel{\text{def}}{=} \lambda_2^z(s_2, xv)$ and $(\delta_1(s_1, u), \delta_2(s_2, xv)) \in \Delta_p((s_1, s_2), x)$.*

Note that the reachable product ODFSM is *well-defined* only if λ_p is a well-defined function. Contrasting to well-definedness of reachable product DFSM, we do not require here that the product next state relation be a function.

Theorem 7.2 *Given DFSM's M_1 and M_2 with their corresponding state transitions represented by $T_1(u, s_1, s_1', v)$ and $T_2(xv, s_2, s_2', uz)$, the product machine $M_1 \times M_2$ has state transitions represented by $T_p(x, (s_1, s_2), (s_1', s_2'), z) = \exists u, v\, [T_1(u, s_1, s_1', v) \cdot T_2(xv, s_2, s_2', uz)]$. The product machine $M_1 \times M_2$ is a* **well-defined ODFSM** *if and only if for each reachable product state pair $(s_1, s_2) \in S_p$ and for each input x, there exists one or more next state pairs $(s_1', s_2') \in S_p$ and there exists a unique output z such that $T_p(x, (s_1, s_2), (s_1', s_2'), z) = 1$.*

To test the above condition implicitly, we check that the following predicate is true:

$$S_p(s_1, s_2) \Rightarrow !z\, \exists s_1', s_2', u, v\, [T_1(u, s_1, s_1', v) \cdot T_2(xv, s_2, s_2', uz)]$$

The only difference of the above predicate than the one in Theorem 7.1 is here we do not require s_1 and s_2 to be unique.

Theorem 7.3 *Given DFSM M_1 and M_2, the set S_p of* **product reachable states** *in $S_1 \times S_2$ can be computed by the following fixed point computation:*

$$\begin{aligned} S_p^0(s_1, s_2) &= \{(r_1, r_2)\} \\ S_p^{k+1}(s_1, s_2) &= S_p^k(s_1, s_2) + [(s_1', s_2') \rightarrow (s_1, s_2)] \\ &\quad \exists u, v, x, z, s_1, s_2\, [S_p^k(s_1, s_2) \cdot T_1(u, s_1, s_1', v) \cdot T_2(xv, s_2, s_2', uz)] \end{aligned}$$

Implementable FSM's

In [83], a definition of product machine is given, based on the existence of a unique pair u, v such that the equations are satisfied. Moreover, a sufficient condition, called implementability, for the existence of such a unique pair u, v is introduced.

Definition 7.3 *Given DFSM $M_1 = (U, V, S_1, \lambda_1, \delta_1, r_1)$, and DFSM $M_2 = (X \cup V, U \cup Z, S_2, \lambda_2, \delta_2, r_2)$, the* **product DFSM** *of M_1 and M_2, denoted by $M_1 \times M_2$, is the DFSM $(X, Z, S_1 \times S_2, \lambda_p, \delta_p, r_p)$ where $r_p \stackrel{\text{def}}{=} (r_1, r_2)$, and there are functions $\lambda_p : (S_1 \times S_2) \times X \rightarrow Z$ and $\delta_p : (S_1 \times S_2) \times X \rightarrow (S_1 \times S_2)$ such that for each state $(s_1, s_2) \in S_1 \times S_2$ and an $x \in X$, there is a unique pair (u, v) where $v = \lambda_1(s_1, u)$ and $u = \lambda_2^u(s_2, xv)$, such that $\lambda_p((s_1, s_2), x) \stackrel{\text{def}}{=} \lambda_2^z(s_2, xv)$ and $\delta_p((s_1, s_2), x) \stackrel{\text{def}}{=} (\delta_1(s_1, u), \delta_2(s_2, xv))$.*

Definition 7.3 is more restrictive than Definition 7.1, because the uniqueness of minterms u, v is a sufficient, but not necessary condition to obtain a well-defined feedback composition of

M_1 and M_2. Moreover, in Definition 7.3 it is requested that the sufficient condition holds also for unreachable states.

The following example shows a feedback composition that is well-defined according to Definition 7.1, but does not satisfy Definition 7.3.

Example Consider the DFSM M_1 that has a unique state s_1, an input u and an output v. Let the output function at s_1 be $\lambda_1(s_1,0) = 0$, $\lambda_1(s_1,1) = 1$. Consider the DFSM M_2 that has a unique state s_2, inputs x, v and outputs u, z. Let the output function at s_2 be $\lambda_2(s_2,00) = 00$, $\lambda_2(s_2,01) = 10$, $\lambda_2(s_2,10) = 01$, $\lambda_2(s_2,11) = 11$. The product machine is well-defined as the DFSM $M_1 \times M_2$ that has a unique state (s_1, s_2), an input x and an output z. The output function at (s_1, s_2) is $\lambda_p((s_1, s_2),0) = 0$, $\lambda_p((s_1, s_2),1) = 1$. But there are two pairs of u, v such that the equations are satisfied, i.e., $u = 0, v = 0$ and $u = 1, v = 1$.

In [86, 85] a sufficient condition called implementability is imposed on M_1, to ensure that for each present state pair and each input, there is a unique (u, v) pair.

Definition 7.4 *Given $M_2 = (X \cup V, U \cup Z, S_2, \lambda_2, \delta_2, r_2)$, a DFSM $M_1 = (U, V, S_1, \lambda_1, \delta_1, r_1)$ is* **implementable** *with M_2 if there exists a pair of circuit implementations of M_1 and M_2 such that no combinational loop is created by connecting them together at u and v.*

A combinational loop is a cycle of gates and wires, with no latch or flip-flop to break the cycle.

Theorem 7.4 *For an implementable machine M_1, and for an arbitrary sequence of $B^{|X|}$, say $\sigma = (x_0, \ldots, x_t)$, 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 $(x_0, \ldots, x_{t-1})$, then x_t defines a unique pair $(u, v) \in B^{|U|} \times B^{|V|}$ such that $u = \lambda_2^{(u)}(s_2, x_t v)$ and $v = \lambda_1(s_1, u)$.*

Proof: Suppose that no combinational loop exists between the variables of U and V. Then either the variables of V do not depend on those of U or vice versa. Suppose w.l.o.g the former case. Then for a state s_1 of M_1 a unique value of v, say $\overline{v}$, is determined. In M_2, given s_2, x and $\overline{v}$, a unique value of u, say $\overline{u}$, is determined. Therefore for each state x, (s_1, s_2), a unique pair $\overline{u}, \overline{v}$ is determined such that $\overline{v} = \lambda_1(s_1, \overline{u})$ and $\overline{u} = \lambda_2^{(u)}(s_2, x\overline{v})$. ■

The following example shows a feedback composition that is well-defined according to Definition 7.1, but does not satisfy Definition 7.4.

Example Consider the DFSM M_1 that has a unique state s_1, inputs u_1, u_2 and outputs v_1, v_2. Let the output function at s_1 be $\lambda_1(s_1,00) = 00$, $\lambda_1(s_1,01) = 10$, $\lambda_1(s_1,10) = 01$, $\lambda_1(s_1,11) = 10$.

Consider the DFSM M_2 that has a unique state s_2, inputs x, v_1, v_2 and outputs u_1, u_2, z. Let the output function at s_2 be $\lambda_2(s_2, -00) = 100$, $\lambda_2(s_2, -01) = 001$, $\lambda_2(s_2, -10) = 011$, $\lambda_2(s_2, -11) = 010$. The product machine is well-defined as the DFSM $M_1 \times M_2$ that has a unique state (s_1, s_2), an input x and an output z. The output function at (s_1, s_2) is $\lambda_p((s_1, s_2), -) = 1$, since there is a unique pair u_1, u_2, v_1, v_2 such that the equations are satisfied, i.e., $u_1 = 0, u_2 = 1, v_1 = 1, v_2 = 0$. M_1 is not implementable because there are combinational loops (for instance between u_1 and v_2).

Note that if M_2 is a **Moore machine** or M_1 is restricted to be Moore, then the implementability of M_1 with M_2 is guaranteed, because the existence of a unique u, v is guaranteed.

A necessary and sufficient condition for M_1 to be implementable is given in [86, 85]. 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:

$$[u_i, v_j] \in E \Leftrightarrow \lambda_1^{v_j} \text{ depends on } u_i,$$
$$[v_j, u_i] \in E \Leftrightarrow \lambda_2^{u_i} \text{ depends on } v_j,$$

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*. Then the following theorem holds.

Theorem 7.5 *M_1 is implementable if and only if G is acyclic.*

In the previous theory there is an hidden hypothesis that u and v are binary variables. If u and v are symbolic variables, cyclic dependency is not an invariant of the encoding process, therefore an appropriate encoding in an actual implementation could break cyclic dependency found at the symbolic level. This statement implies that Watanabe's criterion would be even more conservative when u and v are symbolic variables.

7.1.2 Permissible Behaviors at a Component FSM

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. M can be a non-deterministic machine (to specify a set of behaviors, rather than a single behavior, for the entire system), while M_2 is a completely specified deterministic machine. However, Watanabe only considered M that is completely specified and deterministic in his implementation.

A synthesis problem is to find the complete set of behaviors, such that the product of each DFSM M_1, representing a behavior in the collection, with the fixed DFSM M_2 (i.e., $M_1 \times M_2$) is well-defined and represents a behavior contained in the NDFSM M. If M is a DFSM, the behavior

of $M_1 \times M_2$ must be equal to the behavior of M. The property of M_1 that the behavior of $M_1 \times M_2$ is one of the behaviors of M (i.e., is contained in M) is called containment property of M_1.

A DFSM that satisfies both implementability and containment is a valid choice for M_1 and is taken as the definition of permissible DFSM in [86, 85].

Definition 7.5 *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 DFSM $M_1 = (U, V, S_1, \lambda_1, \delta_1, r_1)$ is* **permissible** *if M_1 is implementable and the behavior of $M_1 \times M_2$ is contained in M.*

We have defined a permissible machine replacing implementability with the less restrictive condition that the (reachable) product machine of M_1 and M_2 is well-defined.

Definition 7.6 *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 DFSM $M_1 = (U, V, S_1, \lambda_1, \delta_1, r_1)$ is* **permissible** *if the (reachable) product machine of M_1 and M_2 is well-defined and the behavior of $M_1 \times M_2$ is contained in M.*

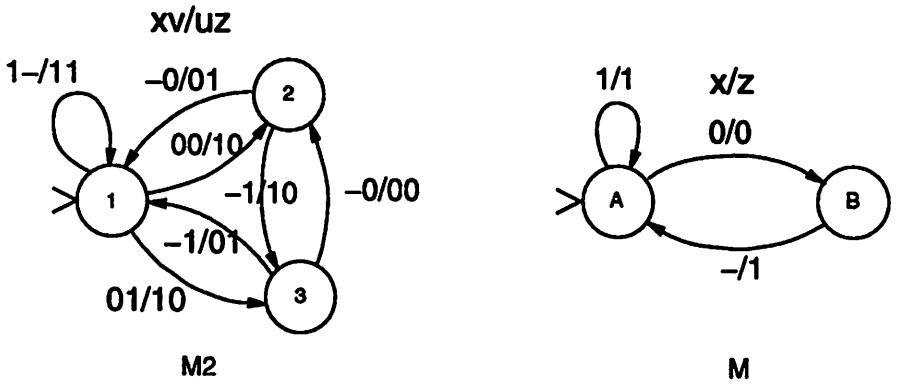

Figure 7.4: Watanabe's example of M_2 and M.

Given M and M_2 (e.g., consider Figure 7.4 from [83]), many permissible behaviors can be implemented at M_1. The problem is how to compute all of them and how to represent them in the most compact way. Notice that a DFSM represents (i.e., contains) a unique behavior, while the same behavior may be represented (i.e., contained) by many DFSM's, potentially an infinite number of them. Therefore our objective is to find the complete set of permissible behaviors at M_1, and not the complete set of permissible machines at M_1.

We will see that one can represent them with an NDFSM computed by a fixed point computation. By determinizing the underlying NDFA (applying a subset construction algorithm),

the NDFSM can be coverted into a PNDFSM. It is also possible to define a fixed point computation that builds directly the PNDFSM. The latter computation has been proposed in [85] and the PNDFSM so obtained has been called the *E-machine* [5]. Similarly, the result of the former computation has been called the *NDE-machine.*

Since determinization could lead to an exponential increase of the state space, the first approach seems more attractive. On the other hand, in the case of PNDFSM's it has been possible to extend traditional algorithms to find a minimum-state contained behavior, while such a direct extension seems to fail for general NDFSM's. It is a matter of current research to compare more in depth the two approaches and explore algorithms that work directly on NDFSM's. We will review next both types of computations.

Instead of capturing the permissible behaviors with an NDFSM and then determinizing it to obtain a PNDFSM, one can perform the two steps together by means of a unique fixed point computation, that is a modification of the one needed to compute the NDFSM. In this section M is not restricted to be a DFSM as in the previous section, but it can be an NDFSM.

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

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

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

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

If the antecedent of the implication (2) is false, then the implication is true and $\Sigma_n = \emptyset$. If the antecedent of the implication (2) is true, and condition (2)(a) of the consequent is false, then the implication is false and so the transition is not added to the relation. Condition (2)(a) rules out

[5] the prefix *E* stands for environment.

transitions such that M_1 would violate the condition that $M_1 \times M_2$ is contained in M. Condition (2)(b) instead builds the set Σ_n. Condition (3) is necessary to restrict Σ_n to those sets of pairs that have their predecessor in Σ_p.

By condition (2), for each pair $(\tilde{s}_2, \tilde{s^*}) \in \Sigma_p$, there might exist more than one $x \in B^{|X|}$ such that $u = \lambda_2^{(u)}(\tilde{s}_2, xv)$.

Since the pair of next states may be different for different x, more than one pair $(\delta_2(\tilde{s}_2, xv), \Delta(\tilde{s^*}, x, \lambda_2^{(x)}(\tilde{s}_2, xv)))$ is added to the next state Σ_n, for each $(\tilde{s}_2, \tilde{s^*})$ and (u, v).

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 exists $\tilde{\Sigma}_p \in \mathcal{S}^{(t)}$, $u \in B^{|U|}$, and $v \in B^{|V|}$ such that $T^{(t+1)}(\tilde{\Sigma}_p, u, v, \Sigma_p) = 1$.

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 $\mathcal{S} = \mathcal{S}^{(K)} \cup \{\phi\}$.

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

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

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

The resulting PNDFSM is defined as a 5-tuple $T = (U, V, \mathcal{S}, T, \Sigma_r)$, where $\Sigma_r = \{(r_2, \{r\})\}$. Each state represents a subset of $S_2 \times 2^S$. If M is a DFSM, a state is a subset of $S_2 \times S$.

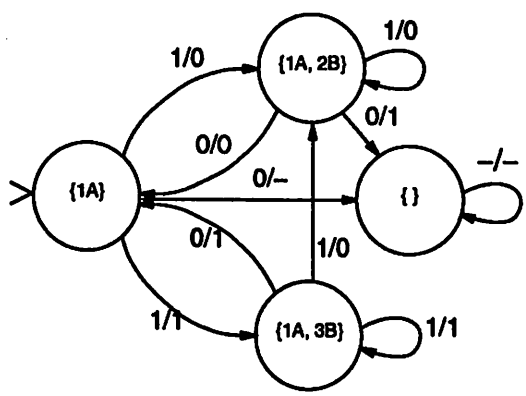

Figure 7.5: E-machine for M_2 and M of previous example.

If the global inputs X drive directly M_1, then each state of the PNDFSM corresponds to a single pair $(s_2, s^*) \in S_2 \times 2^S$. A slightly modified version of the previous fixed point computation

builds the transition relation $T^{(t+1)}(\Sigma_p, xu, v, \Sigma_n) = 1$. In this case it is expected that the PNDFSM has a smaller number of states than when the global inputs X are not available directly to M_1.

It has been proved formally in [83] that a behavior implementable at M_1 is permissible if and only if the behavior is contained in the PNDFSM obtained by the fixed point computation presented in this section.

It is obvious by construction that a behavior implementable at M_1 contained in the PNDFSM is permissible. It is less straightforward to show the converse, i.e., that for each permissible machine M_1, there exists an equivalent machine contained in the E-machine. The latter is proved by using a previous result stating that for each implementable machine M_1 there is an equivalent prime machine. Then the bulk of the proof is to show that such prime machine is contained in the PNDFSM built by the fixed point computation.

Since the PNDFSM is obtained by a constructive procedure, one can conclude that if a DFSM is implementable at M_1, then it is permissible if and only if there is an equivalent DFSM contained in the PNDFSM obtained by the fixed point computation.

Note that the previous results hold also if one replaces implementability with the condition that the product machine of M_1 and M_2 exists.

One could raise the question whether, in the special case of the E-machine computation, it is possible at all that there are contained behaviors M_1 such that the product of M_1 and M_2 is not well-defined (or M_1 is not implementable). The answer if yes, as the following case of a non-implementable behavior contained in a E-machine shows.

Example Consider the DFSM M that has a unique state s, an input x and an output z. Let the output function at s be $\lambda_p(s,0) = 0$, $\lambda_p(s,1) = 1$. Consider the DFSM M_2 that has a unique state s_2, inputs x, v and outputs u, z. Let the output function at s_2 be $\lambda_2(s_2,00) = 00$, $\lambda_2(s_2,01) = 11$, $\lambda_2(s_2,10) = 00$, $\lambda_2(s_2,11) = 11$. Consider the behavior contained in the E-machine represented by the DFSM M_1 that has a unique state s_1, an input u and an output v. The output function at s_1 is $\lambda_1(s_1,0) = 1, \lambda_1(s_1,1) = 0$. M_1 is not implementable and $M_1 \times M_2$ is not well-defined either. The reason is that the output z cannot be uniquely defined for any input value x, because z is oscillating (there is no pair u, v such that the equations are satisfied [6]). Notice that the counterexample is built using the don't cares transitions defined by the implication (2) when the antecedent is false.

[6] One could show a variant of this example where z cannot be uniquely defined, because there are two pairs of stable points u, v defining different z's.

7.2 State Minimization Problems for PNDFSM's

As a summary to the synthesis problem in Figure 7.1, a behavior is permissible at M_1 if and only if

1. M_1 is behaviorally contained in the E-machine, and
2. the product machine $M_1 \times M_2$ is well-defined.

Given DFSM's M_2 and M, Watanabe [85] provided an algorithm and a program to generate the E-machine, which is a PNDFSM. Assuming that we have an E-machine, we address in the remaining sections in this chapter the following problems of selecting one behavior for implementation:

- **P1**: Find a minimum behavior that is contained in the E-machine
- **P2**: Find a Moore behavior that is contained in the E-machine
- **P3**: Find a minimum Moore behavior that is contained in the E-machine
- **P4**: Find a well-defined behavior that is contained in the E-machine
- **P5**: Find a minimum well-defined behavior that is contained in the E-machine

In Figure 7.1, if M_2 is a Moore machine [7], we can choose any behavior at M_1, e.g., one with the minimum number of states, from the E-machine. $M_1 \times M_2$ is guaranteed to be well-defined because of the existence of unique (u, v) pairs. As a result, problem **P1** can be solved exactly by our implicit state minimization algorithm for PNDFSM's as described in Section 6.3.

Note that problems **P2** and **P3** of selecting Moore behavior are easier to solve than problems **P4** and **P5** of selecting well-defined behavior. One reason is that when selecting a well-defined M_1, we must check against the given M_2, whereas for solving **P2** and **P3**, we can ignore M_2. But algorithms for **P4** and **P5** can give better solutions than those for **P2** and **P3** respectively. We are not going to consider here the problem of finding an implementable behavior out of the E-machine because it cannot give a better solution than **P4** or **P5**, and it is not obvious that it can lead to a simpler algorithm.

As compared with **P1**, the problems **P2** to **P5**, which we shall address, have the extra constraint that M_1 should be either Moore or well-defined when connected to M_2. As we are still exploring behaviors contained in a PNDFSM (i.e., E-machine), the theory on closed covers leading

[7] Again we only need M_2 to be Moore in v.

up to Theorem 2.20 is still valid for these problems. But Theorem 2.17 cannot be applied because we cannot restrict our attention only to minimum closed covers. Here we are interested in all closed covers that satisfy the Moore or well-definedness property. One consequence is that compatible dominance cannot be used for prime compatible generation, nor table column dominance. Another consequence is that a selected compatible may be mapped into more than one state in an optimum solution.

There are two approaches that we can use to handle this added constraint. All algorithms first generate the set of compatibles using the implicit method as described in Section 6.4.1 and subsequently solve a binate covering problem. They differ on how the table is set up, and also if the classical branch and bound algorithm needs to be modified.

One approach proposed in [83] to solve **P3** and **P5** is by expanding on the number of table columns so that the problems can be cast exactly as a binate covering problem. For problem **P3**, each column corresponds to a pair (c, v) where c is a compatible and v is an output minterm. For problem **P5**, each column corresponds to a pair (c, f) where c is a compatible and f is an output function. Using these pairs as columns, the Moore and well-definedness conditions can be expressed as binate clauses. Along with the binate closure clauses and the unate covering clauses, a classical branch and bound algorithm for binate covering can explore solutions that satisfy all conditions simultaneously. Such an implicit algorithm for **P3** is presented in Section 7.3. The disadvantage of this approach is that the width of table is larger than necessary. We will show in any minimum solution to **P3**, no two selected columns (corresponding to states in a minimized machine) will have the same compatible c associated to two different output v parts. For problem **P5**, it is highly unlikely that a practical (explicit or implicit) algorithm can represent and manipulate tables whose columns are labeled by (c, f) pairs because there is a huge number of such pairs and it is difficult to enumerate the functions f.

Alternatively, we propose to use as many columns as compatibles for **P2** and **P3**. And we use at most $|S_2|$ copies of each compatible as columns for solving problems **P4** and **P5**. Furthermore, we shall argue that only a subset of these columns which satisfied the Moore or well-definedness conditions, called the **pruned set**, needs to be considered for a solution. To find a solution in **P2** or **P4**, we do not need to solve a binate table, but can always find a solution by a single traversal of the transition graph of the power machine M^{power} as defined in Section 2.8. To find a minimum solution for **P3** or **P5**, we first form a binate table that represents the covering and closure conditions only. Once an intermediate solution (a closed cover) to this binate table is found, we check further to see if it also satisfies the Moore or well-definedness conditions. If so, we have found a true

solution which may still be non-optimum. Then we can continue our branch and bound procedure to search for the globally minimum solution.

As a summary of our second approach, the following outlines a typical algorithm:

- First generate a pruned subset of compatibles
- For **P2** or **P4**, to find a Moore/well-defined contained behavior:
  1. if no pruned compatible covers a reset state, no contained behaviors is Moore
  2. else any closure under T^{power} from a pruned compatible containing a reset state is a solution
- For **P3** or **P5**, to find a minimum-state Moore/well-defined contained behavior:
  1. form a binate table from covering and closure conditions
  2. apply table reductions but not the column dominance nor collapse columns rules
  3. if table reduced to a cyclic core, select a column, and branch to the two subproblems recursively
  4. else table reduced to nothing (no cyclic core)
     (a) check if at least one reduced machine represented by the selected closed cover is Moore/well-defined with respect to M_2
     (b) if so, a solution has been found, set upper bound; continue another branch
     (c) else, no solution has been found; continue another branch

Sections 7.4 and 7.5 will describe in detail this approach of finding Moore and well-defined behavior. In each section, a subsection will be devoted to each of these three subtasks.

Note that the above procedure for **P3** and **P5** is less efficient than the classical branch and bound algorithm for **P1** because (1) we cannot perform dominance between compatibles (therefore we cannot compute prime compatibles nor column dominance), (2) it may take longer to find a real solution as some solutions to the binate table do not satisfy the required behavior, (3) the lower bound is still valid but is relatively weaker. But this method should be more practical than the ones proposed by Watanabe because we have much smaller tables to be covered.

7.3 State Minimization of PNDFSM's for Moore Behavior: 1st Approach

Watanabe defines each compatible to be a pair (q, v) where q represents a set of compatible states, and v an output minterm. The state minimization algorithm presented in this section is joint work with Watanabe. Whenever a compatible is selected, we have chosen a set of mergeable states as well as an output minterm associated with their outgoing transitions. This more complex notion of compatibles leads to simpler binate covering. In the next section, we shall show that the compatible can be defined without associating an output minterm with it.

7.3.1 Implicit Compatible Generation

Balarin *et al.* in [2] observed the following fact that improves the efficiency of compatible generation - a state that cannot produce a same output for all inputs is not involved in any compatible in any reduced Moore machine. As a result, we can simply delete such states from the original machine along with all transitions leading to such pruned states, before we generate compatibles.

The set of compatibles, i.e., (q, v) pairs, is computed using the following fixed point computation:

$$\begin{aligned} C_0(q, v) &= 1 \\ C_{k+1}(q, v) &= \forall u \, \exists q' \, [(\exists v' \, C_k(q', v')) \cdot T^{det}(u, q, q', v)] \end{aligned}$$

where T^{det} is the transition relation over subsets of states, as computed in Section 6.2.1.

First we assume that every pair (q, v), where q is any state set and v is any minterm, is a candidate compatible and so it is in C_0. After the first iteration, $C_1(q, v)$ captures all the state sets q that are output compatibles; each state which can produce the same output minterm v for all inputs. During the k-th iteration, (q, v) will be in C_{k+1} if and only if on output v, for every input u, there exists a next state set $q' \in C_k$ from set q. The iteration can terminate when $C_{k+1} = C_k$ and then the set of compatibles is given by $C(q, v) = C_k(q, v)$.

7.3.2 Implicit Covering Table Generation

We want to construct the set of column labels Col and the set of row labels Row in a format usable by our specialized binate solver as presented in Section 5.7. Using our terminology in Chapter 5, columns are labeled by variables p and rows by pair of variables (c, d). The numbers of Boolean variables used for p, c and d are the same. At the intersection of row (c, d) and column

p, the table entry is a 1 if and only if $p \supseteq d$, and the table entry is a 0 if and only if $p = c$. As any entry cannot be both 0 and 1 simultaneously, we disallow the case $c \supseteq d$.

Each column p in our table is a compatible (i.e., a pair of (q, v)).

$$Col(qv) = C(q, v) \cdot \neg\emptyset(q)$$

Each row label consists of two parts c and d. To match the width of p, the row label is defined so that $c = (q, v)$ and $d = (r, w)$ where q and r represent positional-sets of states, and v and w represent output minterms. None of the clauses involves the w variables so they are always assigned the empty set $\emptyset(w)$. Row_{unate} represents the unate rows corresponding to the covering conditions of the reset states. With the following definition, we have a 1-entry if and only if the state set part q in the column label (q, v) contains a reset state r, i.e. $q \supseteq r$ and $reset\_states(r) = 1$.

$$Row_{unate}(qv, rw) = \emptyset(q) \cdot \emptyset(v) \cdot reset\_state(r) \cdot \emptyset(w)$$

Row_{binate} represents the binate rows corresponding to the closure conditions. In particular, (q, v) must be a compatible ($C(q, v) = 1$), and w must be a state set in a compatible ($\exists w\ C(r, w)$), and they are in the $Row_{binate}(qv, rw)$ relation only if r is the set of next states of q under output minterm v ($\exists u\ [T^{det}(u, q, r, v)]$).

$$Row_{binate}(qv, rw) = [\exists u\ T^{det}(u, q, r, v)] \cdot C(q, v) \cdot [\exists w\ C(r, w)] \cdot \emptyset(w)$$

Finally we collect all these clauses as the set of Row labels, and make sure that $c \not\supseteq d$.

$$Row(qv, rw) = [Row_{unate}(qv, rw) + Row_{binate}(qv, rw)] \cdot (qv \not\supseteq rw)$$

After the implicit table is generated, our specialized binate table solver (for ISFSM minimization) is called for implicit exact binate covering. A first solution from the binate solver is a solution to **P2** whereas the minimum solution of binate covering corresponds to an optimum solution to **P3**.

7.4 State Minimization of PNDFSM's for Moore Behavior: 2nd Approach

State minimization algorithms assumes that at least one minimum machine corresponds to a minimum closed cover of compatibles, in the classical sense. This assumption is true for the state minimization of ISFSM's and PNDFSM's. This assumption cannot be true for the state

minimization problem of general non-deterministic FSM, because each state of a reduced machine corresponds instead to a generalized compatible.

The above assumption is still true for state minimization of PNDFSM's for Moore behavior. For the sake of contradiction, assume that there are two distinct states in a reduced machine which corresponds to one compatible state set. And the states cannot be merged because they have different output functions. If we replace one by the other and remove compatibles that become unreachable, the remaining set of compatibles is still a closed cover, and also it is Moore. Furthermore, it corresponds to a solution with a smaller cardinality, thus contradicting the assumption. So if a solution exists for problems **P2** and **P3**, there must be a reduced machine which states corresponds to distinct compatible state sets.

A similar concept was mentioned by Watanabe in the second half of Section 5.4.3 in [83] but he didn't continue to give an algorithm after noting that the problem reduces to a 0-1 integer non-linear programming problem.

7.4.1 Moore Compatible Generation

To explore behaviors that are *behaviorally contained* in a PNDFSM M, Theorem 2.20 shows that we can construct its power machine M^{power} and explore DFSM's which are *structurally contained* in the power machine whose state space is restricted to the compatibles (i.e., $M^{power}|_{compatibles}$). Here we can confine our attention only to the subset of compatibles that can produce Moore behaviors. This pruned set of compatibles is called the M-compatibles.

Definition 7.7 *A set c of states is an* **M-compatible** *if there exists an output v such that for each input u*

1. *each state in c has a transition under input u and output v, and*

2. *from the set c of states, the set c' of next states under u and v is also an M-compatible.*

The above definition of M-compatibles is different from that of the classical compatibles in Theorem 6.3 in that the order of quantification here is $\exists o\ \forall i$ but $\forall i\ \exists o$ in Theorem 6.3. Note that $\exists$ and $\forall$ are not commutable, and the former definition expresses a stronger condition than the latter, which accounts precisely for the Moore condition at c.

Implicit Generation

Theorem 7.6 *The set C of M-compatibles of a PNDFSM can be computed by the following fixed point computation:*

- *Initially assume all subsets of states to be M-compatible:* $C_0(c) = 1$,
- *By Definition 7.7,* $C_{k+1}(c) = \exists v\, \forall u\, \exists c'\, [C_k(c') \cdot T^{det}(u, c, c', v)]$

The iteration can terminate when $C_{j+1} = C_j$ *and the set of M-compatibles is* $C(c) = C_j(c)$.

Initially, all subsets of states are assumed to be M-compatibles, $C_0(c)$. On the k-th iteration, according to Definition 7.7, we consider a set c to be an M-compatible if and only if there is an output v such that for every input u, c can transit to a C_k M-compatible under u and v.

Note that Balarin's method [2] of trimming the state space to Moore states can also be applied here, but it will generate more compatibles than our set of M-compatibles.

7.4.2 Selecting a Moore Contained Behavior

In this subsection, we consider the problem of finding a Moore behavior that is contained in the E-machine while we will find the exact state-minimum Moore solution in the next subsection.

Definition 7.8 *A set K of M-compatibles* **covers** *the reset state(s) if at least one M-compatible in K contains a reset state.*

Definition 7.9 *A set K of M-compatibles is* **closed** *if for each M-compatible* $c \in K$, *for each input u, there exists an output v such that the set of next states from c under u and v is contained in an M-compatible* $c' \in K$. *i.e., if*

$$\forall c\, \{K(c) \Rightarrow \forall u\, \exists v\, \exists c'\, [K(c') \cdot T^{power}(u, c, c', v)]\} = 1$$

where T^{power} *is defined in Section 2.8.*

Definition 7.10 *A set K of M-compatibles is* **Moore closed** [8] *if for each M-compatible* $c \in K$, *there exists an output v such that for each input u, the set of next states from c under u and v is contained in an M-compatible* $c' \in K$. *i.e., if*

$$\forall c\, \{K(c) \Rightarrow \exists v\, \forall u\, \exists c'\, [K(c') \cdot T^{power}(u, c, c', v)]\} = 1$$

[8] A Moore closed set is represented by a Moore DFSM.

Given a set of M-compatibles, the closure condition (Definition 7.9) is strictly weaker than the Moore closure condition (Definition 7.10) (because $\exists v\ \forall u\ F \Rightarrow \forall u\ \exists v\ F$), and thus it is sufficient to test the latter in order to guarantee the former.

Definition 7.11 *A set K of M-compatibles is* **Moore closed cover** *if K covers a reset state and is Moore closed.*

The power machine M^{power} with its state pace restricted to the M-compatibles (pruned set of compatibles) is denoted by $M^{power}|_{M-compatibles}$. The following theorem shows that we can find a Moore behavior by a traversal of $T^{power}|_{M-compatibles}$.

Theorem 7.7 *If no M-compatible contains a reset state, the E-machine does not contain any Moore behavior.*

Otherwise, the states marked by the following algorithm form a Moore closed cover contained in the E-machine:

- *mark an M-compatible that contains a reset state*
- *for each unprocessed marked state*
  - *choose an output minterm v arbitrarily such that for every input v, there is a transition under (u, v) in $T^{power}|_{M-compatibles}$*
  - *mark every state in its corresponding next state set*

Each minimum Moore closed cover can be enumerated by the algorithm by appropriate choices of the initial M-compatible and the output minterms v when processing each marked state.

Proof: If no M-compatible contains the reset state(s), there is no way we can construct a Moore closed cover.

Provided at least one M-compatible contains a reset state, the algorithm will always terminate because it processes each state at most once, and for each state, we have at least one valid choice of v because the state corresponds to an M-compatible. The set of marked states is a cover because the initial M-compatible marked contains a reset state. The set is also Moore closed by construction.

Given any minimum Moore closed cover, the above algorithm can find it with the following execution steps to obtain the corresponding set of marked states. First, mark any M-compatible

that contains a reset state. For each marked state, choose output v such that the corresponding next state set is contained in the given Moore closed cover. By definition, this can always be done. This algorithm will terminate with the minimum Moore closed cover. ■

A greedy heuristic can be applied to the above algorithm to find a minimal Moore closed cover as follows: pick an initial M-compatible with minimum cardinality, and for each unprocessed marked state, choose the output minterm that results in a smallest number of next states that have not been marked so far.

7.4.3 Selecting a Minimum Moore Contained Behavior

In this subsection, we want to find the exact minimum-state Moore behavior that is contained in a given PNDFSM. We propose an implicit state minimization algorithm that first generates the M-compatibles, and then selects an optimum subset of M-compatibles using an augmented binate table covering procedure. The classical closure condition in Definition 7.9 can be expressed as binate clauses, but the Moore-closure condition in Definition 7.10 cannot.

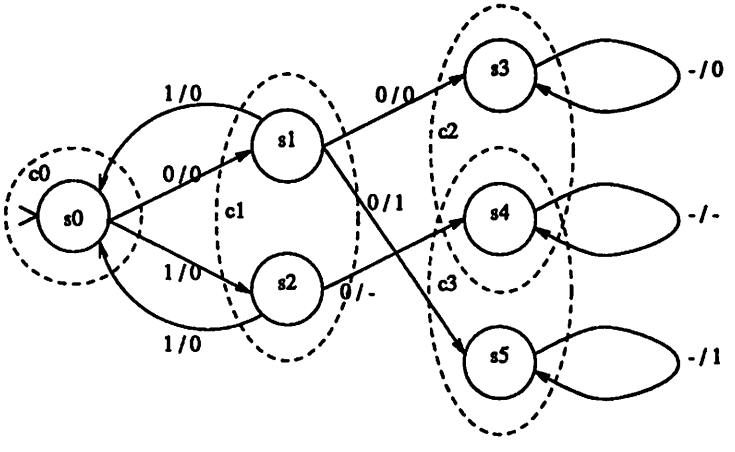

Figure 7.6: A counter example.

Example Figure 7.6 is a counter example, showing that a closed cover of Moore compatibles does not always correspond to a Moore DFSM. There are four M-compatibles: $c0, c1, c2$ and $c3$. The selection $\{c0, c1, c3\}$ is closed and covers the reset state $s0$. But the corresponding reduced machine is not a Moore DFSM, because from $c1$, it outputs 0 on input 1 but outputs 1 on input 0. As a result, we must check Definition 7.10 on each candidate solution returned by a standard branch and bound algorithm for binate covering. Note that $\{c0, c1, c2\}$ is a closed cover of Moore compatibles which corresponds to a Moore DFSM.

Note that the heuristic algorithm presented in the last subsection gives a Moore DFSM because at each step, Moore closure is always maintained at a partial solution.

Implicit Selection of M-Compatibles

The selection problem of a minimum subset of M-compatibles which satisfies the covering and Moore closure condition *cannot* be solved as a binate covering problem, as noted in [83]. This is because the form of each Moore condition requires a disjunction of conjunctions of disjunctions of literals (i.e., M-compatibles) which arises from the quantifications $\exists v\ \forall u\ \exists c'$ in Definition 7.10.

However, the problem can be solved by augmenting the classical branch and bound procedure for binate covering with Moore checks. Once a solution (closed cover in the classical sense) is found, the M-compatible selection is tested to see if it represents a Moore DFSM by the formula in Definition 7.10. The test can be performed implicit using a few BDD operations. If so, we have actually found a Moore solution to our problem. Only then we shall use the Moore-solution as a new upper bound for the continuing search of a better solution.

Note that this augmented branch and bound procedure is correct only if we do not perform column dominance nor column label collapsing (nor prime compatible generation). This is because even if a compatible c' dominates another compatible c in the classical sense, we cannot disregard the compatible c. It is possible that there is a minimum Moore solution containing compatible c, but by replacing c with c', the modified selection ceased to be Moore-closed (though we know it is still a closed cover).

7.5 State Minimization of PNDFSM's for Well-defined Behavior

In this section, we consider the problem of state minimization of a PNDFSM M_1 for some required behavior (e.g., well-defined behavior) when connected to another machine M_2. The unstated assumption of state minimization, each state in the reduced machine corresponds to one compatible set of states, is not true in general here. Classical compatible selection is no longer sufficient because a single compatible may correspond to more than one state in the reduced machine. Different states in M_2 may require the compatible in M_1 to be associated with different output functions. Watanabe *et al.* proposed we select a set of pairs (c, f) where c is a compatible and f is an output function associated with that compatible. By using as many table columns as there are such pairs, his method gives the binate solver the full flexibility of choosing combinations of

compatibles and output functions, so that a selection of compatibles guarantees a closed cover while satisfying some property associated with its output functions. Because it is difficult to represent the f part of the column pair (c, f), Watanabe's proposal is never seen as a practical algorithm.

7.5.1 Well-defined Pairs

We propose an alternative method which uses pairs (c, s_2) as columns in our binate table, where $s_2 \in S_2$ is a state in M_2 [9]. By representing pairs of the form (c, s_2) instead of (c, f), we have effectively reduced the size of the binate table to be solved. To explore well-defined behaviors contained in a PNDFSM, we must first justify that it is sufficient to consider only such (c, s_2) pairs. In particular, we show that for this problem, there is a one-to-one correspondence between each (c, s_2) pair and a potential state in a reduced DFSM.

Given a PNDFSM M, Theorem 2.20 shows that each behaviorally contained DFSM corresponds to a closed set of compatibles. This property is still true for the E-machine M which captures all permissible behaviors, in which we are to pick one which forms a well-defined product machine with M_2. Though it is sufficient to consider classical compatibles (for a PNDFSM), a compatible may be required (by well-definedness with M_2) to carry different conflicting output functions (and different sets of transitions) when it is associated with different states in M_2. As a result, this compatible would correspond to more than one state in the reduced machine M_1. Thus we associate each compatible c with a state s_2 of M_2 such that a different output function (and a different set of transitions) can be associated to each such (c, s_2) pair. With a similar motivation as Watanabe's, this column representation gives the binate solver the full flexibility to select a solution to our problem.

Now we derive a pruned set of such pairs, each of which may correspond to a state in a reduced machine M_1, and when composed with DFSM M_2 will produce a well-defined product with DFSM M_2. By Theorem 2.20, each DFSM representing a contained behavior within M is structurally contained in M^{power}. Taking account of well-defined composition, each DFSM representing a well-defined contained behavior within M is structurally contained in M^p defined as follows. The STG of M^p is an unrolling of the STG of M^{power} as many times as the number states in M_2. More formally, $M^p = \langle C \times S_2, U, V, T^p, (r_c, r_2)\rangle$ is defined where C is the set of compatibles, r_c is a compatible containing a reset state, and $T^p(u, (c, s_2), (c', s_2'), v) = 1$ if and only if $T^{power}(u, c, c', v) = 1$. Note that M^p still captures the same set of permissible behavior as

[9] Later, we will show that we actually use a subset of (c, s_2) pairs. Pruning is applied in two steps, basing on the reachability and well-definedness properties.

in the PNDFSM M (E-machine). The unrolled STG T^p is not in any way related to T_2, yet.

It is obvious that we only need to consider (c, s_2) pairs which are reachable from a reset pair (r_c, r_2) in M^p. Let S_p denote the set of such reachable pairs. M_p will be used to denote the machine derived from M^p with its state space restricted to S_p.

We can prune the set of (c, s_2) pairs further by considering only those that can result in well-defined behaviors. Theorem 7.2 gave a efficient test for well-defined product between two DFSM's. Here we address a more difficult problem: Given a DFSM M_2 and a PNDFSM M_p, we want to check the *existence* of a DFSM M_1 which is behaviorally contained in M such that $M_1 \times M_2$ is well-defined.

Theorem 7.8 *A DFSM M_1, structurally contained in a PNDFSM M_p derived from an E-machine, is well-defined with respect to M_2 if and only if*

$$\forall (s_1, s_2) \textit{ reachable from } (r_1, r_2)\ \forall x\ \exists z\ \exists s_1' s_2'\ \exists uv\ T_1(u, s_1, s_1', v) \cdot T_2(xv, s_2, s_2', uz)$$

Proof: By Theorem 7.2, the first part of the statement (the part in English) is true if and only if

$$\forall (s_1, s_2) \text{ reachable from } (r_1, r_2)\ \forall x\ !z\ \exists s_1' s_2'\ \exists uv\ T_1(u, s_1, s_1', v) \cdot T_2(xv, s_2, s_2', uz).$$

Considering the additional fact that M_1 is behaviorally contained in the E-machine, for each present state pair (s_1, s_2) and for each input x, there can be no more than one z value such that above formula is true. So we can safely replace the unique quantification over z by an existential quantification. ■

Instead of checking one behavior (i.e., one DFSM) at a time for well-definedness, we would prefer to check the E-machine as a whole, and find out if there is a DFSM behaviorally contained in it that is well-defined.

Theorem 7.9 *If there exists a DFSM M_1 structurally contained in PNDFSM M_p which is well-defined with respect to M_2, then*

$$\forall (s_1, s_2) \in S_p\ \forall x\ \exists z\ \exists c' s_2'\ \exists uv\ T_p(u, (c, s_2), (c', s_2'), v) \cdot T_2(xv, s_2, s_2', uz),$$

i.e.,

$$\forall (s_1, s_2) \in S_p\ \forall x\ \exists z\ \exists c' s_2'\ \exists uv\ T^{power}(u, c, c', v) \cdot T_2(xv, s_2, s_2', uz),$$

where S_p denotes the set of reachable state pairs.

If the previous test fails, there is no DFSM associated with S_p that is contained in M_p which is well-defined with respect to M_2. Given this well-definedness test, we defined our pruned set of pairs as well-defined pairs as follows.

Definition 7.12 *Given a PNDFSM M and a DFSM M_2,*
a state set c in M and a state s_2 in M_2 form a **well-defined pair** (c, s_2) *if*

1. *there exists a function $f_{(c,s_2)} : U \rightarrow V$ such that*
2. $\forall x \; \exists uz \; \exists$ *well-defined pair* (c', s_2') $T^{power}(u, c, c', f_{(c,s_2)}(u)) \cdot T_2(x f_{(c,s_2)}(u), s_2, s_2', uz)$.

Unfortunately, the above test cannot be trivially evaluated as a propositional formula. It tests the existence of a function that makes a formula true. We have to enumerate different functions and evaluate the formula for each functional assignment.

Theorem 7.10 *The set of well-defined pairs $P(c, s_2)$ can be computed by the following fixed point computation:*

- $P_0(c, s_2) = 1$ *if and only if c is a compatible of M and s_2 is a state in M_2, and the pair is reachable from reset states, i.e., $(c, s_2) \in S_p$*
- $P_{k+1}(c, s_2) = 1$ *if and only if there exists a function $f_{(c,s_2)} : U \rightarrow V$ such that*

$$\forall x \; \exists uz \; \exists (c', s_2') \in P_k \; T^{power}(u, c, c', f_{(c,s_2)}(u)) \cdot T_2(x f_{(c,s_2)}(u), s_2, s_2', uz).$$

The iteration can terminate when for some j, $P_{j+1} = P_j$ and the set of well-defined pairs is $P(c, s_2) = P_j(c, s_2)$.

For convergence to P, iterative formula testing is necessary because it is possible that the test first passes on a pair (c, s_2) and then one of its implied next state set (c', s_2') is removed during another iteration. As a result the pair (c, s_2) will no longer satisfy the test. So it should be removed also.

At each step of the above fixed point iteration, for each $(c, s_2) \in P_k$, we have to enumerate each function $f_{(c,s_2)}$ and test if it satisfies the logic formula. It seems to be computationally prohibitive because there can be many such functions possible. But the actual number of functions needed to be checked should be much smaller than expected. This is because for most values of u, $f_{(c,s_2)}(u)$ is constrained to take exactly one v value. As a result, we only have to enumerate functions by supplying different output values v for a small number of input values u.

7.5.2 Selecting a Well-defined Contained Behavior

After generating the set of well-defined pairs, we explore well-defined behaviors by finding DFSM's structurally contained in M_p. Given M_p, the following procedure will either (a) find a DFSM M_1 structurally contained in M_p which is well-defined with respect to M_2, or (b) conclude that no DFSM structurally contained in M_p is well-defined with respect to M_2.

If the reset state (r_c, r_2) has been deleted, then no DFSM structurally contained in the original M_p is well-defined with respect to M_2.

Else any DFSM structurally contained in the reduced M_p with reset state (r_c, r_2) is well-defined with respect to M_2 (and structurally contained in the original M_p). Such a structurally contained DFSM can always be found because (1) it has a reset state, (2) no matter which state (c, s_2) is chosen, we can always select an output function for it, and all its next states $(c', s_2') \in P$ can also be chosen. To find such a DFSM $M_1 = \langle S_1, U, V, T_1, (r_c, r_2)\rangle$, we simply start with an M_1 having a state (r_c, r_2) but without any transitions; at each state (c, s_2), an output function $f_{(r_c,r_2)}$ that passes the test is chosen arbitrarily [10], and the corresponding next states reached are added to S_1 while the corresponding transitions chosen are added to T_1. This iterative procedure can terminate when no new states are reached/added. Then M_1 is a solution to **P4**. We are not claiming optimality with this method.

The above procedure is similar to the one in Section 7.4.2 for minimal Moore behavior. A greedy strategy can also be applied here to obtain a heuristically minimal well-defined contained behavior.

7.5.3 Selecting a Minimum Well-defined Contained Behavior

To find a minimum well-defined contained behavior for **P5**, we solve the problem by augmenting the classical branch and bound algorithm for binate covering with well-definedness checks. The covering and closure clauses are expressed in terms of the first part of the selected well-defined pairs (i.e., compatibles c). A solution to this binate covering problem needs to be checked for well-definedness by testing Definition 7.12. This is necessary because closed cover of well-defined pair may not corresponds to a well-defined behavior. If the binate covering problem is such that many solutions have to be visited before we can conclude with an optimum one, the total time spend for performing this non-trivial well-definedness test may be considerable.

[10] For minimum well-defined behavior, f cannot be arbitrarily chosen. In this case, backtracking is necessary, e.g., using branch-and-bound procedure as in Section 7.5.3.

Chapter 8

Conclusions

This dissertation has focused on the state minimization of various kinds of finite state machines (FSM's). In particular, implicit algorithms were presented for state minimization of ISFSM's, PNDFSM's, and a PNDFSM within a network of FSM's.

In Chapter 2, we first presented a complete taxonomy on different kinds of FSM's and other related concepts. This served as a unified notational framework on which we developed new theories and new algorithms for state minimization. A proof of correctness was given for our state minimization algorithm for PNDFSM's in Chapter 6. Because an ISFSM is also a PNDFSM, our corresponding minimization algorithm in Chapters 4 and 5 is therefore proved as well.

The multi-valued decision diagram (MDD) was introduced in Chapter 3. MDD is an extension of BDD for representing multi-valued functions, and has many useful applications (see [36]). To explore sequential behaviors in a non-deterministic specification, 1-hot encoded MDD's were used as a compact representation of sets of sets (e.g., to represent subsets of compatibles). A suite of implicit techniques was developed for manipulating these set-theoretic objects. An assumption was made that the number of elements (i.e., states) is small as compared to the number of sets of elements (i.e., state sets). The computational time and space complexities for our representation and manipulation of such sets of sets are not directly proportional to their cardinalities, and in practice they are very efficient. The only explicit dependency is on the number of elements (i.e., states).

An exact state minimization algorithm consists of two steps: compatible generation and compatible selection. Chapter 4 presented an algorithm that implicitly generates the various subsets of compatibles needed to solve exactly the state minimization problem of ISFSM's. Compatibles, maximal compatibles, prime compatibles and implied classes are all represented implicitly using

1-hot encoded MDD's. If it is possible to construct these MDD's, computations on these sets of compatibles are easy. We have demonstrated with experiments from a variety of benchmarks that implicit techniques allow us to handle examples exhibiting a number of compatibles up to 2^{1500}, an achievement outside the scope of minimization programs based on explicit enumeration [63]. We have shown, when discussing the experiments, that ISFSM's with a very large number of compatibles may be produced as intermediate steps of logic synthesis algorithms, for instance in the cases of asynchronous synthesis [46], of learning I/O sequences [23], and of synthesis of interacting FSM's [65]. This shows that the proposed approach has not only a theoretical interest, but also practical relevance for current logic synthesis applications.

A fully implicit exact algorithm for solving the binate covering problem [68] was presented in Chapter 5. This is used as the final step of an implicit exact state minimization procedure for ISFSM's, to select a minimum closed cover of compatibles. A literature survey has been given for table reduction rules currently known to us. A complete formulation of an implicit binate covering algorithm has been worked out. A novel encoding scheme has been proposed where a binate table is represented by a set of row labels and a set of column labels, and as a result, individual table entries are not represented. We also showed how table reductions and various branch-and-bound operations can be performed on the binate covering table implicitly. Our implementation includes (1) a generic binate covering table solver, (2) a binate covering table solver which assumes each row has at most one 0, and (3) a specialized binate table solver fine-tuned for state minimization of ISFSM's. Results on binate covering were reported for our ISFSM state minimization application where tables with up to 10^6 rows and columns have been solved implicitly.

In Chapter 6, we have presented both theoretical and practical contributions to the problem of exploring contained behaviors and selecting one with a minimum number of states for classes of NDFSM's. In particular, we have contributed the following: (1) A theoretical solution to the problem of exact state minimization of general NDFSM's, based on the proposal of a notion of generalized compatibles. This gives an algorithmic foundation for exploring behaviors contained in a general NDFSM. (2) A fully implicit algorithm for exact state minimization of PNDFSM's. The results of our implementation are reported and shown to be superior to the explicit formulation described in [86]. We could solve exactly all the problems of the benchmark used in [83] (except one case, where a minimal solution not guaranteed to be the minimum was found). The explicit program could complete approximately one half of the examples, and in those cases with longer running times.

Current research in synthesis and verification requires the analysis and optimization of an

FSM operating within a network of interacting FSM's. Chapter 7 addressed the abstracted problem where two DFSM's M_1 and M_2 are interconnected in a feedback composition. We first characterize the necessary and sufficient condition when the composition $M = M_1 \times M_2$ is a well-defined FSM. Previously known conditions, such as requiring M_1 or M_2 to be Moore, or requiring M_1 to be 'implementable' as defined in [86], are sufficient conditions only. Then we considered a dual problem of practical interest: given a DFSM M_2 and a specification M, find a behavior permissible at M_1 such that $M_1 \times M_2$ is well-defined. Watanabe proved in [85] that the E-machine captures the maximum set of permissible behaviors. Our contribution is to propose improved algorithms to find such a behavior contained in the E-machine which is either (1) Moore, or (2) well-defined with respect to M_2. The first problem is intrinsically easier as M_2 need not be considered. The second problem requires checking the existence of well-defined behavior within a PNDFSM, which we reduced to checking the existence of a function which makes a formula true. For future work, an implementation is needed to show that this proposed method can be made efficient in practice.

To solve each of the two problems, we proposed both a heuristic procedure to find a minimal-state behavior and an exact algorithm to find a minimum-state behavior. The heuristic procedure holds the promise of being turned into an efficient computer program as future work. It involves forming a special power machine, pruning its state space, and then finding a DFSM structurally contained in the pruned power machine. The exact algorithm, on the other hand, requires classical binate table covering, augmented with Moore/well-definedness checks.

A key concept used throughout the dissertation is that of a behavior of a DFSM, and that an NDFSM specifies a set of behaviors. We showed that for a PNDFSM, each closed cover of compatibles corresponds to a contained behavior, and vice versa. An immediate consequence is that our two-step procedure of compatible generation and compatible selection is a complete procedure for exploring all sequential behaviors contained in a PNDFSM. In the case of state minimization, one wants a minimum cardinality closed collection of compatibles. But one can replace the requirement of minimum cardinality with any other desired cost function or property (such as well-definedness) and obtain a new behavior selection problem. Therefore the exploration of all contained behaviors is a key technology for future applications in the synthesis of sequential networks.

It is worth while to underline that besides the intrinsic interest of state minimization and its variants for sequential synthesis, the implicit techniques reported in this dissertation can be applied to other problems of logic synthesis and combinatorial optimization. For instance similar techniques are used in an implicit algorithm [81] for exact state encoding based on generalized prime implicants (GPI's). Also, the implicit computation of maximal compatibles given here can be easily converted

into an implicit computation of prime encoding-dichotomies (see [71]). Moreover, the binate covering problem is at the core of many CAD problems, and our general implicit binate covering algorithm should be useful in solving large binate tables for those applications as well. Therefore the computational methods described here contribute to build a body of implicit techniques whose scope goes much beyond a specific application.

Bibliography

[1] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, vol. 27:509–516, 1978.

[2] A. Aziz, F. Balarin, R. K. Brayton, M. D. Di Benedetto, A. Saldanha, and A. L. Sangiovanni-Vincentelli. Supervisory control of finite state machines. In *Proceedings of International Conference on Computer-Aided Verification*, 1995.

[3] A. Aziz, F. Balarin, S. Cheng, R. Hojati, T. Kam, S. Krishnan, R. Ranjan, T. Shiple, V. Singhal, S. Tasiran, H. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. HSIS: A BDD-based environment for formal verification. In *Proceedings of the Design Automation Conference*, pages 454–459, June 1994.

[4] M. Di Benedetto, A. Saldanha, and A. Sangiovanni-Vincentelli. Model matching for finite state machines. In *Proceedings of the IEEE Conference on Decision and Control*, December 1994.

[5] M. Di Benedetto, A. Saldanha, and A. Sangiovanni-Vincentelli. Strong model matching for finite state machines. *Submitted for publication*, September 1995.

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

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

[8] R. Brayton, A. Sangiovanni-Vincentelli, and G. Hachtel. Multi-level logic synthesis. *Proceedings of the IEEE*, vol. 78(no. 2):264–300, February 1990.

[9] R. Brayton, A. Sangiovanni-Vincentelli, G. Hachtel, and R. Rudell. *Multi-level logic synthesis.* unpublished book, 1992.

[10] R. Brayton and F. Somenzi. An exact minimizer for Boolean relations. In *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pages 316–319, November 1989.

[11] R. Bryant. Graph based algorithm for Boolean function manipulation. *IEEE Transactions on Computers*, vol. 35(no. 8):667–691, 1986.

[12] J. R. Burch, D. Dill, E. Wolf, and G. DeMicheli. Modelling hierarchical combinational circuits. In *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pages 612–617, November 1993.

[13] E. Cerny. Characteristic functions in multivalued logic systems. *Digital Processes*, vol. 6:167–174, June 1980.

[14] E. Cerny. Verification of I/O trace set inclusion for a class of non-deterministic finite state machines. In *Proceedings of the International Conference on Computer Design*, pages 526–530, October 1992.

[15] R. P. Colwell and R. L. Steck. A 0.6um BiCMOS processor with dynamic execution. In *Proceedings of IEEE International Solid-State Circuits Conference*, February 1995.

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

[17] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, June 1989.

[18] O. Coudert, H. Fraisse, and J. C. Madre. A new viewpoint on two-level logic minimization. In *Proceedings of the Design Automation Conference*, pages 625–630, June 1993.

[19] O. Coudert and J. C. Madre. Implicit and incremental computation of prime and essential prime implicants of Boolean functions. In *Proceedings of the Design Automation Conference*, pages 36–39, June 1992.

[20] M. Damiani. Nondeterministic finite-state machines and sequential don't cares. In *European Conference on Design Automation*, pages 192–198, 1994.

[21] M. Davio, J.-P. Deschamps, and A. Thayse. *Discrete and Switching Functions*. Georgi Publishing Co. and McGraw-Hill International Book Company, 1978.

[22] S. Devadas and R. Newton. Exact algorithms for output encoding, state assignment and four-level Boolean minimization. *IEEE Transactions on Computer-Aided Design*, vol. 10(no. 1):13–27, January 1991.

[23] S. Edwards and A. Oliveira. Synthesis of minimal state machines from examples of behavior. *EE290LS Class Project Report, U.C. Berkeley*, May 1993.

[24] S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Transactions on Computer-Aided Design*, vol. 39(no. 5):710–713, May 1990.

[25] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvements of Boolean comparison method based on binary decision diagrams. In *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pages 2–5, November 1988.

[26] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, 1979.

[27] J. Gimpel. A reduction technique for prime implicant tables. *IRE Transactions on Electronic Computers*, vol. 14:535–541, August 1965.

[28] A. Grasselli and F. Luccio. A method for minimizing the number of internal states in incompletely specified sequential networks. *IRE Transactions on Electronic Computers*, vol. 14(no. 3):350–359, June 1965.

[29] A. Grasselli and F. Luccio. Some covering problems in switching theory. In *Networks and Switching Theory*, pages 536–557. Academic Press, New York, 1968.

[30] J. Hartmanis and R. E. Stearns. Some dangers in the state reduction of sequential machines. *Information and Control*, vol. 5:252–260, September 1962.

[31] J. E. Hopcroft. n log n algorithm for minimizing states in finite automata. *Tech. Report Stanford Univ. CS 71/190*, 1971.

[32] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, 1979.

[33] R. W. House and D. W. Stevens. A new rule for reducing CC tables. *IEEE Transactions on Computers*, vol. 19:1108–1111, November 1970.

[34] S. Robinson III and R. House. Gimpel's reduction technique extended to the covering problem with costs. *IRE Transactions on Electronic Computers*, vol. 16:509–514, August 1967.

[35] S.-W. Jeong and F. Somenzi. A new algorithm for 0-1 programming based on binary decision diagrams. In *Proceedings of ISKIT-92, Inter. symp. on logic synthesis and microproc. arch., Iizuka, Japan*, pages 177–184, July 1992.

[36] T. Kam and R. K. Brayton. Multi-valued decision diagrams. *Tech. Report No. UCB/ERL M90/125*, December 1990.

[37] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Implicit generation of compatibles for exact state minimization. *Tech. Report No. UCB/ERL M93/60*, August 1993.

[38] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. A fully implicit algorithm for exact state minimization. In *Proceedings of the Design Automation Conference*, pages 684–690, June 1994.

[39] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Implicit state minimization of non-deterministic FSM's. In *Proceedings of the International Conference on Computer Design*, October 1995.

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

[41] E. Kuh and T. Ohtsuki. Recent advances in VLSI layout. *Proceedings of the IEEE*, vol. 78(no. 2):237–263, February 1990.

[42] L. Lavagno. Heuristic and exact methods for binate covering. *EE290ls Report*, May 1989.

[43] L. Lavagno. Personal communication. *UC Berkeley*, December 1991.

[44] L. Lavagno. *Synthesis and Testing of Bounded Wire Delay Asynchronous Circuits from State Transition Graphs.* PhD thesis, University of California, Berkeley, December 1992.

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

[46] L. Lavagno, C. W. Moon, R. K. Brayton, and A. Sangiovanni-Vincentelli. Solving the state assignment problem for signal transition graphs. In *Proceedings of the Design Automation Conference*, pages 568–572, June 1992.

[47] B. Lin. *Synthesis of VLSI designs with symbolic techniques*. PhD thesis, University of California, Berkeley, November 1991.

[48] B. Lin, O. Coudert, and J. C. Madre. Symbolic prime generation for multiple-valued functions. In *Proceedings of the Design Automation Conference*, pages 40–44, June 1992.

[49] B. Lin and A. R. Newton. Implicit manipulation of equivalence classes using binary decision diagrams. In *Proceedings of the International Conference on Computer Design*, pages 81–85, September 1991.

[50] B. Lin and F. Somenzi. Minimization of symbolic relations. In *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pages 88–91, November 1990.

[51] S. Malik. Analysis of cyclic combinational circuits. In *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pages 618–625, November 1993.

[52] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pages 6–9, November 1988.

[53] M. P. Marcus. Derivation of maximal compatibles using Boolean algebra. *IBM Journal of Research and Development*, pages 537–538, November 1964.

[54] E. McCluskey. Minimization of Boolean functions. *Bell System Technical Journal*, vol. 35:1417–1444, November 1956.

[55] M. McFarland, A. Parker, and R. Camposano. The high level synthesis of digital systems. *Proceedings of the IEEE*, vol. 78(no. 2):301–318, February 1990.

[56] R. E. Miller. *Switching theory. Volume II: sequential circuits and machines.* J. Wiley & Sons, Inc., N.Y., 1965.

[57] S. Minato. Zero-suppressed BDD's for set manipulation in combinatorial problems. In *Proceedings of the Design Automation Conference*, pages 272–277, June 1993.

[58] E. Moore. Gedanken-experiments on sequential machines. In C. Shannon and J. McCarthy, editors, *Automata Studies*. Princeton University Press, 1956.

[59] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.

[60] M. Paull and S. Unger. Minimizing the number of states in incompletely specified state machines. *IRE Transactions on Electronic Computers*, vol. 8:356–367, September 1959.

[61] C. P. Pfleeger. State reduction in incompletely specified finite state machines. *IEEE Transactions on Computers*, pages 1099–1102, October 1973.

[62] 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.

[63] J.-K. Rho, G. Hachtel, F. Somenzi, and R. Jacoby. Exact and heuristic algorithms for the minimization of incompletely specified state machines. *IEEE Transactions on Computer-Aided Design*, vol. 13(no. 2):167–177, February 1994.

[64] J.-K. Rho and F. Somenzi. STAMINA. *Computer Program*, 1991.

[65] J.-K. Rho and F. Somenzi. The role of prime compatibles in the minimization of finite state machines. In *Proceedings of the European Design Automation Conference*, 1992.

[66] F. Rubin. Worst case bounds for maximal compatible subsets. *IEEE Transactions on Computers*, pages 830–831, August 1975.

[67] R. Rudell. ESPRESSO. *Computer Program*, 1987.

[68] R. Rudell. *Logic synthesis for VLSI design.* PhD thesis, University of California, Berkeley, April 1989.

[69] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pages 42–47, November 1993.

[70] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Transactions on Computer-Aided Design*, vol. 6:727–750, September 1987.

[71] A. Saldanha, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. A uniform framework for satisfying input and output encoding constraints. *Proceedings of the Design Automation Conference*, pages 170–175, June 1991.

[72] S. C. De Sarkar, A. K. Basu, and A. K. Choudhury. Simplification of incompletely specified flow tables with the help of prime closed sets. *IEEE Transactions on Computers*, vol. 18:953–956, October 1969.

[73] E. Sentovich, K. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *Proceedings of the International Conference on Computer Design*, pages 328–333, October 1992.

[74] M. Servit and J. Zamazal. Exact approaches to binate covering problem. *Manuscript in preparation*, October 1992.

[75] T. Shiple, H. Touati, and G. Berry. Causality analysis of circuits. *Manuscript in preparation*, December 1994.

[76] F. Somenzi. COOKIE. *Computer Program*, 1989.

[77] F. Somenzi. Gimpel's reduction technique extended to the binate covering problem. *Unpublished manuscript*, 1989.

[78] A. Srinivasan, T. Kam, S. Malik, and R. Brayton. Algorithms for discrete function manipulation. In *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pages 92–95, November 1990.

[79] G. Swamy, R. Brayton, and P. McGeer. A fully implicit Quine-McCluskey procedure using BDD's. *Tech. Report No. UCB/ERL M92/127*, 1992.

[80] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. In *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pages 130–133, November 1990.

[81] T. Villa. *Encoding Problems in Logic Synthesis*. PhD thesis, University of California, Berkeley, May 1995.

[82] H.-Y. Wang and R. K. Brayton. Input don't care sequences in FSM networks. In *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pages 321–328, November 1993.

[83] Y. Watanabe. *Logic Optimization of Interacting Components in Synchronous Digital Systems*. PhD thesis, University of California, Berkeley, April 1994.

[84] Y. Watanabe and R. Brayton. Heuristic minimization of multi-valued relations. *IEEE Transactions on Computer-Aided Design*, vol. 12(no. 10):1458–1472, October 1993.

[85] Y. Watanabe and R. K. Brayton. The maximum set of permissible behaviors for FSM networks. In *IEEE International Conference on Computer-Aided Design*, pages 316–320, November 1993.

[86] Y. Watanabe and R. K. Brayton. State minimization of pseudo non-deterministic FSM's. In *European Conference on Design Automation*, pages 184–191, 1994.