Encoding Problems in Logic Synthesis

by

Tiziano Villa

Laurea in Matematica (Università Statale di Milano, Italy), 1977
Mathematical Tripos, Part III, D.A.M.T.P., Cambridge University, U.K., 1982
M.S. (University of California at Berkeley) 1987

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering:
Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Alberto Sangiovanni-Vincentelli, Chair
Professor Robert Brayton
Professor Shmuel Oren

1995

The dissertation of Tiziano Villa is approved:

Chair Date

Date

Date

University of California at Berkeley

1995

Abstract

Encoding Problems in Logic Synthesis

by

Tiziano Villa

Doctor of Philosophy in Engineering:
Electrical Engineering and Computer Sciences

University of California at Berkeley

Professor Alberto Sangiovanni-Vincentelli, Chair

A key step in the implementation of a digital system is to map a given symbolic representation into an implementable representation with two-valued logic variables. This step is called encoding and it impacts area, speed, testability and power consumption of the realized circuit.

I focus on algorithms to encode symbolic input and output variables of finite state machines (FSM's) represented by state transition graphs (STG's) or state transition tables (STT's), when the cost function is minimum two-level area. Various techniques developed here were applied or are applicable also to encoding problems with different cost functions and objectives. The technical contributions can be divided into two parts: algorithms based on heuristic symbolic minimization and algorithms based on minimization of generalized prime implicants (GPI's).

I present two main results about symbolic minimization: a new procedure to find minimal two-level symbolic covers, under face, dominance and disjunctive constraints, and a unified frame to check encodeability of encoding constraints and find codes of minimum length that satisfy them. This frame has been used for various types of encoding constraints arising in problems that range from encoding for minimum multi-level representation to race-free encoding of asynchronous FSM's. Experiments for different applications are reported.

Then I present two main results on symbolic minimization using GPI's: an implicit procedure to compute minimum or minimal encodeable covers of GPI's, and an implicit algorithm to solve table covering problems. The implicit procedure to find minimum encodeable covers of GPI's features an implicit algorithm to check encodeability of encoding constraints, and it uses the implicit table solver. The latter algorithm is a general binate table solver and as such it is applicable

to a variety of other applications. It has been applied also to select implicitly minimum contained behaviors in FSM state minimization. In the second part of the thesis the emphasis is on design of algorithms based on the manipulation of binary decision diagrams (BDD's). The reason is that symbolic minimization requires the construction and manipulation of very large sets that can be often constructed efficiently with BDD's.

_____

Professor Alberto L. Sangiovanni-Vincentelli
Dissertation Committee Chairman

A mia madre, Marta Ricorda, e a mio padre, Franco Villa,
e in memoria dei miei nonni, Cleonice e Giuseppe, Melania e Valente
To my mother, Marta Ricorda, and to my father, Franco Villa,
and in memory of my grandparents, Cleonice and Giuseppe, Melania and Valente

Contents

List of Figures

List of Tables

Acknowledgements

I am very grateful to Prof. Alberto Sangiovanni-Vincentelli, who has been my research advisor throughout all the years in graduate school and has made me possible to pursue higher education in Berkeley, by associating me to a very distinguished research group that he, more than anybody else, contributed to create. I hope to continue my association with him and to carry on his vision of rigorous mathematical modelling applied to relevant engineering problems.

Prof. Robert Brayton has been a constant source of scientific inspiration and has followed closely my progress, contributing with advice to most of my research activities. He has been a model of dedication to scholarship and gentleman's style.

Prof. Shmuel Oren kindly agreed to be in my Thesis Committee. He was part also of my Qualifying Committee, together with Prof. Jan Rabaey.

I was honoured to embark, 3 years ago, on a joint research project, SILK, with Timothy Kam. The results of this common endeavor are documented in many chapters of this dissertation and in published papers. Tim has always been a source of ingenious ideas, effective solutions and an example of personal and scholarly integrity. I will count him always as a great friend and a precious work colleague.

Since my early time in Berkeley I shared personal friendship and work cooperation with Luciano Lavagno and Alex Saldanha. I carried on fruitful research with Alex presented also in some chapters of this dissertation. I met Luciano as a colleague in Italy and since then we have been faithful friends and co-workers. I looked often for Rajeev Murgai to discuss a hard technical point or share a personal discussion, relying on his keen mind and human wisdom.

Among those with whom I interacted in the group a special mention goes to Arlindo De Oliveira, with whom I hope to continue a Southern European research connection, Yosinori Watanabe, Felice Balarin, William Lam, Huey-Yih Wang, Yuji Kukimoto, Rick McGeer, Gitanjali Swamy who contributed efficient software for a prototype of mine, and Tom Shiple. They were always available for discussing sticky points, sharing an impressive knowledge in their fields of expertise.

Among established professors and researchers outside, Fabio Somenzi, Ney Calazans, Sharad Malik, Giovanni De Micheli, Srinivas Devadas, Pranav Ashar, Bruce Holmer, and Kurt Keutzer were available to discuss technical questions and share literature. I thank Kurt for sending me a draft of a paper on computational complexity of logic synthesis. Prof. Eugene Goldberg of the Academy of Sciences of Belarus has been recently in contact with me, updating me on his progress

in topics of common interest. I look forward to intensify our cooperation.

The Berkeley CAD group has been a lively place thanks to the many outstanding students that are (or have been) part of it: Krishnan Sriram, Jagesh Sanghavi, Chris Lennard, Cho Moon, Serdar Tasiran, Szu-Tsung Cheng, Stephen Edwards, Vigyan Singhal, Adnan Aziz, Desmond Kirkpatrick, Mark Beardslee, Paul Stephan, Sunil Khatri, Harry Hsieh, Amit Narayan, Luca Carloni.

A special acknowledgement goes to my wife, Maria De Nigris, who has been a blessing for me since when we met and then started a family, enriched four years ago by the birth of Marta (and soon to be increased by a new member). Maria and Marta put up with me putting work duties above family duties. I am looking forward to long years of joyful life in common. I hope to prove to Maria that it was worthy for her to leave her secure life in Italy to join me here.

No words would be enough to thank my mother, Marta Ricorda, and father, Franco, for having nurtured me throughout the years, and always supported me in any possible way. Given that I was in a far away country during the last years, I was almost never to the side of my mother who has been fighting an uphill battle against illness. May the God of life give her strength and some more time to spend with us. To them, and to the grandparents, I dedicate this humble achievement as a token of gratitude.

Chapter 1

Introduction

1.1 Logic Synthesis of Sequential Behaviors

The task of logic synthesis is to produce a circuit that realizes a given behavior. We will be concerned with *sequential behaviors*, that can be defined as mappings of sequences of input vectors to sequences of output vectors. When the mapping of an input vector does not depend on the input vectors previously seen in the current input sequence, the behavior is said to be *combinational*. The original specification may be described in ways ranging from natural languages to formal hardware description languages or algorithmic formalisms. Often the wanted behavior is specified only on a subset of the input sequences, leaving the rest as a *don't care* condition to be freely exploited by the implementor, or the specification may admit some possible behaviors as equally acceptable. These situations are referred also as *non-determinism* of the specification. A given specification (or set of specifications, if we interpret nondeterminism as expressing a set of behaviors, out of which one is implemented) may be realized by a large variety of circuits all reproducing the wanted sequential behavior, but very different in terms of structure and characteristics.

An automatic way of synthesizing digital circuits is to input a description of the behavior in textual or graph format to an high-level synthesis system, that will perform scheduling and allocation and produce a register-transfer level description of the synthesized design, that consists of a controller and a data path. A controller captures the dynamics of a sequential behavior, while the data path operates on the data under the supervision of the controller. This RTL description can then be optimized by means of logic synthesis.

A controller can be produced by an high-level synthesis tool, or it can be provided directly by the designer or extracted from an already existing circuit. A controller is usually specified by

means of a *finite state machine (FSM)*, that is a discrete dynamical system translating sequences of input vectors into sequences of output vectors. FSM's are a formalism growing from the theory of finite automata in computer science. An FSM has a set of states and 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. From the other side, systems whose past history cannot be condensed in a finite number of states are not physically realizable. FSM's are usually represented by *state transition graphs (STG's)* and *state transition tables (STT's)*, that are equivalent ways of enumerating all transitions as edges of a graph or rows of a table. They can be seen as *symbolic two-level representations*, because they map naturally into two-level logic after encoding the states (and any other symbolic variables) with binary vectors. In the other words, the edges of the graph (rows of the table) can be interpreted as symbolic representations of *and-or* logic.

A typical logic synthesis procedure includes FSM restructuring, like *state minimization*, followed by a *state assignment* step to obtain a logic description that can be mapped optimally into a target technology. Often optimization is done first on a representation independent from the technology, as in the multi-level synthesis system SIS, where the number of literals of a Boolean network is minimized first, and then the Boolean network is mapped using the cells of a given library. Optimization and mapping depend not only on the target technology (PLA's, custom IC's, Standard Cells, Field Programmable Gate Arrays), but also on the cost functions: besides area, speed and power consumption are of growing importance. Moreover, issues like testing and verificability play an important role. At the end of logic synthesis a sequential behavior is represented by a set of logic gates. Views of a sequential behavior are shown in Fig. 1.1.

Given a system, the overall theoretical objective is to synthesize a circuit that optimizes a cost function involving area, delay, power and testability. It is very difficult to come up with mathematical models that capture the problem in its generality. Furthermore, only for very limited domains, e.g., two-level logic implementations, there is a clearly defined notion of optimality and algorithms to achieve optimality. Moreover, with complex cost functions and a very large solution space, a good model must not only be "exact", but also amenable to efficient synthesis algorithms on problems instances of practical interest. A way to cope with complexity is to pursue the optimization objective by breaking down the global problem into independent steps, each having a restricted cost function, at the expense of jeopardizing global optimality. For instance it is customary to minimize

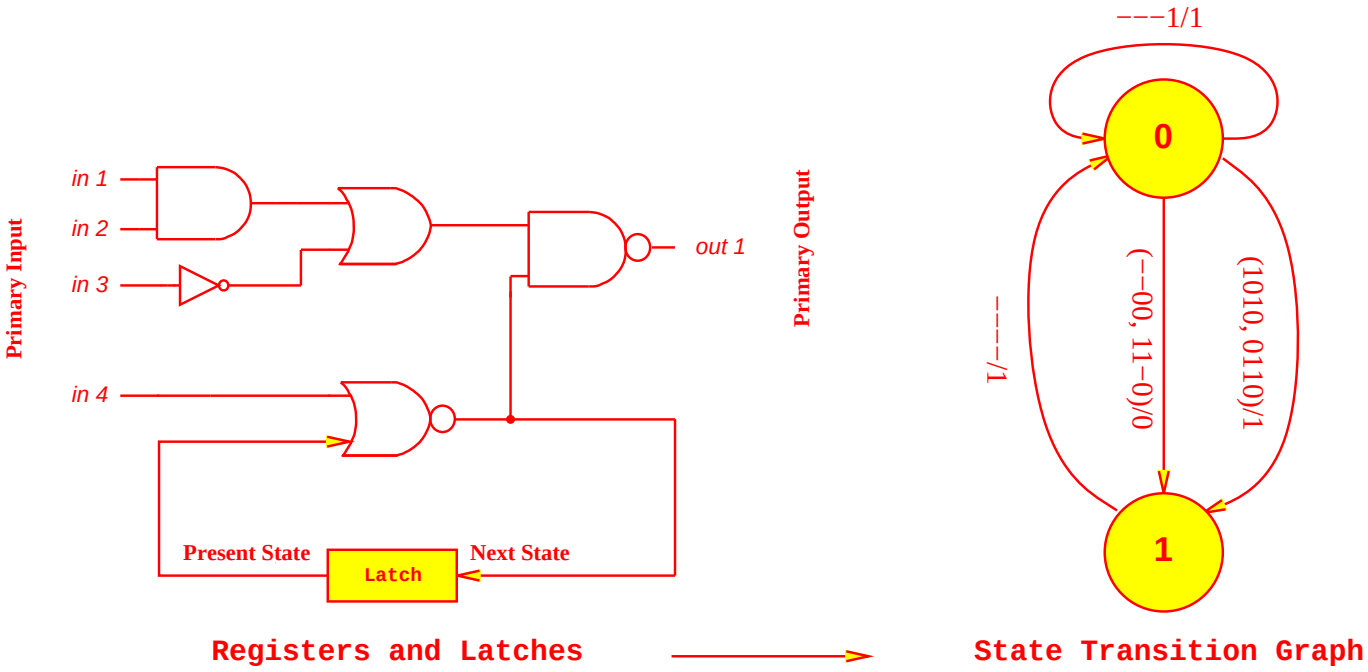


Figure 1.1: Views of a sequential circuit

the states of an FSM before encoding it: 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 [55], yet in practice this approach leads to good 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 alleviated.

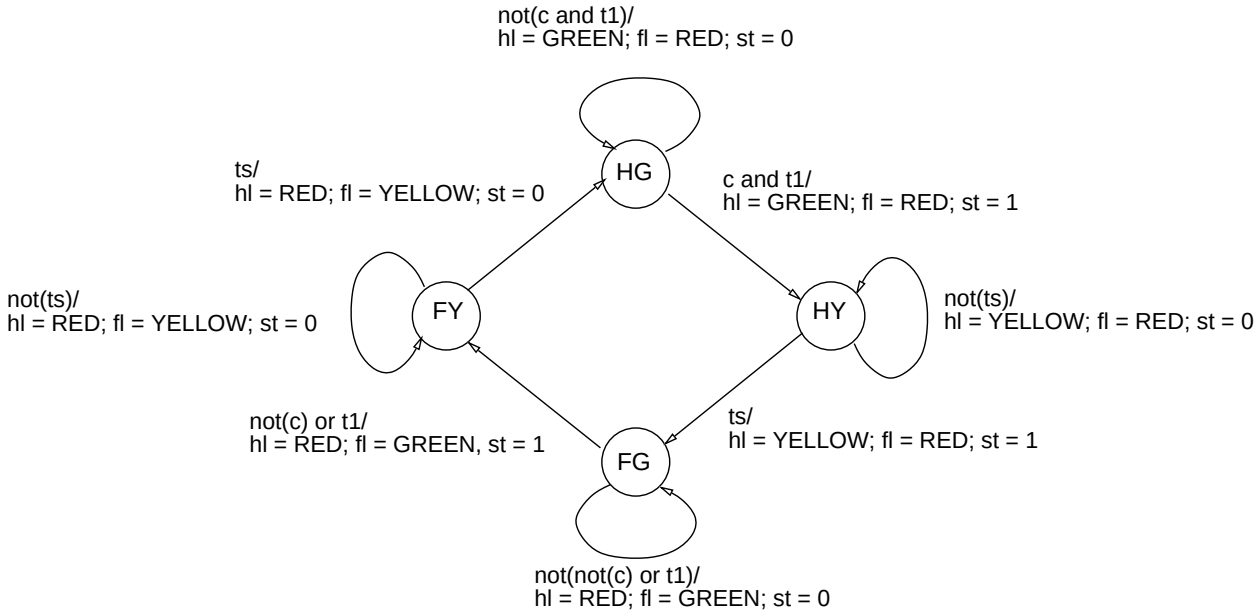
1.2 The Encoding Problem: from Symbolic to Boolean domain

The specification of a sequential behavior may include binary and symbolic variables. As an example, consider the well-known Mead-Conway traffic light controller [90]. Figure 1.2 presents a description in the BDS language from [127], as slightly modified in [84] to highlight the symbolic nature of the variables. An STG and STT representation of the FSM denoted by the BDS description is shown in Fig. 1.3.

The specific syntax and semantics of BDS are unimportant here: it suffices to say that they express naturally the evolution of the traffic light controller. Let us focus on the use of *symbolic variables*, i.e. variables that take on values from a set of symbols. For instance, the

```
MODEL traffic_light
    hl, fl          ! control for highway and farm lights
    st<0>,          ! to start the interval timer
    nextState =
    c<0>,           ! indicating a car on the farm road
    ts<0>, tl<0>    ! timeout of short and long interval timers
    presentState ;
ROUTINE traffic_light_controller;
    nextState = presentState; st = 0;
    SELECT presentState FROM
        [HG]: BEGIN
            hl = GREEN;   fl = RED;
            IF c AND tl THEN BEGIN
                nextState = HY; st = 1;
            END;
          END;
        [HY]:BEGIN
            hl = YELLOW;   fl = RED;
            IF ts THEN BEGIN
                nextState = FG; st = 1;
            END;
          END;
        [FG]: BEGIN
            hl = RED;   fl = GREEN;
            IF NOT c or tl THEN BEGIN
                nextState = FY; st = 1;
            END;
          END;
        [FY]:BEGIN
            hl = RED;   fl = YELLOW;
            IF ts THEN BEGIN
                nextState = HG; st = 1;
            END;
          END;
    ENDSELECT;
ENDROUTINE;
ENDMODEL;
```

Figure 1.2: Hardware description language representation of traffic light controller



State Transition Graph: Example

PS	IN	NS	OUT
HG	(not(c and t1)	HG	hl = GREEN; fl = RED; st = 0
HG	c and t1	HY	hl = GREEN; fl = RED; st = 1
HY	not(ts)	HY	hl = YELLOW; fl = RED; st = 0
HY	ts	FG	hl = YELLOW; fl = RED; st = 1
FG	not(not(c) or t1)	FG	hl = RED; fl = GREEN; st = 0
FG	not(c) or t1	FY	hl = RED; fl = GREEN; st = 1
FY	not(ts)	FY	hl = RED; fl = YELLOW; st = 0
FY	ts	HG	hl = RED; fl = YELLOW; st = 1

State Transition Table: Example

Figure 1.3: STG of traffic light controller example

variable representing the state of the traffic lights is represented in symbolic form and can take on four possible values. Similarly, the output variables representing the highway and farm lights are symbolic and can take on three values. Because there are symbolic variables, we say that this is a *symbolic specification*.

Current digital circuits can only store one of two values, since available storage elements are bistable circuits (even though experimental multistable circuits have been investigated and built). Therefore one says that symbolic variables need to be encoded, i.e., each symbolic variable must be replaced by a set of binary-valued (or two-valued) variables, to map an abstract specification onto a physical circuit. Let us examine more carefully the last statement.

Notice that also two-valued variables need to be encoded, for instance, given a variable C with values $\{green, red\}$ one might map $green$ to 0 and red to 1, or vice-versa. In this case, given a two-valued variable, one assigns to each of the two symbolic values one of the values of a variable defined on the Boolean algebra $\{0, 1\}$ (called a logic variable) [1]. Problems like optimal phase assignment of two-level logic attest that even the encoding of a two-valued variable may affect considerably the size of the final representation.

Therefore, rigorously speaking, *encoding* is the process of assigning to each value of a symbolic variable X a unique combination of values of a set of logic variables defined on $\{0, 1\}^n$. To have enough codes, it is necessary that $\log n \geq |X|$, where $|X|$ is the cardinality of X. Then the values of the encoding logic variables are mapped into stable levels of circuit signals. These subtle distinctions are often ignored in common parlance, so that one simply says that encoding is a map from values of symbolic variables to values of sets of binary variables.

When variables are defined on Boolean algebras, it is possible to use the formalism of the latter in the manipulation of logic circuits, as was discovered independently by Nakasima, Shestakov and Shannon [15].

For example, in the case of the traffic light controller, the four state values `HG`, `HY`, `FG`, `FY` may be represented as the bit patterns `00`, `01`, `10`, `11` on two binary-valued encoding variables. The resulting logic depends on the chosen encoding and so do area, performance and testability of the circuit. This gives rise to the *encoding problem* in logic synthesis wherein an encoding needs to be determined for a symbolic variable such that the resulting logic is optimal under some metric. The versions of the problem where the symbolic variables are inputs or outputs of the combinational logic are referred to as the input and output encoding problems respectively. An

[1] Alternatively, one could encode C with two logic variables, mapping, say, $green$ to 01 and red to 10.

FSM has a symbolic variable, the state, that appears both as input (present state) and output (next state) variable. The encoding problem for FSM's is referred to as the state assignment problem and is a case of input-output encoding, with the constraint that the values of the present state must be given the same codes as the values of the next states. This taxonomy was first introduced in [91].

1.3 Thesis Overview

This thesis focusses on algorithms to encode symbolic input and output variables of sequential behaviors represented by STG's or STT's, when the cost function is minimum two-level area. Various techniques developed here were applied or are applicable also to encoding problems with different cost functions and objectives.

We can divide the technical contributions into two parts: algorithms based on heuristic symbolic minimization (Chapters 5 and 6) and algorithms based on minimization of generalized prime implicants (Chapters 7, 8, 9 and 11). Minimization of GPI's required the development of implicit techniques developed in Chapters 11 and 10.

Let us clarify briefly the two approaches. Classical logic minimization aims to find a minimum sum-of-products [2] expression of binary-valued inputs binary-valued outputs functions [87]. It was extended to functions with multi-valued inputs and binary-valued outputs in [139, 114, 112], as multi-valued minimization.

This extension inspired a solution to the input encoding problem in [92], that was applied to encoding the present states of an FSM, and to other problems in combinational synthesis. The solution consists of performing a multi-valued minimization of the given function and then converting the result to a two-valued sum-of-products [3], by satisfying certain conditions on the codes of the states that are called input or face constraints. For any group of face constraints there is a satisfying encoding, but one wants to find codes of minimum code-length that satusfy the face constraints. We call encoding constraints any types of conditions imposed on the codes of a set of symbols. We call encodeable a symbolic sum-of-products whose encoding constraints are satisfiable.

When there are multi-valued output variables, we call *symbolic minimization*, according to the terminology established in [91, 147], the problem of finding a minimum symbolic sum-of-products that can be converted into a two-valued sum-of-products of the same cardinality. A

[2] A sum-of-product is also called a cover of product-terms.
[3] Also called an encoded sum-of-products.

procedure for symbolic minimization is complete if it can yield at least a minimum encoded sum-of-products. A procedure for symbolic minimization has a part to construct a cover of symbolic product-terms and a part to satisfy encoding constraints [4] that let transform the symbolic cover into an equivalent encoded cover. The encoding constraints required for a complete symbolic minimization procedure involve new types of conditions on the codes of the states, that go under the name of dominance, disjunctive and disjunctive-conjunctive constraints. In [91, 147] algorithms for symbolic minimization were proposed that used only face and dominance constraints.

The first part of this dissertation contains two main results on symbolic minimization: a new procedure to find minimal two-level symbolic covers, under face, dominance and disjunctive constraints (Chapter 5), and a unified frame to check encodeability of encoding constraints and find codes of minimum length that satisfy them (Chapter 6). This frame has been used for various types of encoding constraints arising in problems that range from encoding for minimum multi-level representation [5] to race-free encoding of asynchronous FSM's [74]. Experiments for different applications are reported.

The procedure for symbolic minimization presented in Chapter 5 is not complete because it is not able to explore all possible symbolic cubes needed to build minimum symbolic covers. Moreover, it does not use disjunctive-conjunctive constraints, that are required for completness in some cases. This is why it is described also as heuristic symbolic minimization, and it is reminiscent of the heuristic mode for classical minimization of two-valued logic. A complete symbolic minimization algorithm was proposed in [39]. It extends to the symbolic case the two main features of exact classic two-level minimization: generation of a set of product-terms sufficient to find at least a minimum cover, i.e. the prime implicants, and computation of a minimum cover as solution of a set covering problem, represented as a table covering problem [87]. To handle symbolic minimization, in [39] the notion of prime implicants is extended to the notion of generalized prime implicants (GPI's) and the set covering problem is extended to a constrained set covering problem, because it is not sufficient to find a minimum symbolic cover, but one must find a minimum encodeable symbolic cover, i.e., a minimum symbolic cover whose associated encoding constraints are satisfiable so that it can be mapped into an equivalent encoded cover. We will refer to this exact algorithm for symbolic minimization as minimization of GPI's.

The second part of this dissertation contains two main results on symbolic minimization

[4] More precisely, one must check satisfiability of sets of encoding constraints, and, if they are satisfiable, find codes of minimum length that satisfy them.

[5] Theory and algorithms for multi-level minimization of multi-valued input functions were presented in [85] and applied to the encoding of the present state of an FSM for minimum multi-level literals.

using GPI's: a novel theory of encodeability of GPI's (Chapter 9), an implicit procedure to compute minimum or minimal encodeable covers of GPI's (Chapter 11), and an implicit algorithm to solve table covering problems (Chapter 10) [6]. The implicit procedure to find minimum encodeable covers of GPI's features an implicit algorithm to check encodeability of encoding constraints, and it uses as a key subroutine the implicit algorithm to solve covering problems described in Chapter 10. The latter algorithm is a general binate table solver [7] and as such it is applicable to a variety of other applications. Indeed it was originally developed to select implicitly minimum contained behaviors in FSM minimization [66].

In the second part of the thesis the emphasis is on design of implicit algorithms. The reason is that symbolic minimization requires the construction and manipulation of very large sets (the set of GPI's, the set of encoding constraints and many others). Implicit techniques have been shown to outperform traditional methods in the task of computing the primes of logic functions [27, 53] and of solving unate covering tables [53, 29]. Therefore, we deemed our application to be the perfect challenge for implicit techniques and we did not save efforts to extend their capabilities.

We stress that GPI minimization is harder than standard logic minimization:

1. The number of GPI's is much larger than the number of primes of functions with multi-valued inputs binary-valued outputs.

2. Choosing a minimum cover of GPI's is not sufficient. The cover must be also encodeable, i.e. it must be possible to find encoding functions such that the chosen symbolic primes can be converted into two-valued primes. A consequence is that some of the traditional simplifications that can be applied to unate tables are disallowed [8].

In other words, potentially we are exploring all possible primes of all possible encodings. GPI's can be seen as templates of primes of encoded representations, by means of the existence of encodings that map symbolic cubes into two-valued cubes. Experimental results are reported that assess the progress made and the bottlenecks still remaining.

This thesis contains 10 main chapters. In Chapter 2 basic definitions regarding FSM's, Boolean logic, multi-valued minimization and Boolean networks are provided.

[6] We say that an algorithm is implicit if it represents and manipulates sets and functions using binary decision diagrams [16] as data structure.

[7] A binate table represents general product-of-sums expressions, while a unate table represents product-of-sums expressions with only positive literals

[8] Constrained binate covering appears also in different problems, like finding minimum contained behaviors of nondeterministic FSM's that can be composed with a given FSM [63].

In Chapter 3 the computational complexity of some key problems in logic minimization and state assignment is demonstrated.

In Chapter 4 a survey of previous approaches to state assignment and other encoding problems is presented. Special attention is given to the techniques based on symbolic minimization that are at the heart of the technical contributions reported in this dissertation.

In Chapter 5 we present a new algorithm for encoding input-output symbolic variables for two-level implementations. In particular the case of state assignments of FSM's is considered. The new algorithm is based on an extension of the scheme of symbolic minimization presented in [91] and obtains better results than previously known through state-of-art tools [147].

In Chapter 6 we present comprehensive algorithms to check encodeability of sets of encoding constraints, including face, dominance, disjunctive, conjunctive-disjunctive constraints. If a set of encoding constraints is encodeable, it is shown how to find codes of minimum code-length that satisfy them. These algorithms have been already used in various applications, including our symbolic minimization scheme, that motivated first their development.

In Chapters 7 and 8 a theory of GPI minimization is presented. The theory is an exact frame to solve input and output encoding problems targetting optimal two-level area implementations. The paradygm is based on extending the traditional notion of prime implicants to generalized prime implicants. Optimum state assignment for two-level implementation is solved by finding a minimum encodeable cover of GPI's. The theory of encodeability of GPI's is established in Chapter 9, with an host of new results based on the notions of *raising graphs* and *updating sets*.

In Chapter 10 an implicit solution of table covering problems and other implicit computations needed to solve implicit GPI minimization are presented. The algorithms described here may solve exactly binate table covering problems occurring in various phases of logic synthesis. In Chapter 11 implicit algorithms to generate and compute minimum encodeable sets of GPI's are presented. Results of a prototype implementation are discussed.

Finally Chapter 12 summarizes what has been achieved and what is left to be done.

Chapter 2

Definitions

2.1 Sequential Functions and their Representation

Sequential functions [1] transform input sequences into output sequences. A sequence is a function from the set of natural numbers to any set. Here we are interested only to finite sets and to "well-behaved" or "regular" sequential functions: those such that at any stage the output symbol depends only on the sequence of input symbols which have been already received and such that they can "hold" only a certain amount of the information received, i.e., they cannot always make use of all the information contained in that portion of the input sequence which has been received. Such sequential functions have been called *retrospective finitary* sequential functions by Raney [49]. A sequential function can be represented in many possible ways. A naive representation would be to give a collection of pairs of input and output sequences. Since these sequences are of unbounded length, this would not be a practical way. For the class of regular functions mentioned above it is possible to derive a finite state representation, that corresponds to the usual notion of finite state machine (FSM) [2].

2.2 Finite State Machines

Retrospective finitary functions admit of a finite state representation. We are now going to define formally FSM's, that are the most common way of representing a finite state system. We

[1] Called also sequential machines or mathematical machines. A sequential machine receives input symbols in a sequence, works on this sequence in some way, and yields a sequence of output symbols.

[2] We note that the notion of state is usually introduced at the structural level, but it can be done also at the function (or behavioral level) as shown by Raney [49].

will see that an FSM represents a "behavior", i.e., a regular sequential function and that collections of behaviors can be represented by adding non-determinism to the FSM, that so becomes a non-deterministic FSM (NDFSM). The same behavior of course may have many different representations. We will see that the chosen representation (or the one that happens to be available) affects the quality of the implemention derived by an encoding step.

Definition 2.2.1 *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.*

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 [32, 114]. 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 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 an edge $p \xrightarrow{i/o} n$, the FSM transits from state p on input i to state n with output o.

Definition 2.2.2 *Given an FSM* $M = \langle S, I, O, T, R \rangle$*, the state transition graph of* M*,* $STG(M) = \langle V, E \rangle$*, is a labeled directed graph 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* q*, and the edge is labeled by the input/output pair* i/o*.*

To capture flexibility in the next state n and/or the output o from a state p at 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 relations 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.

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 type of transitions allowed in T. FSM's can be categorized by answering the following questions:

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[3] $\Delta : I \times S \times S \rightarrow B$ and an output relation[4] $\Lambda : I \times S \times O \rightarrow B$ 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 $\Lambda(i, p, o) = 1$.

Definition 2.2.3 *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.*

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 2.2.4 *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 on the input).

The definition of Moore machine presented here is the one given by Moore itself in [96] and followed by other authors [148]. The key fact is that the output is associated with the present state. In other words, the common output associated to a given state, goes on all edges that leave

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

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

that state. This is a reasonable assumption when modeling an hardware system. However, it is common to find in textbooks [70, 58] 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 has the advantage that it is always possible to convert a Mealy machine into a Moore machine. 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 and only 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 2.2.5 *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 don't cares in the next states and/or outputs. We warn that even though "incompletely specified" is established terminology in the logic 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.

Other classes of NDFSM's have been recently characterized in logic synthesis applications. Most noticeable are pseudo non-deterministic FSM's (PNDFSM's) that are such that for each triple $(i, p, o) \in I \times S \times O$, there is a unique state n such that $T(i, p, n, o) = 1$ [5]. Since these machines are not of direct interest to the investigations reported in this dissertation we will not give a formal taxonomy.

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

[5] They are called "pseudo" non-deterministic because their underlying finite automaton is deterministic.

$T(i,p,n,o) = 1$, i.e., there is a unique transition from (i,p). In addition, R contains a unique reset state.

Definition 2.2.6 *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.*

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 2.2.7 *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.*

We now show that a DFSM realizes a behavior while an NDFSM realizes a set of behaviors.

Definition 2.2.8 *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 2.2.9 *A* **trace set** *is simply a set of traces.*

Definition 2.2.10 *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 $\mathcal{L}(M|_{s_0})$ [6], 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 2.2.11 *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 $\forall k : 0 \leq k \leq j$,*

[6] 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 $\mathcal{L}(M|_{s_0})$.

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

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 2.2.12 *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 an initial 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 specify that one of a 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.

2.3 Taxonomy of Encoding Problems

Synthesis of an FSM is the process of producing an implementation starting from a behavioral specification of a sequential function. In our case we suppose that the starting point is an STG or an STT. Combinational functions are FSM's with only one state. It was mentioned in the introduction that we assume the usual paradigm of state minimization followed by state assignment, even though our encoding techniques do not depend on it.

The step that translates a representation where some variables are symbolic into one where they are all binary-valued is called encoding. An encoding must at least be correct, which means that the encoded representation must behave as the symbolic representation (usually an encoding must establish an injection from symbols to codes), but more interestingly it is often required that the encoded implementation satisfies some further condition or optimality criterion. For instance, suppose that the encoded representation must be implemented with two-level logic, then a definition of optimum encoding may be that the encoded implementation after two-level minimization has smallest area, or smallest number of product-terms. Also an encoding can be used to enforce a structural property, like testability, of the encoded representation, i.e. that it is possibile to find sequences of input vectors that distinguish a good and a faulty physical realization of the encoded representation.

It should be noticed that also the choice of the memory element (JK or RS or T or D flip-flop) matters since an encoding can be optimal with one type of bistable, but not with another one. We will assume that unless otherwise stated memory elements are of type D, i.e., they simply transfer the input to the output at the appropriate time.

There is almost no end to the variations of optimality objectives that can be imposed, according to different applications. We will review later a number of them.

Definition 2.3.1 *Given the sets of symbols* $S_i = \{s_{i_1}, s_{i_2}, \cdots, s_{i_p}\}$ *and* $S_o = \{s_{o_1}, s_{o_2}, \cdots, s_{o_q}\}$, *and a Boolean function:*

$$f : \{0,1,2\}^n \times S_i \rightarrow S_o \times \{0,1,2\}^m,$$

an **input-output encoding** *is given by a pair of integers* k_i, k_o *and a pair of injective functions* $e_i : S_i \rightarrow \{0,1\}^{k_i}$ *and* $e_o : S_i \rightarrow \{0,1\}^{k_o}$. *The encoded representation of* f, *i.e. the representation of* f *where the symbols are replaced by Boolean vectors in* B_{k_i} *and* B_{k_o}, *according to* e_i *and* e_o , *is denoted by* f_{e_i,e_o}.

Notice that the case of more than one symbolic variable in the input or output part can be treated similarly [7].

The definition of encoding can be specialized if symbolic variables appear only as input or output variables. For instance, if symbolic variables appear only as input variables, one has an input encoding problem:

Definition 2.3.2 *Given a set of symbols $S_i = \{s_1, s_2, \cdots, s_p\}$ and a Boolean function:*

$$f : \{0,1\}^n \times S_i \rightarrow \{0,1,2\}^m,$$

an **input encoding** *is given by an integer k_i and an injective function $e_i : S_i \rightarrow \{0,1\}^{k_i}$. The encoded representation of f, i.e. the representation of f where the symbols are replaced by Boolean vectors in B_{k_i}, according to e_i, is denoted by f_{e_i}.*

If symbolic variables appear only as output variables, one has an output encoding problem:

Definition 2.3.3 *Given a set of symbols $S_o = \{s_1, s_2, \cdots, s_q\}$ and a Boolean function:*

$$f : \{0,1\}^n \rightarrow S_o \times \{0,1,2\}^m,$$

an **output encoding** *is given by an integer k_o and an injective function $e_o : S_o \rightarrow \{0,1\}^{k_o}$. The encoded representation of f, i.e. the representation of f where the symbols are replaced by Boolean vectors in B_{k_o}, according to e_o, is denoted by f_{e_o}.*

Since e is an injective function, different symbols are mapped into different codes and so the encoded representation behaves as the symbolic representation.

Definition 2.3.4 *Given an operator $\mathcal{O}$ an encoding $\tilde{e}$ is optimal with respect to $\mathcal{O}$ if*

$$\mathcal{O}(f_{\tilde{e}}) = opt_e(\mathcal{O}(f_e)).$$

As an example, $\mathcal{O}$ can be the cardinality of a two-level minimized encoded cover of f_e and opt the minimum.

Definition 2.3.5 *Given a decision operator $\mathcal{O}$ an encoding $\tilde{e}$ satisfies $\mathcal{O}$ if:*

$$\mathcal{O}(f_{\tilde{e}})$$

is true.

[7] Let V_1 and V_2 be two symbolic variables taking values from sets S_{V_1} and S_{V_2} respectively. These may be replaced by a single symbolic variable V taking values from $S_{V_1} \times S_{V_2}$. This is in fact potentially better than considering V_1 and V_2 separately since the encoding for V takes into account the interactions between V_1 and V_2.

As an example, $\mathcal{O}$ can be a testability procedure that given an encoded cover returns true if it is testable, false otherwise.

Some encoding problems may have various sets of symbolic variables with mutual dependencies. A well-known one is the problem of assigning codes to the states of FSM's, where the state variable appears both as input variable (present state) and output variable (next state). Therefore a common value must be assigned to the same symbol in the present state and next state variable. We are going to repeat the definition for the state encoding or state assignment problem, since it is one of the most widely studied encoding problems.

Definition 2.3.6 *Given the sets of symbols* $S = \{s_1, s_2, \cdots, s_p\}$ *and an FSM with transition function:*

$$f : \{0,1,2\}^n \times S \rightarrow S \times \{0,1,2\}^m,$$

a **state assignment** *or* **state encoding** *is given by an integer* k *and an injective functions* $e : S \rightarrow \{0,1\}^k$. *The encoded representation of* f*, i.e. the representation of* f *where the symbols are replaced by Boolean vectors in* B_k*, according to* e*, is denoted by* f_e.

The optimality criteria investigated for state assignment have been more commonly the best two-level or multi-level area of the encoded circuit. Attention has been paid also to state assignment of asynchronous circuits, where one must guarantee correctness, for instance that the change of the state of the circuit does not depend on races among the transitions of signal values. Some work has been done on state assignment for testability. Little work has been done on state assignment to improve performance. Recently state assignment for low-power has received some attention, likely to grow in the near future.

2.4 Behavior vs. Structure in Encoding Problems

Some issues deserve discussion at this point. Does the encoding always need to be a function or can it be a mapping that assigns more than one code to a state (still preserving the fact that a code cannot be assigned to more than one symbol) ? The answer is that in general it is possible to derive e as a mapping that is not a function. Given n symbols to encode, one needs at least $k = log\ n$ bits to distinguish them. The difference $n - 2^k$ gives the number of spare codes that are available and could be used to assign more than one code to a state. Therefore one could replace "function e" with "relation e" in the previous definitions. An intermediate degree of freedom would be to define e as a function into the set $\{0, 1, *\}$, where $*$ denotes for output encoding a don't care

condition, for input encoding both 0 and 1. One must say that rarely existing encoding algorithms are able to exploit directly this degree of freedom, so we choose the more restrictive definition where f is a function, unless otherwise specified [8].

Does the encoding function need to be always injective or two different symbols can be given the same code ? The answer is that in general injectivity is necessary, unless in a given application one has an equivalence relation among the symbols such that symbols in the same class of the equivalence relation do not need to be distinguished. We will see later such examples, as for instance equivalent states of a CSFSM.

It is important to underline that the optimality of an encoding can only be guaranteed with respect to the starting symbolic representation. To be more specific, consider optimal state assignment. If we start with a given symbolic cover of a CSFSM and try, say in an exhaustive way, all possible encodings and choose the best according to a given cost function, we cannot rule out that a different symbolic cover representing the same behavior can produce, after encoding, a better result. The fact is that a CSFSM represents a behavior and that many different representations can be given of the same behavior. We do not know how to explore all possible representations of a behavior, for instance all possible STG's [9]. So we restrict our notion of optimality to the best that can be done starting from a given representation.

The situation is even more complex with state assignment of ISFSM's. An ISFSM represents a collection of behaviors. We will see that optimal state assignment procedures for two-level implementations have a limited capability of exploring different behaviors by the flexibility of choosing how to implement the don't care transitions (edges not specified or partially specified in the description). But they cannot explore all possible contained behaviors as for instance it is done by computing closed covers of compatible sets of states (a set of states is compatible if for every sequence there is at least one output sequence that all the states in the set can produce). Therefore when doing state assignment of an ISFSM (or another type of FSM that contains more than one behavior), one must gauge the optimality of state assignment against the fact that neither all behaviors nor all representations of the same behavior can be explored (unless otherwise shown). Some proposals to use more behavioral information when encoding CSFSM's (equivalent states) and ISFSM's (compatible states) will be seen later.

[8] Unused codes are usually given as don't care conditions when the smallest area of an encoded representation is obtained.

[9] Of course it may not be necessary to explore all possible STG's representing a given behavior, one should characterize the class of interesting STG's with respect to a certain notion of optimal encoding.

2.5 Boolean Algebras and Boolean Functions

This section provides a brief review of the background material on Boolean algebras and Boolean functions. There are many classical expositions of it. We refer to [15, 32] for a complete treatment.

Definition 2.5.1 *Consider a quintuple* $(B, +, \cdot, 0, 1)$ *in which* B *is a set, called the carrier,* $+$ *and* $\cdot$ *are binary operations on* B*, and* 0 *and* 1 *are distinct members of* B*. The algebraic system so defined is a* **Boolean algebra** *provided the following postulates are satisfied:*

1. Commutative Laws. *For all* $a, b \in B$*:*

$$\begin{aligned} a + b &= b + a \\ a \cdot b &= b \cdot a \end{aligned}$$

2. Distributive Laws. *For all* $a, b, c \in B$*:*

$$\begin{aligned} a + (b \cdot c) &= (a + b) \cdot (a + c) \\ a \cdot (b + c) &= (a \cdot b) + (a \cdot c) \end{aligned}$$

3. Identities. *For all* $a \in B$*:*

$$\begin{aligned} 0 + a &= a \\ 1 \cdot a &= a \end{aligned}$$

4. Complements. *For any* $a \in B$*, there is a unique element* $a' \in B$ *such that:*

$$\begin{aligned} a + a' &= 1 \\ a \cdot a' &= 0 \end{aligned}$$

Useful identities can be derived from the axioms. Of very common use are De Morgan's laws.

Definition 2.5.2 *Given a Boolean algebra* B*, the set of* **Boolean formulas** *on the* n *symbols* $x_1, x_2, \ldots, x_n$ *is defined by the following rules:*

1. *The elements of* B *are Boolean formulas.*
2. *The symbols* $x_1, x_2, \ldots, x_n$ *are Boolean formulas.*

3. *If g and h are Boolean formulas, then so are:*

   (a) $(g) + (h)$

   (b) $(g) \cdot (h)$

   (c) $(g)'$

4. *A string is a Boolean formula if and only if its being so follows from finitely many applications of the rules above.*

Definition 2.5.3 *An n-variable function $f : B^n \mapsto B$ is called a Boolean function if and only if it can be expressed as an n-variable Boolean formula.*

2.6 Discrete Functions as Boolean Functions

Many functions needed to specify the behavior of digital systems are binary-valued functions of binary-valued variables ($\{0,1\}^n \mapsto \{0,1\}$). These are also referred to as *switching functions* [15]. The fact that all switching functions are also Boolean functions [15] enables all properties of Boolean functions to be directly applied to switching functions.

However not all functions that arise in the context of circuit specification and design are switching functions. We are especially interested here to those functions, like the transition function of an FSM, that are usually given as symbolic functions. These symbolic or **discrete** functions are in the most general case multiple-valued functions of multiple-valued variables, It would be very useful if discrete functions would be Boolean functions, as switching functions are. Apparently this is not the case, or at least one should check case by case if the requirements for being Boolean functions are satisfied. Fortunately one can associate to a discrete function a Boolean function which can be used to represent and manipulate the discrete function, capitalizing on all the niceties of Boolean algebra, including compactness of representation. This association can be done both if one takes the relational or functional view of a discrete function. This section is heavily indebted to the exposition in [84].

Let $f : P_0 \times P_1 \times \ldots \times P_{n-1} \mapsto P_n$ be a discrete function with $P_j = \{0, 1, \ldots, p_{j-1}\}$. Let $P = \{P_0 \times P_1 \times \ldots \times P_n\}$. f is not a Boolean function since it does not meet the condition that $f : B^n \mapsto B$ for some Boolean algebra B. Corresponding to f there is the relation $R \subseteq P$ defined in the natural way as the set of points in P consistent with f. Let $B = 2^P$, the power set of P, i.e.

the set of all subsets of P. B is a Boolean algebra described by $(2^P, \cup, \cap, \phi, P)$. Let $\xi : B \mapsto B$ be defined as:

$$\xi(x) = R \cap x \quad x \in B \tag{2.1}$$

$R \cap x$ is a Boolean formula and hence ξ is a Boolean function. Equation 2.1 is the *minterm canonical form* for this function.

Let $m \in \{P_0 \times P_1 \times \ldots \times P_{n-1}\}$ and $\psi(m) = \{m\} \times P_n$. $\psi(m)$ is the set of $n+1$-tuples corresponding to the n-tuple m that have all p_n possible values in the last field. ξ corresponds to f in the sense that given any m, $f(m)$ may be computed by ξ as follows. $\xi(\psi(m))$ is a singleton set containing the tuple in R with the first n fields the same as that of m. Field $n+1$ in this tuple is $f(m)$.

Example 2.6.1 *The switching function corresponding to an* AND *gate is used to illustrate the above. Here* $f : \{0,1\}^2 \mapsto \{0,1\}$. *Consider* $m = (0,1)$.

$$\begin{aligned} R &= \{(0,0,0),(0,1,0),(1,0,0),(1,1,1)\} \\ \psi(m) &= \{(0,1)\} \times \{0,1\} \\ &= \{(0,1,0),(0,1,1)\} \\ \xi(\psi(m)) &= \{(0,0,0),(0,1,0),(1,0,0),(1,1,1)\} \cap \{(0,1,0),(0,1,1)\} \\ &= \{(0,1,0)\} \end{aligned}$$

$f(m)$ *is the last field of the* $n+1$*-tuple* $(0,1,0)$*, i.e.* $f(m) = 0$.

Example 2.6.2 *Each person in a certain university town in to be classified as being one of* {good, bad, ugly} *(abbreviated as* $\{g, b, u\}$*). This classification is to be done based on the person's occupation which is one of* {professor, teaching assistant, outlaw} *(abbreviated as* $\{p, t, o\}$*) and their nature which is one of* {honest, selfish, cruel} *(abbreviated as* $\{h, s, c\}$*). To be good you have to be a professor or be honest and not an outlaw. Cruel outlaws are ugly. Everyone else is just bad.*

The classification function is a discrete function $f : \{p,t,o\} \times \{h,s,c\} \mapsto \{g,b,u\}$. *Consider* $m = (t,c)$.

$$\begin{aligned} R &= \{(p,h,g),(p,s,g),(p,c,g),(t,h,g),(t,s,b),(t,c,b),(o,h,b),(o,s,b),(o,c,u)\} \\ \psi(m) &= \{(t,c)\} \times \{g,b,u\} \\ \xi(\psi(m)) &= R \cap \psi(m) \\ &= \{(t,c,b)\} \end{aligned}$$

$f(m)$ is the last field of the $n+1$-tuple (t,c,b), i.e. $f(m)=b$.

By clustering points in the domain one can get a more compact representation of f. Let $B = 2^P$ and $m \in P$. Let $m[j]$ be the value of field j in m. One natural way to cluster points in P is to group all points with the same value of $m[j]$ (for some given j) together and refer to them collectively. Let $\chi_j^{S_j} = P_0 \times \ldots \times P_{j-1} \times S_j \times P_{j+1} \times \ldots \times P_n$. Thus, $\chi_j^{S_j}$ has all points for which $m[j] \in S_j$. For Example 2.6.2 $\chi_0^{\{p\}}$ is the set of all points for which $m[0] = p$. Note that $\chi_j^{S_j} \in B$ and $\overline{\chi_j^{S_j}} = \chi_j^{P_j - S_j}$.

Theorem 2.6.1 *Let $\chi = \{\chi_j^{S_j} | j \in \{0,1,\ldots,n\}, S_j \subseteq P_j\}$. Let $b \in B$. b can be expressed in terms of a Boolean expression restricted to elements of χ.*

Proof: The statement needs to be proven only for the atoms of B, the singleton sets, since any other element of B can be obtained by a union of the atoms. Let $\{a\}$ be an atom and $a[i]$ be field i of a, then $a = \cap_i \left(\chi_i{}^{\{a[i]\}}\right)$. ■

An immediate corollary of this result is that R can be expressed as a Boolean expression restricted to elements of χ. Thus the Boolean formula in Equation 2.1 can be re-written by expressing R as a Boolean expression restricted to the elements of χ. In practice Theorem 2.6.1 is not used to re-write R, but rather R is derived directly from some description of the function.

Example 2.6.3 *Consider f in Example 2.6.2. R is derived directly from the conditions specified as follows. The set of points in P that represent professors or honest people who are not outlaws is naturally expressed as:*

$$\chi_0^{\{p\}} \cup \left(\chi_1^{\{h\}} \cap \overline{\chi_0^{\{o\}}}\right)$$

This can be simplified to:

$$\chi_0^{\{p\}} \cup \left(\chi_1^{\{h\}} \cap \chi_0^{\{t\}}\right)$$

Similarly the set of points that represent cruel outlaws is: $\chi_0^{\{o\}} \cap \chi_1^{\{c\}}$. The rest of the people are obviously expressed as:

$$\overline{\chi_0^{\{p\}} \cup \left(\chi_1^{\{h\}} \cap \chi_0^{\{t\}}\right) \cup \left(\chi_0^{\{o\}} \cap \chi_1^{\{c\}}\right)}$$

This can be simplified to:

$$\left(\chi_0^{\{t\}} \cap \chi_1^{\{s,c\}}\right) \cup \left(\chi_0^{\{o\}} \cap \chi_1^{\{h,s\}}\right)$$

Thus, R can be expressed as:

$$(\chi_2^{\{g\}} \cap (\chi_0^{\{p\}} \cup (\chi_1^{\{h\}} \cap \chi_0^{\{t\}}))) \cup (\chi_2^{\{b\}} \cap ((\chi_0^{\{t\}} \cap \chi_1^{\{s,c\}}) \cup (\chi_0^{\{o\}} \cap \chi_1^{\{h,s\}}))) \cup (\chi_2^{\{u\}} \cap (\chi_0^{\{o\}} \cap \chi_1^{\{c\}}))$$

In conclusion, taking the relational view of a discrete function, we have associated a Boolean function to a discrete function and described how the Boolean formula corresponding to this Boolean function can be represented compactly. This enables to apply any Boolean identity to simplify the Boolean expression.

Instead of the relational view, we can manipulate discrete functions taking the functional view. Suppose that the domain is partitioned based on the value of the function. Let $\Pi = \{\pi_0, \pi_1, \ldots, \pi_{p_n-1}\}$ be a partition of $P_0 \times P_1 \times \ldots \times P_{n-1}$ such that: $m \in \pi_i \Leftrightarrow f(m) = i$. For the switching function f in Example 2.6.1, $\pi_0 = \{(0,0),(0,1),(1,0)\}$ and $\pi_1 = \{(1,1)\}$.

Each π_i may be described by its characteristic function $\tilde{f}_i$ defined as follows.

$$\tilde{f}_i : P_0 \times P_1 \times \ldots \times P_{n-1} \mapsto \{0,1\} \quad i \in P_n$$

$$\tilde{f}_i(m) = \begin{cases} 1 & \text{if } m \in \pi_i \\ 0 & \text{otherwise} \end{cases}$$

$\tilde{f}_i$ tests for membership in π_i; it evaluates to 1 for exactly the points in π_i. The following representation has been commonly used to describe the $\tilde{f}_i$ in the literature. Let $S_j \subseteq P_j$ and X_j be a p_j-valued variable. $X_j^{S_j}$ is termed a literal of X_j and is defined as:

$$X_j^{S_j}(m) = \begin{cases} 1 & \text{if } m[j] \in S_j \\ 0 & \text{otherwise} \end{cases}$$

$X_{j_1}^{S_{j_1}} \cdot X_{j_2}^{S_{j_2}}$ is defined as the logical AND of $X_{j_1}^{S_{j_1}}$ and $X_{j_2}^{S_{j_2}}$. Similarly, $X_{j_1}^{S_{j_1}} + X_{j_2}^{S_{j_2}}$ is defined as the logical OR of $X_{j_1}^{S_{j_1}}$ and $X_{j_2}^{S_{j_2}}$. The complement of a literal $X_j^{S_j}$ is denoted as $\overline{X_j^{S_j}}$ and defined as $\overline{X_j^{S_j}} = X_j^{P_j - S_j}$. In this way sum-of-products (SOP's) and factored forms (recursive products and sums of SOP forms) are constructed.

Example 2.6.4 *For f in Example 2.6.1 the following is the SOP representation of the $\tilde{f}_i$:*

$$\tilde{f}_0 = X_0^{\{0\}} + X_1^{\{0\}}$$
$$\tilde{f}_1 = X_0^{\{1\}} \cdot X_1^{\{1\}}$$

Example 2.6.5 *For f in Example 2.6.2 the following is the SOP representation for $\tilde{f}_g$ and $\tilde{f}_u$:*

$$\tilde{f}_g = X_0^{\{p\}} \cup \left(X_1^{\{h\}} \cap \overline{X_0^{\{o\}}} \right)$$
$$\tilde{f}_u = X_0^{\{o\}} \cap X_1^{\{c\}}$$

However, there seems to be no direct way to obtain $\tilde{f}_b$ since these expressions are not Boolean and De Morgan's identities cannot be directly applied in this case, and if they do apply, it must be proven separately for each expression. This is a limitation of this representation.

Along the lines of the previous derivation of the Boolean function ξ from the relation R, one can obtain ξ from the expressions representing the characteristic functions of the partitions. First one derives a Boolean formula for each factored form expression. Then these Boolean formulas are combined to give $\xi(x)$. So all properties for Boolean formulas hold for factored form expressions and they need not be proven separately. We refer to [84] for a detailed derivation, that we simply demonstrate on an example.

Example 2.6.6 *For the switching function in Example 2.6.1:*

$$\begin{aligned}
\chi_0^{\{0\}} &= \{(0,0,0),(0,0,1),(0,1,0),(0,1,1)\} \\
\chi_1^{\{0\}} &= \{(0,0,0),(0,0,1),(1,0,0),(1,0,1)\} \\
\chi_0^{\{1\}} &= \{(1,0,0),(1,0,1),(1,1,0),(1,1,1)\} \\
\chi_1^{\{1\}} &= \{(0,1,0),(0,1,1),(1,1,0),(1,1,1)\} \\
\chi_2^{\{0\}} &= \{(0,0,0),(0,1,0),(1,0,0),(1,1,0)\} \\
\chi_2^{\{1\}} &= \{(0,0,1),(0,1,1),(1,0,1),(1,1,1)\} \\
\xi(x) &= \left(\left(\chi_0^{\{0\}} \cup \chi_1^{\{0\}}\right) \cap x \cap \chi_2^{\{0\}}\right) \cup \left(\left(\chi_0^{\{1\}} \cap \chi_1^{\{1\}}\right) \cap x \cap \chi_2^{\{1\}}\right) \\
&= \{(0,0,0),(0,1,0),(1,0,0),(1,1,1)\} \cap x
\end{aligned}$$

This, as expected, is the same as that derived in Example 2.6.1.

2.7 Two-level Minimization of Multi-Valued Functions

We review basic definitions of two-level multi-valued minimization. For a more complete treatment the reader is referred to [114].

Definition 2.7.1 *Let $p_i, i = 1, \ldots, n$ be positive integers. Define $P_i = \{0, \ldots, p_i - 1\}$ for $i = 1, \ldots, n$, and $B = \{0, 1, *\}$. A multiple-valued input, binary-valued output function, f, is a function*

$$f : P_1 \times P_2 \times \cdots \times P_n \rightarrow B$$

The function f has n multiple-valued inputs. Each input variable i assumes one of the p_i values in P_i. The value $* \in B$ is used when the function value is unspecified (i.e., it is allowed to be either 0 or 1).

An n-input, m-output switching function can be represented by a multiple-valued function of $n+1$ variables where $p_i = 2$ for $i = 1, \ldots, n$, and $p_{n+1} = m$. The minimization problem for multiple-output functions is equivalent to the minimization of a multiple-valued function of this form [119].

Definition 2.7.2 *Let X_i be a variable taking a value from the set P_i, and let S_i be a subset of P_i. $X_i^{S_i}$ represents the Boolean function*

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

$X_i^{S_i}$ is called a **literal** of variable X_i. If $S_i \equiv \emptyset$, then the value of the literal is always 0, and the literal is called empty. If $S_i \equiv P_i$, then the value of the literal is always 1, and the literal is called full.

Two-valued (or **binary**) functions are a special case of multi-valued functions where $P_i = \{0, 1\}$ for $i = 1, \ldots, n$. In the case of a two-valued single-output function, some notational simplification is then possible. A cube may be written as a vector on a set of variables with each position representing a distinct variable. The values taken by each position are 1, 0 or 2 (same as $-$, don't-care), signifying the true form, negated form or both of the variable corresponding to that position. A **minterm** is a cube with only 0 and 1 entries. Cubes can be classified based on the number of 2 entries. A cube with k entries or bits which take the value 2 is called a k**-cube**. A minterm thus is a 0**-cube**.

A **product term** (or **cube**) is a Boolean product (AND) of literals. A minterm or 0-cube is a product-term in which the sets of values of all literals are singletons. If a product term evaluates to 1 for a given minterm, the product term is said to **contain** (or **cover**) the minterm.

A **sum-of-products** (or **cover**) is a Boolean sum (OR) of product terms. If any product term in the sum-of-products evaluates to 1 for a given minterm, then the sum-of-products is said to contain the minterm. If a literal in a product-term is empty, the product term contains no minterms, and is called the null product (written $\emptyset$). The **on-set** of a function is the set of minterms for which the function value is 1. Likewise, the **off-set** is the set of minterms for which the function value is 0, and the **DC-set** is the set of minterms for which the function value is unspecified.

In the definitions which follow, $S = X_1^{S_1} X_2^{S_2} \cdots X_n^{S_n}$ and $T = X_1^{T_1} X_2^{T_2} \cdots X_n^{T_n}$ represent product terms.

The product term S contains the product term T ($T \subset S$) if $T_i \subset S_i$ for $i = 1 \ldots n$. The **complement** of the literal $X_i^{S_i}$ (written $\overline{X_i^{S_i}}$) is the literal $X_i^{P_i - S_i}$. The complement of the product

term S ($\overline{S}$) is the sum-of-products $\bigcup_{i=1}^{n} \overline{X_i^{S_i}}$.

The **intersection** of product terms S and T ($S \cap T$) is the product term

$$X_1^{S_1 \cap T_1} X_2^{S_2 \cap T_2} \cdots X_n^{S_n \cap T_n}.$$

If $S_i \cap T_i = \emptyset$ for some i, then $S \cap T = \emptyset$ and S and T are said to be disjoint. The intersection of covers F and G is the union of $f \cap g$ for all $f \in F$ and $g \in G$. The **distance** between S and T ($distance(S,T)$) is $|\{i | S_i \cap T_i = \emptyset\}|$.

The **consensus** of S and T ($consensus(S,T)$) is the sum-of-products

$$\bigcup_{i=1}^{n} X_1^{S_1 \cap T_1} \cdots X_i^{S_i \cup T_i} \cdots X_n^{S_n \cap T_n}.$$

If $distance(S,T) \geq 2$ then $consensus(S,T) = \emptyset$. If $distance(S,T) = 1$ and $S_i \cap T_i = \emptyset$, then $consensus(S,T)$ is the single product term $X_1^{S_1 \cap T_1} \cdots X_i^{S_i \cup T_i} \cdots X_n^{S_n \cap T_n}$. If $distance(S,T) = 0$ then $consensus(S,T)$ is a cover of n terms. If the consensus of S and T is nonempty, it is the set of maximal product terms (ordered by containment) which are contained in $S \cup T$ and which contain minterms of both S and T. The consensus of two covers F and G is the union of $consensus(f,g)$ for all $f \in F$ and $g \in G$.

The **cofactor** (or **cube restriction**) of S with respect to T (S_T) is empty if S and T are disjoint. Otherwise, the cofactor is the product term

$$X_1^{S_1 \cup \overline{T_1}} \cdots X_2^{S_2 \cup \overline{T_2}} \cdots X_n^{S_n \cup \overline{T_n}}.$$

The cofactor of a cover F with respect to a product term S is the union of f_S for all $f \in F$.

An **implicant** of a function is a product term which does not contain any minterm in the off-set of the function. A **prime implicant** of a function is an implicant which is not contained by any other implicant of the function. An **essential prime implicant** is a prime implicant which contains a minterm which is not covered by any other prime implicant.

The product term S can be represented in **positional cube notation** as a binary vector in the following form:

$$c_1^0 c_1^1 ... c_1^{p_1 - 1} - c_2^0 c_2^1 ... c_2^{p_2 - 1} - c_n^0 c_n^1 ... c_n^{p_n - 1}$$

where $c_i^j = 0$ if $j \notin S_i$, and $c_j^i = 1$ if $j \in S_i$. In other words, a symbolic variable that can take values from a set of cardinality n is represented in positional cube notation by an n-bit vector to denote a literal of that variable such that each position in the vector corresponds to a specific element of the set. A 1 in a position in the vector signifies the presence of an element in the literal

while a 0 signifies the absence. This method of representation is commonly known as **one-hot**. By complementing the n-bit vector that represents the one-hot encoding of a symbolic variable, one gets a representation called **complemented one-hot**.

Up to now we have introduced multi-valued inputs and binary outputs functions, represented by multiple-valued functions where the set of binary outputs is treated as one more multi-valued input variable. Positional cube notation allows also to represent any function with multi-valued input and multi-valued output variables. This is commonly done in programs like ESPRESSO-MV, when a function with symbolic inputs and outpus (e.g., an FSM) is 1-hot encoded and then minimized. But the minimization problem for functions with multi-valued input and output variables is not known to be equivalent to the minimization of a multiple-valued function of this form. After 1-hot encoding the onsets of the minterms (values) of a symbolic output are treated as disjoint and so are minimized separately. To handle the minimization problem of functions with multi-valued input and multi-valued output variables the concept of generalized prime implicants will be introduced later.

2.8 Multi-level Minimization of Multi-Valued Functions

We now introduce multi-level networks with multi-valued input variables. By convention, in this section we will we use upper case letters for multi-valued variables and lower-case letters for binary-valued variables.

A *sum-of-products* (SOP) is a Boolean sum (OR) of product terms. For example: $X^{\{0,1\}}y_1y_2$ is a cube and $X^{\{0,1\}}y_1y_2 + X^{\{3\}}y_2y_3$ is an SOP. A function $\mathbf{f}$ may be represented by an SOP expression f. In addition $\mathbf{f}$ may be represented as a factored form. A factored form is defined recursively as follows.

Definition 2.8.1 *An* SOP *expression is a factored form. A sum of two factored forms is a factored form. A product of two factored forms is a factored form.*

$X^{\{0,1,3\}}y_2(X^{\{0,1\}}y_1 + X^{\{3\}}y_3)$ is a factored form for the SOP expression given above.

A logic circuit with a multiple-valued input is represented as an MV-network. An MV-*network* η, is a directed acyclic graph (DAG) such that for each node n_i in η there is associated a binary-valued, MV input function $\mathbf{f_i}$, expressed in SOP form, and a binary-valued variable y_i which represents the output of this node. There is an edge from n_i to n_j in η if f_j explicitly depends on y_i. Further, some of the variables in η may be classified as *primary inputs* or *primary outputs*.

These are the inputs and outputs (respectively) of the MV-network. The MV-network is an extension of the well-known Boolean network [12] to permit MV input variables; in fact the latter reduces to the former when all variables have binary values. Since each node in the network has a binary-valued output, the non-binary(MV) inputs to any node must be primary inputs to the network. The MV-network computes logical functions in the natural way. Each node in the DAG computes some function, the result of which is used in all the nodes to which an edge exists from this node.

The cost of a boolean network is typically estimated as the sum over all nodes of the number of literals in a minimum (i.e. one with a least number of literals) factored form of the node function. This cost estimation has a good correlation with the cost of an implementation of the network in various technologies, e.g. standard cells or CMOS gate matrix.

2.9 Multiple-Valued Relations

Definition 2.9.1 *A* **multiple-valued relation** R *is a subset of* $D \times B^m$. D *is called the* **input set** *of* R *and is the Cartesian product of* n *sets* $D_1 \times \cdots \times D_n$*, where* $D_i = \{0, \ldots, P_i - 1\}$ *and* P_i *is a positive integer.* D_i *provides the set of values that the* i*-th variable of* D *can assume.* B^m *designates a Boolean space spanned by* m *variables, each of which can assume either* 0 *or* 1. B^m *is called the* **output set** *of* R*. If* P_i *is* 2 *for all* i*'s, then* R *is called a* **Boolean relation***. The variables of the input set and the output set are called the* **input variables** *and the* **output variables** *respectively.* R *is* **well-defined** *if for every* $\mathbf{x} \in D$*, there exists* $\mathbf{y} \in B^m$ *such that* $(\mathbf{x}, \mathbf{y}) \in R$.

We represent a relation R by its characteristic function $R : D \times B^m \rightarrow B$ such that $R(\mathbf{x}, \mathbf{y}) = 1$ if and only if $(\mathbf{x}, \mathbf{y}) \in R$. In the implementation, we represent a characteristic function by using a multi-valued decision diagram (MDD, see [64, 136]). An MDD is a data structure to represent a function with multiple-valued input variables and a single binary output, which employs a BDD [16] as the internal data structure.

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

Definition 2.9.2 *For a given relation* R *and a subset* $A \subseteq D$*, the* **image** *of* A *by* R *is a set of minterms* $\mathbf{y} \in B^m$ *for which there exists a minterm* $\mathbf{x} \in A$ *such that* $(\mathbf{x}, \mathbf{y}) \in R$*, i.e.* $\{\mathbf{y} \mid \exists \mathbf{x} \in A : (\mathbf{x}, \mathbf{y}) \in R\}$.

The image is denoted by $r(A)$. $r(A)$ may be empty.

Definition 2.9.3 *For a given relation $R \subseteq D \times B^m$, a multiple-valued function $f : D \to B^m$ is* **compatible** *with R, denoted by $f \prec R$, if for every minterm* **x** $\in D$, $f(\mathbf{x}) \in r(\mathbf{x})$. *Otherwise f is incompatible with R. Clearly, $f \prec R$ exists if and only if R is well-defined.*

2.10 Binary Decision Diagrams

Definition 2.10.1 *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 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 2.10.2 *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 2.10.3 *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 2.10.4 *A* **reduced ordered binary decision diagram** *(ROBDD) is an BDD which is both reduced and ordered.*

Chapter 3

Complexity Issues

3.1 Computational Complexity

In this section we will present some results on the computational complexity of state assignment for minimum area. We refer to [46, 104, 9] as standard references on computational complexity and the theory of NP-completness in particular. Computational complexity of logic optimization problems has been discussed in [69], from which we will draw results.

An instance of a problem is encoded as a string (or word) of a language. So the solution of a problem is equivalent to decide whether a given string (an instance of the problem) is in that language or not. A decision algorithm is usually given by means of a Turing machine, that is a universal computational device. The game is to find out how much of space and time resources a Turing machine must use to recognize words in the language. The resources taken by a Turing machine are polynomially related to those of the other commonly used computational mechanisms (for instance a C program running on a Von Neumann computer). Of course there are also insolvable problems, but they are not of interest here.

Let $L \subseteq \Sigma^*$ be a language. The complement of L, denoted $\overline{L}$, is the language $\Sigma^* - L$, i.e., the set of all strings in the appropriate alphabet that are not in L. The complement of a decision problem A (sometimes denoted A COMPLEMENT), is the decision problem whose answer is "yes" whenever A answers "no" and viceversa.

Example 3.1.1 *SAT is the problem of deciding if a given Boolean expression in conjunctive normal form (CNF) has a satisfying assignment. Given a reasonable rule to encode any CNF expression, the language SAT will contain all strings that encode instances of CNF expressions that are satisfiable.*

Then given any string one can construct an algorithm that first checks whether the string encodes a CNF expression and then finds if a satisfying assignment exists. SAT COMPLEMENT is the problem: given a CNF, is it unsatisfiable ? Strictly speaking the languages corresponding to the problems SAT and SAT COMPLEMENT are not the complements of one another, since their union is not Σ^* but rather the set of all strings that encode CNF's.

A set of languages (representing decision problems) recognizable with the same amount of computational resources and/or the same computational mode (for instance, deterministic vs. non-deterministic) are said to be a complexity class. For instance, P is the class of problems for which polynomial time is sufficient. NP is the class of problems that can be solved by a non-deterministic Turing machine in polynomial time. Another characterization of NP is the class of problems whose solution can be verified in polynomial time. As an example SAT is in NP because it takes linear time to verify if an assignment satisfies a CNF expression, but it seems hard to decide whether a satisfying assignment exists and it is not known whether SAT is in P. If NP is the class of problems that have succinct certificates, $co-NP$ contains those problems that have succinct disqualifications. That is a "no" instance of a problem in $co-NP$ has a short proof if its being a "no" instance; and only "no instances" have a short proof. Alternatively, $co-NP$ is the class of problems whose complement is in NP. In general for any complexity class C, $co-C$ denotes the class $\{\overline{L} : L \in C\}$.

Example 3.1.2 *VALIDITY of Boolean expressions is in $co-NP$. We are a given a Boolean expression ϕ, and we are asked whether it is valid, i.e. satisfiable by all truth assignments. If ϕ is not a valid formula, then it can be disqualified very succinctly, by providing a truth assignment that does not satisfy it. No valid formula has such a disqualification. Also VALIDITY of restricted Boolean expressions in sum-of-product forms (SOP) is in $co-NP$. VALIDITY is also called TAUTOLOGY.*

Problems as hard as any in NP are called NP-hard. Problem A is at least as hard as problem B if B reduces to A. B reduces to A if there is a transformation R that, for every input x of B, produces an input $R(x)$ of A, such that the answer to $R(x)$ as input to A is the same as the answer to x as input to B. In other words, to solve B on input x it is sufficient to compute $R(x)$ and solve A on $R(x)$. Of course R should be reasonably simple to compute: often one requires that R is computable by a deterministic Turing machine in space $\mathcal{O}(\log n)$. More simply one wants a reduction R that can be computed in polynomial time.

A fundamental result due to Cook [46] shows that SAT is as hard as any problem in NP, i.e. knowing how to solve SAT efficiently (in polynomial time) would enable us to solve efficiently any other problem in NP. By transitivity, to show that a problem is NP-hard it is enough to show that it is as hard as SAT. Any language L in $co-NP$ is reducible to VALIDITY. Indeed, if L is in $co-NP$, then $\overline{L}$ is in NP, and thus there is a reduction R from $\overline{L}$ to SAT. For any string x, $x \in \overline{L}$ iff $R(x)$ is satisfiable. The reduction from L to VALIDITY is $R'(x) = \neg R$.

NP-complete problems are the NP-hard problems that are also in NP. In general if C is complexity class and L is a language in C, L is C-complete if any language $L' \in C$ can be reduced to L. No NP-complete problem is known to be in P, but no super-polynomial lower bound is known either.

A problem as hard as any in $co-NP$ is called $co-NP$-hard, which means that its complement is NP-hard, i.e. as hard as any problem in NP. $co-NP$-complete problems are the $co-NP$-hard problems that are also in $co-NP$, i.e. whose complementary problem is NP-complete. In general if L is NP-complete, then its complement $\overline{L} = \Sigma^* - L$ is $co-NP$-complete.

It is not known whether $co-NP$-complete are harder than NP-complete ones. Still $co-NP$-complete seem harder than NP-complete ones: e.g., deciding VALIDITY intuitively requires checking whether all assignments satisfy a Boolean expression, while SAT can be answered as soon as a satisfying assignment is found. So, unless a theoretical breakthrough proves that the two classes coincide, it is useful to classify precisely a problem as belonging into one vs. the other, as it is recommended in [69], reacting against sloppy statements in the literature on algorithms for computer-aided design.

Beyond P and NP there is a whole world of complexity classes. We are going to introduce the rudiments of the polynomial hierarchy because they are needed to classify correctly some versions of state assignment.

Say that a Turing machine is equipped with an oracle, when it has available a subroutine tha charges one unit of computation to give an answer, e.g., an oracle could be a subroutine that decides whether a word is in SAT. For instance, one names as P^{SAT} the class of problems that can be solved in polynomial time by a deterministic Turing machine augmented with a SAT oracle. In general, if C is any complexity class, C^A is the class of languages decided by machines as those that decide the languages of C, only that they are also equipped with oracle A.

Example 3.1.3 *Let $\langle E, k \rangle$ be an instance of the problem EQUIVALENT FORMULAS, which consists of deciding whether boolean expression E (we will use Boolean expression and Boolean*

formula as synonyms) admits an equivalent formula E' including, at most, k occurrences of literals, where two Boolean formulas are equivalent if for any assignment of values E is satisfied iff E' is satisfied.

Theorem 3.1.1 *SATISFIABILITY can be solved in polynomial time by a deterministic Turing machine with oracle EQUIVALENT FORMULAS.*

Proof: There are only two types of formulas equivalent to a formula including 0 occurrences of literals, that is, a formula consisting only of Boolean constants: those equivalent to **true**, also called *tautologies*, that are satisfied by all possible assignments of values, and those equivalent to **false** which cannot be satisfied by any assignment of values.

Let E be a formula in CNF form. To decide whether E is satisfiable it is sufficient to check first whether E is a tautology. If so, E is satisfiable; otherwise, we have only to check whether E is equivalent to a formula containing 0 occurrences of literals. If this is the case, E is not satisfiable, otherwise it is satisfiable. The first check can be done in polynomial time on a CNF; the second can also be done in polynomial time by querying the oracle EQUIVALENT FORMULAS with the word $< E, 0 >$. If the oracle answers positively, E is not satisfiable, otherwise it is satisfiable. ■

No construction is known in the opposite direction: no deterministic Turing machine having an NP-complete language as oracle and deciding EQUIVALENT FORMULAS in polynomial time has been found. It is however possible to define a nondeterministic Turing machine having the above characteristics.

Theorem 3.1.2 *EQUIVALENT FORMULAS can be solved in polynomial time by a nondeterministic Turing machine with oracle SAT.*

Proof: Non-determinism can be exploited to generate all possible formulas E' including, at most, k occurrences of literals and to query the oracle to determine whether E' is not equivalent to E, that is, if $\neg((\neg E' \vee E) \wedge (\neg E \vee E'))$ is satisfiable. If this last formula is not satisfiable, then E' is the required k-literal formula. Conversely, if all k-literal formulas E' are not equivalent to E, then the instance $\langle E, k \rangle$ does not belong to EQUIVALENT FORMULAS . ■

Given a class of languages C define the class P^C as

$$P^C = \bigcup_{L \in C} P^L$$

and NP^C as

$$NP^C = \bigcup_{L \in C} NP^L$$

where P^L and NP^L denote the classes P and NP augmented with oracle L.

It follows that the problem SATISFIABILITY belongs to the class $P^{EQUIVALENT\ FORMULAS}$ while EQUIVALENT FORMULAS belongs to NP^{SAT}. By iterating the previous definitions, one gets the polynomial hierarchy. The polynomial hierarchy is an infinite set $\{\Sigma_k^p, \Pi_k^p, \Delta_k^p : k \geq 0\}$ of classes of languages such that

1. $\Sigma_0^p = \Pi_0^p, \Delta_0^p = P$.
2. $\Sigma_{k+1}^p = NP^{\Sigma_k^p}, \Pi_{k+1}^p = co\Sigma_{k+1}^p$ and $\Delta_{k+1}^p = P^{\Sigma_k^p}$ with $k \geq 0$.

The infinite union of all Σ_k^p's (or of all Π_k^p's or of all Δ_k^p) is denoted as PH.

An alternate characterization of the polynomial hierarchy is as follows.

Theorem 3.1.3 *For each $k \geq 0$, a language L belongs to Σ_k^p iff a language $A \in P$ and a polynomial p exist such that*

$$x \in L \leftrightarrow (\exists y_1)(\forall y_2) \cdots (Qy_K)[\langle x, y_1, \cdots, y_k \rangle \in A]$$

where $|y_i| \leq p(|x|)$ with $1 \leq i \leq k$ and where the sequence of quantifiers consists of an alternation of existential and universal quantifiers (Q is $\exists$ or $\forall$ if k is odd or even).

Similarly, for each $k \geq 0$, a language L belongs to Π_k^p iff a language $A \in P$ and a polynomial p exist such that

$$x \in L \leftrightarrow (\forall y_1)(\exists y_2) \cdots (Qy_K)[\langle x, y_1, \cdots, y_k \rangle \in A]$$

where $|y_i| \leq p(|x|)$ with $1 \leq i \leq k$ and where the sequence of quantifiers consists of an alternation of universal and existential quantifiers (Q is $\forall$ or $\exists$ if k is odd or even).

A word $\langle x, l \rangle$ belongs to the language associated with EQUIVALENT FORMULAS iff a formula y_1 exists such that, for all possible possible assignments of values y_2, $\langle \langle x, k \rangle, y_1, y_2 \rangle \in A$ holds, where the language $A \in P$ is defined as: $\langle \langle x, k \rangle, y_1, y_2 \rangle \in A$ iff y_2 is an assignment of values which satisfies the formula $(\neg x \vee y_1) \wedge (\neg y_1 \vee x)$ where y_1 denotes a formula which includes, at most, k occurrences of literals. So EQUIVALENT FORMULAS is in Σ_2^p.

Very few interesting problems have been shown to be complete with respect to a given level of the polynomial hierarchy. For example, it is not known whether EQUIVALENT FORMULAS is Σ_2^p-complete.

Let E be a Boolean formula built on a set of Boolean variables $\cup_i^k X_i$ where $X_i = \{x_{ij} : 1 \leq j \leq m_i\}$ with m_i positive integer. The problem k-QBF consists of deciding whether the

formula

$$(\exists)(\forall)\cdots(QX_k)[E(X_1,\cdots,X_k)]$$

is true, where $(\exists X_i)$ reads as "there exists an assignment of values to the variables $x_{i_1},\cdots,x_{i_{m_i}}$", and $(\forall X_i)$ reads as "for all assignments of values to the variables $x_{i_1},\cdots,x_{i_{m_i}}$". For all $k \geq 1$, k-QBF is Σ_k^p-complete (and thus k-QBF is one of the hardest problems in Σ_k^p).

Of the classes in the polynomial hierarchy we will need soon Σ_2^p: the class of problems solvable in polynomial time by a non-deterministic Turing machine augmented with an oracle in NP. To strengthen the intuition, let us say that a problem in Σ_2^p is such that not only finding a solution requires the power of non-determinism, but also checking it, while for NP-complete ones only the first task requires the power of non-determinism and the second one is easy. So the fact that a problem is in Σ_2^p and not in a lower complexity class is a valuable information also for the algorithm developer.

Now we have the setting to state and prove the results related to some versions of state assignment problems. State assignment for area has the goal to find an encoded FSM that gives the best two-level or multi-level implementation (another target could be some specific Programmable Gate Array architecture). At the core one must minimize a logic function and produce the best two-level or multi-level representation.

Definition 3.1.1 *Given a representation of a Boolean function F by means of the minterms of the onset and positive integers k and l, MIN-SOP-1 is the problem "is there a SOP representation of F with k or fewer product-terms and l or fewer literals ?".*

Theorem 3.1.4 *MIN-SOP-1 is in NP-complete.*

Proof: MIN-SOP-1 is in NP. A non-deterministic Turing machines can guess a SOP representation G with k or fewer product-terms and l or fewer literals, then it must check whether G is equivalent to F. The check can be done by replacing each product-term in G with the minterms that it covers. Given that F is available as a sum-of-minterms it is easy to verify whether the minterms contained in the representation of G are all and only the minterms that describe F.

MIN-SOP-1 is NP-hard. Let us show that MINIMUM COVER [1] reduces to MIN-SOP-1. Consider an instance of MINIMUM COVER, we suppose for conveniency that the subsets in C

[1] MINIMUM COVER: Given a collection C of subsets of a finite set S and a positive integer $k \leq |C|$, does C contain a cover for S of size $\leq k$, i.e. a subset $C' \subseteq C$ with $|C'| \leq k$ such that every element of S belongs to at least one member of C' ? It shown to be NP-complete in [46].

and the set S are specified by a matrix whose columns are the subsets in C and whose rows are the elements of S, such that entry (i, j) is a 1 iff element i is in subset j and 0 otherwise. Say that there are n rows and m columns. It has been shown by Gimpel [47] that one can build an incompletely specified Boolean function on the set of variables $x_1, x_2, \cdots, x_{m+n}$. Its onset has as many minterms as rows and a generic minterm m_j is given by:

$$m_j = x_1 x_2 \cdots x_{j-1} \neg x_j x_{j+1}, x_{m+n}.$$

Let the primes of the function be as many as the columns of the original table, with a prime P_i given by:

$$P_i = x_{n+i} \prod_{j \in F_i} x_j$$

where $F_i = \{j \mid a_{ij} = 0\}$. The minterms of the dcset are the vertices contained in the primes that are not minterms of the onset. Since minterm m_j is in prime P_i iff entry (i, j) in the table is 1, it follows that an instance of MINIMUM COVER has answer "yes" iff the corresponding instance of MIN-SOP-1 has answer "yes" (same k used in both cases, l is not needed). ■

Definition 3.1.2 *Given a sum-of-products (SOP) representation of a Boolean function F and positive integers k and l, MIN-SOP-2 is the problem "is there a SOP representation of F with k or fewer product-terms and l or fewer literals ?".*

Theorem 3.1.5 *[69] MIN-SOP-2 is in $co - NP$-hard (lower bound).*

Proof: We show that VALIDITY for SOP forms reduces to MIN-SOP-2. We already stated the well-known result that VALIDITY is $co - NP$-hard (precisely it is $co - NP$-complete). Consider an instance of VALIDITY, i.e. a SOP form V. It is easy to check whether V has at least one satisfying assignment, otherwise the answer to VALIDITY of V is no. Suppose that V is satisfiable. Let x by a Boolean variable that does not appear in V and multiply it by the expression V, obtaining $W = V.x$. One can have W in SOP form, by multiplying x by each product of V. In this way W is in SOP form and therefore one can build an instance of MIN-SOP-2 where F is W and $k = l = 1$. It is the case that this instance of MIN-SOP-2 has answer "yes" iff V has an answer "yes" for VALIDITY, because every representation of W must have at least one product term and literal for x, and V can be either a tautology or it must contribute at least one more literal to W (the case that V is not satisfiable has been handled at the beginning). ■

MIN-SOP-2 does not appear to be $co-NP$-easy, since it is not known yet whether having an oracle for any problem in $co-NP$ would enable to solve MIN-SOP-2 in polynomial time. The next theorem shows that MIN-SOP-2 can be solved in polynomial time by a nondeterministic Turing machine with an oracle in NP.

Theorem 3.1.6 *[69] MIN-SOP-2 is in Σ_2^p (upper bound).*

Proof: Consider a nondeterministic Turing machine equipped with SAT as an oracle. Notice that we need a version of SAT for general Boolean expressions (it is still in NP). Non-determinism can be exploited to generate all possible SOP forms, with k or fewer product terms and l or fewer literals, say G is a generic one, and to query the oracle to determine whether, say, G is not equivalent to F, that is, if $\neg((\neg G \vee F) \wedge (\neg F \vee G))$ is satisfiable. If this last formula is not satisfiable, then G is the required POS with $\leq k$ product terms and $\leq l$ literals. Conversely, if no POS with $\leq k$ product terms and $\leq l$ literals is equivalent to E, then the instance $\langle F, k, l \rangle$ does not belong to MIN-SOP-2. ■

The previous results extend easily to the case of minimum SOP forms of encoded FSM's. Notice that a symbolic cover is simply a two-level SOP representation of an FSM. An encoded cover of an FSM is the symbolic cover after syntactic replacement of each state symbol with a code, according to an encoding function e. Basically the previous theorems can be all be rephrased having symbolic covers instead than two-valued covers and adding the requirement that an encoding function be guessed nondeterministically.

First we get an equivalent of MIN-SOP-1. For that we introduce the notion of minterm symbolic cover, that is a symbolic cover of an FSM where each proper input and proper output is a minterm. One can take a symbolic cover and obtain easily a minterm symbolic cover, by replacing each symbolic cube by a set of symbolic cubes which are minterms in the input and output space and add up to the original cube.

Definition 3.1.3 *Given a minterm symbolic representation of an FSM M and positive integers k and l, SA-MIN-SOP-1 is the problem "is there an encoding e that produces an encoded cover M_e that has a SOP representation with k or fewer product-terms and l or fewer literals ?".*

Theorem 3.1.7 *MIN-SA-SOP-1 is in NP-complete.*

Proof: MIN-SA-SOP-1 is in NP-hard. Restrict MIN-SA-SOP-1 to MIN-SOP-1, by noticing that a Boolean function is an FSM with no state variable in its representation.

MIN-SA-SOP-1 is in NP. By nondeterminism one can guess an encoding function e and a minimized encoded SOP form N. M_e is the SOP form obtained from M by replacing syntactically states with codes. Each product-term of M_e is a minterm. We must prove that M_e is equivalent to N. Replace each product-term in N by all minterms that it covers. and call it $N_{minterms}$. Since both M_e and $N_{minterms}$ contain only minterms, their equality can be checked in time polynomial in the original representation. ■

Definition 3.1.4 *Given a symbolic representation of an FSM M and positive integers k and l, SA-MIN-SOP-2 is the problem "is there an encoding e that produces an encoded cover M_e that has a SOP representation with k or fewer product-terms and l or fewer literals ?".*

Theorem 3.1.8 *MIN-SA-SOP-2 is $co-NP$-hard (lower bound).*

Proof: Restrict MIN-SA-SOP-2 to MIN-SOP-2, by noticing that a Boolean function is an FSM with no state variable in its representation. ■

Theorem 3.1.9 *MIN-SA-SOP-2 is in Σ_2^p (upper bound).*

Proof: As in the proof that MIN-SOP-2 is in Σ_2^p. Thanks to nondeterminism one guesses an encoding e and a minimized encoded SOP form N. M_e is the SOP form obtained from M by replacing syntactically states with codes. We must prove that the SOP form M_e is equivalent to the SOP form N. This is exactly what was done with SAT as an oracle for MIN-SOP-2. ■

This classification lumps together, for instance, MIN-SOP-2 and SA-MIN-SOP-2, and therefore is not satisfactory with respect to the experimental fact that the latter problem is much harder that the former. This is in part due to the lack of fine tuning of the complexity classes of the polynomial hierarchy. It would be worthy to see if a finer classification can be achieved looking into approximation complexity classes [46, 104, 9].

Similar results could be obtained for other optimization objectives, like minimum number of literals of multi-level implementations [69]. Also the introduction of don't care conditions in the original representations, allowing for choices in the encoded implementations, can be handled with minor variant of the previous techniques.

3.2 Counting State Assignments

Suppose that there are v symbols to encode and 2^n codes, with $n \geq \lceil \log v \rceil$. There are $\begin{pmatrix} 2^n \\ v \end{pmatrix} v!$ possible assignments, since there are $\begin{pmatrix} 2^n \\ v \end{pmatrix}$ ways to select v distinct state codes and $v!$ ways to permute them.

Suppose that a state assignment is given by a matrix, whose i-th column carries the i-th encoding bit of every symbol and each row is the code of a symbol. One can introduce an equivalence relation on the set of state encodings, lumping in the same equivalence class all state encodings that produce the "same" encoded representation. The "same" means that the encoded representation are not intrinsically different. For instance if we permute columns of an encoding it is intuitively obvious that the encoded Boolean function does not change, except that variables have been renamed. What happens if we complement a column of an encoding ? In case of state assignment things depend on the chosen memory element. If one uses D flip-flops, then the size of a minimal encoded representation is strongly affected by the chosen phase. Instead, with other types of flip-flops, state encodings that differ only by complementation of some columns can be considered equivalent.

The number of equivalence classes of state assignments, where equivalence is by permutation and complementation of columns, and $2^{n-1} < v \leq 2^n$, was computed in [88] as:

$$A(v) = \frac{(2^n - 1)!}{(2^n - v)!n!}.$$

The number of equivalence classes of state assignments, where equivalence is only by permutation of columns, and $2^{n-1} < v \leq 2^n$, was computed in [149] as:

$$B(v) = \frac{(2^n)!}{(2^n - v)!n!}.$$

The fact that $A(v)$ is correct for SR, JK and T flip-flops was pointed out first in [110]. This does not extend to D flip-flops, for which $B(v)$ is the correct formula, because in a D flip-flop the excitation expression for the complemented state variable is the complement of the expression for the uncomplemented state variable.

The formulas for the general case, i.e., where v is not restricted to $2^{n-1} < v \leq 2^n$, were published by Harrison and Parchman ([109, 106]). They introduced the definition of degenerate state assignments, i.e., those where a column is constant or two or more columns are equal. Let the following definitions hold:

1. $T(n,v)$ is the number of nonequivalent state assignments with respect to permutations of the columns;

2. $R(n,v)$ is the number of non degenerate state assignments with respect to permutations of the columns;

3. $T^*(n,v)$ is the number of nonequivalent state assignments with respect to permutations and complementations of the columns;

4. $R^*(n,v)$ is the number of non degenerate state assignments with respect to permutations and complementations of the columns.

Then the following identities hold, where $s(v,j)$ are the Stirling numbers of the first kind:

1.
$$T(n,v) = \sum_{j=1}^{v} \left(\begin{array}{c} n+2^j-1 \\ n \end{array} \right) s(v,j),$$

2.
$$T^*(n,v) = \sum_{j=1}^{v} \left(\begin{array}{c} n+2^{j-1}-1 \\ n \end{array} \right) s(v,j),$$

3.
$$R(n,v) = \sum_{j=1}^{v} \left(\begin{array}{c} 2^j-2 \\ n \end{array} \right) s(v,j)$$

4.
$$R^*(n,v) = \sum_{j=1}^{v} \left(\begin{array}{c} 2^{j-1}-1 \\ n \end{array} \right) s(v,j).$$

They have been obtained with non-elementary combinatorial tools.

Chapter 4

Previous and Related Work

4.1 Algorithms for Optimal Encoding

The following optimal encoding problems may be defined:

(A) Optimal encoding of inputs of a logic function. A problem in class A is the optimal assignment of *opcodes* for a microprocessor.

(B) Optimal encoding of outputs of a logic function.

(C) Optimal encoding of both inputs and outputs (or some inputs and some outputs) of a logic function.

(D) Optimal encoding of both inputs and outputs (or some inputs and some outputs) of a logic function, where the encoding of the inputs (or some inputs) is the same as the encoding of the outputs (or some outputs). Encoding the states of a finite state machine (FSM) is a problem in class D since the state variables appear both as input (present state) and output (next state) variables. Another problem in class D is the encoding of the signals connecting two (or more) combinational circuits.

Optimality may be defined in various ways. A common objective is minimum area of the encoded implementation. Each target implementation has a different cost function. The cost of a two-level implementation is the number of product-terms or the area of a programmable logic array (PLA). A commonly used cost of a multi-level implementation is the number of literals of a technology-independent representation of the logic. Another cost function is the complexity of an implementation with field programmable gate arrays (FPGA's). Other optimization objectives

may have to do with power consumption, speed, testability or any combination of the above. In some cases the objective is the satisfaction of a correctness requirement like in state assignment of asynchronous FSM's, where it is required that it be race-free.

Here we will describe various approaches to the problem of optimal encoding from the classical papers of the 60's to the more recent research dating from the mid 80's. We will devote more space to state assignment for minimum area: "state assignment" because in some sense it subsumes the other encoding problems, and "minimum area" because it has been the most studied objective, even though we will survey also contributions for other problems and objectives [1].

4.1.1 Early Contributions

A well-written survey of early literature on state assignment can be found in [75]. Here we will review the key contributions.

Among the first to define input and output encoding problems for combinational networks were [33] and [100]. The former based his theory of input encoding on partitions and set systems. The latter tried to minimize the variable dependency of the output functions and studied the problem of the minimum number of variables required for a good encoding.

In [3] Armstrong described one of the first programmed algorithms to assign internal codes to FSM's, with the goal of obtaining economical realizations of the combinational logic of an FSM. The key idea of the method is to insure that as many vertices as possible in the onset and offset of each next state and output function are pairwise adjacent, so that they can be clustered in subcubes. This may be achieved by examining the rows and columns of the state table for state pairs that can be given adjacent codes and so directly yield simplified Boolean equations for the next state and output variables in terms of the present state and input variables. Various adjacency conditions were derived based on the relations between states. Then the problem was reduced to a graph embedding problem, where a graph represents adjacency relations between the codes of the states, to be preserved by a subgraph isomorphism on the encoding cube. The method was then refined in [2].

As a partial solution to the fact that enumerating all encodings and measuring their cost is not a practical solution, Dolotta and McCluskey in [41] proposed a method based on the concept of codable columns, that are fewer in number than the possible codes, and whose combinations give the actual encodings. The codable columns for a state table are represented by a base matrix that

[1] We must mention that there is a rich literature on state assignment authored by researchers of the former Soviet Union, but we are not in a position to survey it here.

represents the mapping of codable columns into the next state columns; the rows of a base matrix correspond to states and the columns correspond to codable columns. By examining each column mapping in turn and evaluating the result in terms of some minimization objective one determines a best coding. A "scoring procedure" was defined requiring the comparison of each base entry column with the next state entries on a column-by-column basis and allocating a score according to given criteria. Armstrong argued in [2] that the scoring array of [41] could be read in the framework that he proposed.

Story, Harrison et al. [141, 138] proposed algorithms to derive minimal-cost assignments based on the lower-bound approach first described by Davis [33] and extending the technique to find the cost of an assignment proposed by Torng [141]. A set of columns, each composed of a binary element for each row of a partially assigned state table, is derived. From this matrix it is possible to generate all possible distinct state assignments. Input equations for JK bistables are derived from the matrix based on single column partial state assignments (PSA's), and then a minimum number (MN), which represents a lower bound on the cost, is selected for each column. The best state assignment is then found by comparing the sets of MN's with corresponding actual cost numbers for complete encodings consisting of a set of PSA's. Notice that MN is calculated for a particular column by applying the column to the given state table as if the column were a complete state assignment and then deriving the input equations for, say, a JK bistable in the usual way. For instance the expression of the J input of a JK bistable includes all total states (proper input and present state) with present to next state transitions of $0 \rightarrow 1$, and, as a don't care, those of the transitions $1 \rightarrow 0, 1 \rightarrow 1, 0 \rightarrow -, 1 \rightarrow -$. Then the resulting combinational equations must be minimized, in such a way to guarantee a lower bound (notice that we still do not have a complete encoding); this is done by a "modified map" method where any subset of states is considered to be in a subcube in the encoding space, so that the cost of the implementation cannot be decreased in any actual coding. A lower bound for an encoding is the sum of MN's associated with its columns (MNS), because in the cost one does not consider sharing of logic among next state functions. The actual cost number (AN) of an assignment is the number of actual (not lower bound) AND-OR inputs for each bistable input equation minimized separately. The values of MN and AN are compared for each PSA combination to determine the best encoding. The algorithm has an exhaustive nature mitigated by lower bounding. It does not guarantee optimality (contrary to the claim in the title) of an encoded FSM because it disregards multiple-output minimization , since the cost is defined to be the sum of the AND-OR inputs needed to realize each next state transition separately - so it does not account for output encoding - and proper output logic is not

taken into consideration in the optimization procedure. This work was refined and commented by other contributions [101, 102, 103].

Others, as [54, 137, 67], proposed algebraic methods based on the algebra of partitions and on the criterion of reduced dependency. In these methods the state assignment is made in a such a way that each binary variable describing a next state depends on as few variables of the present state as possible. In general reduced dependency has various advantages that included better testability features, but suffers from a weak connection with the logic optimization steps after the encoding.

More recent approaches [124, 125] rely on local optimization rules defined on a control flowgraph. There rules are expressed as constraints on the codes of the internal variables and an encoding algorithm tries to satisfy most of these constraints.

4.1.2 Encoding for Two-level Implementation

Reduction of Input Encoding to Multiple-Valued Minimization

A major step towards an exact solution of encoding problems was the reduction of input encoding to multiple-valued minimization followed by input constraints satisfaction [92]. Efficient algorithms have been devised both for multiple-valued minimization [114] and input constraints satisfaction [92, 145, 116].

Even though state encoding is an input-output encoding problem [2], it can be approximated as an input encoding problem [92] and solved by a two-step process. In the first step, a tabular representation of the FSM is optimized at the symbolic level, e.g., using the program ESPRESSO by Rudell. Multiple-valued minimization generates constraints on the codes that can be assigned to the states. In the second step, states are encoded in such a way that the constraints are satisfied. The goal in deriving constraints from the minimized symbolic cover is to encode the states in such a way that the cardinality of the resulting two-level Boolean implementation is no greater than the cardinality of the minimized symbolic cover. A sufficient condition to preserve the cardinality of the minimized symbolic cover after encoding is to ensure that each multiple-valued input literal in the minimized symbolic cover translates into a single cube in the Boolean domain. In other words, given a multiple-valued literal, the states present in it should form a face (in the Boolean encoding space) that does not include the states absent from the same multiple-valued literal. Such constraints are called **face** or **input constraints** and finding codes that satisfy them is the **face**

[2] Moreover the same symbols appear both in the input and in the output part.

0 s1 s2 1	
1 s1 s4 0	
0 s2 s2 1	**0 (s1, s2, s4) s2 1**
1 s2 s1 1	**1 (s2, s4) s1 1**
0 s3 s3 0	**1 (s1, s3) s4 0**
1 s3 s4 0	**0 (s3) s3 0**
0 s4 s2 1	
1 s4 s1 1	
(a)	**(b)**

Figure 4.1: Original and minimized symbolic cover of an FSM

embedding problem.

An example from [4] of a tabular representation of an FSM is shown in Figure 4.1(a). Multiple-valued minimization of this FSM - where the states are the possible values of a multiple-valued variable - yields the cover shown in Figure 4.1(b). This can be done by representing the symbolic variables using the positional cube notation [139, 114], and then invoking a multiple-valued minimizer, such as [114]. The minimized cover is output disjoint and all the reduction in the cardinality of the symbolic cover is due to the input part, i.e. due to the fact that some present states fan out to the same next state for certain primary inputs. To get a compatible boolean representation, one must assign each of the groups of present states obtained by multi-valued minimization, to subcubes of a boolean k-cube, for a minimum k, in a way that each subcube contains all and only all the codes of the states included in the face constraint. Codes satisfying the face-embedding constraints implied by the minimized symbolic cover of Figure 4.1(b) are shown in Figure 4.2(a). Three binary variables are necessary and sufficient to satisfy the face-embedding constraints. Figure 4.2(b) shows these codes in the Boolean 3-space. The cover obtained after substitution of the state codes in the symbolic cover and a successive two-level Boolean minimization is shown in Figure 4.3.

It is worth mentioning that the face constraints obtained through straightforward symbolic minimization are sufficient, but not necessary to find a two-valued implementation matching the upper bound of the multi-valued minimized cover. As it was already pointed out in [91], for each implicant of a minimal (or minimum) multi-valued cover, one can compute an *expanded implicant*, whose literals have maximal (maximum) cardinality and a *reduced implicant* whose literals have minimal (minimum) cardinality. By bit-wise comparing the corresponding expanded and reduced implicant, one gets *don't cares* in the input constraint, namely, in the bit positions where the expanded implicant has a 1 and the reduced implicant has a 0. The face embedding problem

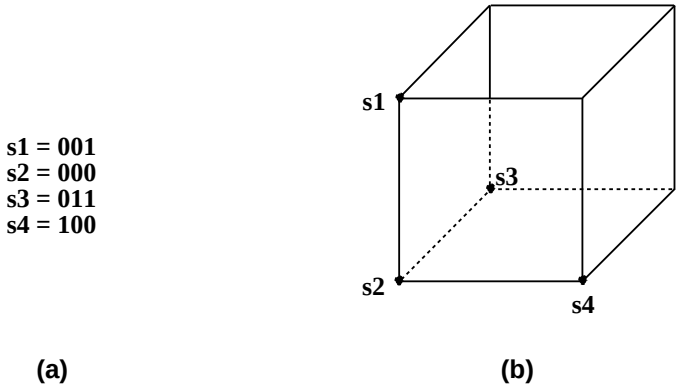


Figure 4.2: Codes satisfying input constraints

```
0 −1−  011 0
1 −−1  100 0
1 −−0  001 1
0 −0−  000 1
```

Figure 4.3: Two-level implementation of encoded FSM

with *don't cares* becomes one of finding a cube of minimum dimension k, where, for every face constraint, one can assign the states asserted to vertices of a subcube that does not include any state not asserted, whereas the *don't care* states can be put inside or outside of that subcube. One can build examples where the presence of *don't cares* allows to satisfy the input constraints in a cube of smaller dimension, than it would be possible otherwise. Consider the state table of an FSM and its 1-hot encoded representation shown in Figure 4.4. In Figure 4.5 the expanded and reduced minimized multi-valued covers of the FSM of Figure 4.4 are shown. Figure 4.6 shows the expanded and reduced present state literals of the same FSM and the *don't care* face constraints.

A novel observation is that by choosing another minimum multi-valued cover, a different set of face embedding constraints (with don't cares, if any) could be generated and they might be satisfiable with a smaller k than the one required by the previous minimum cover.

Symbolic Minimization

Any encoding problem, where the symbolic variables appear only in the input part, can be solved by setting up a multiple-valued minimization followed by satisfaction of the induced face constraints. However, the problem of state assignment of FMS's is only partially solved by this scheme, because the encoding of the symbolic output variables is not taken into account (e.g., the

00	st0	st0	0		00	10000000000	10000000000	0
10	st0	st1	-		10	10000000000	01000000000	-
01	st0	st2	-		01	10000000000	00100000000	-
10	st1	st1	1		10	01000000000	01000000000	1
00	st1	st3	1		00	01000000000	00010000000	1
11	st1	st5	1		11	01000000000	00000100000	1
01	st2	st2	1		01	00100000000	00100000000	1
00	st2	st7	1		00	00100000000	00000001000	1
11	st2	st9	1		11	00100000000	00000000010	1
00	st3	st3	1		00	00010000000	00010000000	1
01	st3	st4	1		01	00010000000	00001000000	1
01	st4	st4	1		01	00001000000	00001000000	1
00	st4	st0	-		00	00001000000	10000000000	-
11	st5	st5	1		11	00000100000	00000100000	1
01	st5	st6	1		01	00000100000	00000010000	1
01	st6	st6	1		01	00000010000	00000010000	1
00	st6	st0	-		00	00000010000	10000000000	-
00	st7	st7	1		00	00000001000	00000001000	1
10	st7	st8	1		10	00000001000	00000000100	1
10	st8	st8	1		10	00000000100	00000000100	1
00	st8	st0	-		00	00000000100	10000000000	-
11	st9	st9	1		11	00000000010	00000000010	1
10	st9	st10	1		10	00000000010	00000000001	1
10	st10	st10	1		10	00000000001	00000000001	1
00	st10	st0	-		00	00000000001	10000000000	-

Figure 4.4: Initial and 1-hot encoded covers of FSM-1

01	01010111011	000000010001	01	00010001000	000000010001
01	01001110111	000000001001	01	00001000100	000000001001
10	00111101110	000000000101	10	00000100010	000000000101
10	00111011101	000000000011	10	00000010001	000000000011
00	01011010000	000100000001	00	01010000000	000100000001
11	11011101111	000010000001	11	01001000000	000010000001
00	00101110000	000001000001	00	00100100000	000001000001
11	10110111111	000000100001	11	00100010000	000000100001
10	11111001100	010000000001	10	11000000000	010000000001
01	11100110011	001000000001	01	10100000000	001000000001
00	10001011111	100000000000	00	10000001111	100000000000

Figure 4.5: Expanded and reduced minimized covers of FSM-1

01010111011	00010001000	0-010--10--
01001110111	00001000100	0-001--01--
00111101110	00000100010	00---10--10
00111011101	00000010001	00---01--01
01011010000	01010000000	0101-0-0000
11011101111	01001000000	-10-1-0----
00101110000	00100100000	0010-1-0000
10110111111	00100010000	-01-0-1----
11111001100	11000000000	11---00--00
11100110011	10100000000	1-100--00--
10001011111	10000001111	1000-0-1111

Figure 4.6: Expanded and reduced implicants and don’t care face constraints of FSM-1

next state variable). Simple multiple-valued minimization disjointly minimizes each of the on-sets of the symbolic output functions, and therefore disregards the sharing among the different output functions taking place when they are implemented by two-valued logic. Sharing of logic is crucial to obtain minimum encoded two-level implementations.

Therefore extensions of multiple-valued minimization have been proposed in [91, 147]. These extensions replace a single multiple-valued minimization of the whole symbolic cover by a sequence of minimization operations on parts of the symbolic cover in such a way as to recognize sharing of logic among next states, if some constraints on their codes are satisfied. These extensions of multiple-valued minimization have been called symbolic minimization. In [91, 147] symbolic minimization was introduced to exploit bit-wise dominance relations between the binary codes assigned to different values of a symbolic output variable. The fact is that the input cubes of a dominating code can be used as don't cares for covering the input cubes of a dominated code. The core of the approach is a procedure to find useful **dominance** (called also **covering**) constraints between the codes of output states. The translation of a cover obtained by symbolic minimization into a compatible boolean representation defines simultaneously a face embedding problem and an output dominance satisfaction problem. Any output encoding problem can be solved by symbolic minimization. Symbolic minimization was applied also in [115], where a particular form of PLA partitioning is examined, by which the outputs are encoded to create a reduced PLA that is cascaded with a decoder.

However, to mimic the full power of two-valued logic minimization, another fact must be taken into account. When the code of a symbolic output is the bit-wise disjunction of the codes of two or more other symbolic outputs, the on-set of the former can be minimized by using the on-sets of the latter outputs, by redistributing the implementation of some cubes. An extended scheme of symbolic minimization can therefore be defined to find useful dominance and **disjunctive** relations between the codes of the symbolic outputs. This will be thoroughly investigated in a later chapter of the thesis. The translation of a cover obtained by extended symbolic minimization into a compatible boolean representation induces a face embedding, output dominance and output disjunction satisfaction problem.

A variety of other applications may also generate similar constraints satisfaction problems, as in the case of synthesis for sequential testability [35], and optimal re-encoding and decomposition of PLA's [40, 21, 122, 120, 119, 121, 123]. Given a PLA, it is possible to group the inputs into pairs and replace the input buffers with two-bit decoders to yield a bit-paired PLA with the same number of columns and no more product-terms than the original PLA. In a more general case, a

single PLA is decomposed into two levels of cascaded PLA's. A subset of inputs is selected such that the cardinality of the multiple-valued cover, produced by representing all combinations of these inputs as different values of a single multiple-valued variable, is smaller than the cardinality of the original binary cover. The encoding problem consists of finding the codes of the signals between the PLA's, so that the constraints imposed by the multiple-valued cover are satisfied. This problem is usually approximated as an input encoding problem [40, 21], but in its generality is an input-output encoding problem referred in [39] as four-level Boolean minimization.

Exact Encoding with Generalized Prime Implicants

An exact procedure for output encoding has been reported in [39]. A notion of generalized prime implicants (GPI's), as an extension of prime implicants defined in [87], is introduced, and appropriate rules of cancellation are given. Each GPI carries a tag with some output symbols. If a GPI is accepted in a cover, it asserts as output the intersection (bit-wise *and*) of the codes of the symbols in the tag. To maintain functionality, the coded output asserted by each minterm must be equal to the bit-wise *or* of the outputs asserted by each selected GPI covering that minterm. Given a selection of GPI's, each minterm yields a boolean equation constraining the codes of the symbolic values. If an encoding can be found that satisfies the system of boolean equations, then the selection of GPI's is encodable. We will devote some later chapters to GPI's and explain in detail the notion of encodabilities of GPI's. Given all the GPI's, one must select a minimum subset of them that covers all the *minterms* and forms an encodable cover. This can be achieved by solving repeated covering problems that return minimum covers of increasing cardinality, until an encodable cover is found, i.e. the minimum cover that is also encodable. Figure 4.7 shows output encoding based on GPI's with a simple example taken from [39].

4.1.3 Encoding for Multi-level Implementation

Automatic multi-level logic synthesis programs are now available to the logic designer [52, 12, 8]), since sometimes a PLA implementation of the circuit does not satisfy the area/timing specifications.

A two-level encoding program, such as those described in the previous sections, can often give a good result when multi-level realization is required, but in order to get the maximum advantages from multi-level logic synthesis we need a specialized approach.

This section describes such approaches, giving some information on the relative strengths

1101	out1	1101	(out1)	110-	(out1,out2)	110-	01
1100	out2	1100	(out2)	11-1	(out1,out3)	11-1	10
1111	out3	1111	(out3)	000-	(out4)	000-	00
0000	out4	110-	(out1,out2)				
0001	out4	11-1	(out1,out3)				
		000-	(out4)				

Figure 4.7: Initial cover, GPI's, encodable selection of GPI's and encoded cover of OUT-1

and weaknesses.

There are two main classes of multi-level encoding algorithms:

1. Estimation-based algorithms, that define a distance measure between symbols, such that if "close" symbols are assigned "close" (in terms of Hamming distance) codes it is likely that multi-level synthesis will give good results. Programs such as MUSTANG [36], JEDI [77] and PESTO [57] belong to this class.

2. Synthesis-based algorithms, that use the result of a multi-level optimization on the unencoded or one-hot encoded symbolic cover to drive the encoding process. Programs such as MIS-MV [85] and MUSE [42] belong to this class.

Mustang

MUSTANG uses the state transition graph to assign a weight to each pair of symbols. This weight measures the desirability of giving the two symbols codes that are "as close as possible".

MUSTANG has two distinct algorithms to assign the weights, one of them ("fanout oriented") takes into account the next state symbols, while the other one ("fanin oriented") takes into account the present state symbols. Such a pair of algorithms is common to most multi-level encoding programs, namely MUSTANG, JEDI and MUSE.

The fanout oriented algorithm is as follows:

1. For each output o build a set O^o of the present states where o can be asserted. Each state p in the set has a weight OW_p^o that is equal to the number of times that o is asserted in p.

2. For each next state n build a set N^n of the present states that have n as next state. Again each state p in the set has a weight NW_p^n that is equal to the number of times that n is a next state of p (each cube under which a transition can happen appears as a separate edge in the state transition graph) multiplied by the number of state bits (the number of output bits that the next state symbol generates).

3. For each pair of states k, l let the weight of the edge joining them in the weight graph be $\sum_{n \in S} NW_k^n \times NW_l^n + \sum_{o \in O} OW_k^o \times OW_l^o$.

This algorithm gives a high weight to present state pairs that have a high degree of similarity, measured as the number of common outputs asserted by the pair.

The fanin oriented algorithm (almost symmetric with the previous one) is as follows:

1. For each input i build a set ON^i of the next states that can be reached when i is 1, and a set OFF^i of the next states that can be reached when i is 0. Each state n in ON^i has a weight ONW_n^i that is equal to the number of times that n can be reached when i is 1, and each state n in OFF^i has a weight $OFFW_n^i$ that is equal to the number of times that n can be reached when i is 0.

2. For each present state p build a set P^p of the next states that have p as present state. Again each state n in the set has a weight PW_n^p that is equal to the number of times that n is a next state of p multiplied by the number of state bits.

3. For each pair of states k, l let the weight of the edge joining them in the weight graph be $\sum_{p \in S} PW_k^p \times PW_l^p + \sum_{i \in I} ONW_k^i \times ONW_l^i + OFFW_k^i \times OFFW_l^i$.

This algorithm tries to maximize the number of common cubes in the next state function, since next states that have similar functions will be assigned close codes.

The embedding algorithm identifies clusters of nodes (states) that are joined by maximal weight edges, and greedily assigns to them minimally distant codes. It tries to minimize the sum over all pairs of symbols of the product of the weighted distance among the codes.

The major limitation of MUSTANG is that its heuristics are only distantly related with the final minimization objective. It also models only common cube extraction, among all possible multiple-level optimization operations ([12]).

Jedi

JEDI is aimed at generic symbol encoding rather than at state assignment, and it applies a set of heuristics that is similar to MUSTANG's to define a set of weights among pairs of symbols. Then it uses either a simulated annealing algorithm or a greedy assignment algorithm to perform the embedding.

The proximity of two cubes in a symbolic cover is defined as the number of non-empty literals in the intersection of the cubes. It is the "opposite" of the Hamming distance between two cubes, defined as the number of empty literals in their intersection. For example, cubes $ab\overline{c}$ and $c\overline{d}e$ have proximity 4, because their intersection has four non-empty literals ($a, b, \overline{d}$ and e), and distance 1, because their intersection has an empty literal ($c \cap \overline{c}$).

Each pair of symbols (s_i, s_j) has a weight that is the sum over all pairs of cubes in the two-level symbolic cover, where s_i appears in one cube and s_j appears in the second one, of the proximity between the two cubes.

The cost function of the simulated annealing algorithm is the sum over all symbol pairs of the weighted distance among the codes.

The greedy embedding algorithm chooses at each step the symbol that has the strongest weight connection with already assigned symbols, and assigns to it a code that minimizes the above cost function.

Pesto

PESTO [57] is a new tool that resembles JEDI with respect to the basic model, but by means of very skilled algorithmic engineering obtains codes that produce often (as of today) the best starting points for multi-level implementations.

The model starts form the observation, justified in [144], that if x and y are two binary input vectors, $f(x)$ is a single output boolean function, and

$$P = \{(x, y) \mid hamming\_distance(x, y) = 1 \ and \ f(x) = f(y)\},$$

then, within a class of "related" functions, the larger the size of P, the simpler the implementation of f.

An adjacency matrix is constructed and a metric that is a function of the matrix and of the state encodings is maximized by means of simulated annealing. For problems like state assignment

the adjancency matrix is a weighted sum of an input adjacency matrix and an output adjacency matrix.

Binary vectors are considered adjacent when they have Hamming distance one. For each pair of states there is an entry in the input adjacency matrix set to the number of pairs of 1's in the outputs that would be adjacent if that present state pair were adjacent. Adjacent outputs means that the input vectors for the two outputs differ only in one bit position, i.e., the codes of the present states are at Hamming distance one and the proper inputs are equal. For the proper outputs this information is easily known. For the next state outputs this information is obviously unavailable, so an average number of times that pairs of next states have 1's in the same bit positions is computed by generating random encodings.

For each pair of states there is an entry in the output adjacency matrix set to the number of times it has adjacent inputs. The inputs can be adjacent when the proper inputs are adjacent and the present states are identical or the proper inputs are identical and the present states are adjacent. The former situation is easily known. In the latter situation the information about present states is obviously unavailable, so an average of times that pairs of present states are adjacent is computed by generating random encodings.

The goal is to find a state assignment that maximizes a weighted sum of the contributions of the input and output adjacency matrices. Given a state assignment, an adjacency matrix contributes the sum of pairs of adjacent states weighted by the coefficient of the corresponding entry.

A careful study is made of the relative importance of the weighting factor of the input and output matrices, the number of repeated experiments (since simulated annealing is used to find the maximizing codes), the importance of using information on input don't cares, the parameters of simulated annealing and others. One of the lessons that the implementation of PESTO teaches is that even a simple model, if all algorithmic choices are carefully evaluated, can produce high-quality results. In this case from the experiments PESTO seems to enjoy a noticeable advantage over its competitors JEDI and MUSE especially in the case of large examples, that are those where the robustness of an heuristic is tested and the quality of the result matters more.

Muse

MUSE uses a multi-level representation of the finite state machine to derive the set of weights that are used in the encoding problem.

Its algorithm is as follows:

1. Encode symbolic inputs and outputs with one-hot codes.

2. Use MISII ([12]) to generate an optimized boolean network.

3. Compute a weight for each symbol pair (see below).

4. Use a greedy embedding algorithm trying to minimize the sum over all state pairs of the weighted distance among the codes.

5. Encode the symbolic cover, and run MISII again.

The weight assignment algorithm examines each node function (in sum-of-product form) to see if any of the following cases applies (S_i denotes a state symbol, s_i denotes the corresponding one-hot present state variable, other variables denote primary inputs):

1. $s_1ab + s_2ab + \ldots$: if S_1 and S_2 are assigned adjacent codes, then the cubes can be simplified to a single cube, and we obtain a saving in the encoded network cost.

2. $s_1ab + s_2abc + \ldots$: if S_1 and S_2 are assigned adjacent codes, then the cubes can be simplified (even though they will remain distinct cubes, due to the appearance of c only in the second one) and a common cube (the common state bits and ab) can be extracted. For example, if S_1 is encoded as $c_0\overline{c_1}\overline{c_2}$ and S_2 is encoded as $c_0\overline{c_1}c_2$, the expression above can be simplified as $c_0\overline{c_1}abc + c_0\overline{c_1}\overline{c_2}ab$.

3. $s_1abc + s_2abd + \ldots$: same as above, but only a common cube (the common state bits and ab) can be extracted.

For each occurrence of the above cases the weight of the state pair is increased by an amount that is proportional to the estimated gain if the two states are assigned adjacent codes. For example, if abc is extracted from $f = abcd$, $g = abce$, (cost 8 literals) then we obtain $f = hd$, $g = he$, $h = abc$ (cost 7 literals), and the gain obtained extracting h is 1.

Each gain is also multiplied by the number of distinct paths from the node to a network output. This heuristic gives a higher gain to common subexpressions that are used in many places in the network, so that their extraction gives a high reduction in the network cost. If the codes in the pair are assigned adjacent codes, then hopefully MISII will be able to extract again useful subexpressions after the encoding.

The algorithm described above takes into account only present state symbols. Another heuristic algorithm is used to estimate the "similarity" among the next state functions. This "next-state oriented" algorithm adds to the weight of each pair of states the gain of common subexpressions that can be extracted from the functions generating that pair of next states in the one-hot encoded network. For example, if n_i denotes a one-hot next state variable and N_i the corresponding state symbol, $n_1 = abcd$ and $n_2 = abce$ have a common subexpression abc of gain 1 (see above), so the weight of the (N_1, N_2) pair is incremented by 1 due to this subexpression.

The embedding algorithm, using the weights computed above, chooses the unencoded state that has a maximum weight connection with the already encoded states and assigns to it a code that has the minimum weighted distance from the already encoded states.

MUSE uses a cost function that is a closer representation of reality with respect to MUSTANG and JEDI, but there is no guarantee that the optimizations performed on the one-hot encoded network are the best ones for all possible encodings, and that MISII will choose to perform the same optimizations when it is run on the encoded network.

Mis-mv

In order to have a satisfactory solution of the multi-level encoding problem we must have a closer view of the real cost function, the number of literals in the encoded network. The weight matrix is rather far from giving a complete picture of what happens to this cost function whenever an encoding decision is made.

Following the pattern outlined in the previous sections for the two-level case, we should perform a multi-level symbolic minimization, and derive constraints that, if satisfied, can guarantee some degree of minimality of the encoded network.

MIS-MV, unlike the previous programs, performs a full multi-level multiple-valued minimization of a network with a symbolic input. Its algorithms are an extension to the multiple-valued case of those used by MISII (the interested reader is referred to [85] for a detailed explanation of these algorithms).

Its overall strategy is as follows:

1. Read the symbolic cover. The symbolic output is encoded one-hot, the symbolic input is left as a multiple-valued variable.

2. Perform multi-level optimization (simplification, common subexpression extraction, decomposition) of the multiple-valued network.

3. Encode the symbolic input so that the total number of literals in the encoded network is minimal (simulated annealing is used for this purpose, while extensions of constrained embedding algorithms from the two level case are being studied).

A set of theorems, proved in [73], guarantees that step 2 of the above algorithm is complete, i.e. that all possible optimizations in all possible encodings can be performed in multiple-valued mode *provided that the appropriate cost function is available*.

The last observation is a key to understand both strengths and limits of this approach: the cost function that MIS-MV minimizes is only an approximate *lower bound* on the number of literals that the encoded network will have (much in the same spirit as what happens in the two-level case with symbolic minimization). This lower bound can be reached if and only if all the face constraints from all the nodes in the multiple-valued network can be simultaneously satisfied in a minimum length encoding, which is not possible in general (each node has a multiple-valued function, so the constraints can be extracted as described in Section 4.1.2). This lower bound is approximate because further optimizations on the encoded network can still reduce the number of literals.

In order to take this limitation into account, MIS-MV computes at each step the currently optimal encoding, and uses it as an estimate of the cost of each multiple-valued node.

For example, if one denotes by $S^{\{1,2,3,4\}}$ a multiple-valued literal representing the boolean function that is true when variable S has value 1, 2 , 3 or 4, the estimated cost of $S^{\{1,2,3,4\}}$ with the codes:
$e(S^{\{1\}}) = \overline{c_1}\overline{c_2}\overline{c_3}$, $e(S^{\{2\}}) = \overline{c_1}\overline{c_2}c_3$, $e(S^{\{3\}}) = \overline{c_1}c_2\overline{c_3}$, $e(S^{\{4\}}) = \overline{c_1}c_2c_3$, $e(S^{\{5\}}) = c_1\overline{c_2}\overline{c_3}$, $e(S^{\{6\}}) = c_1\overline{c_2}c_3$
would be 1, since the minimum sum of products expression for $\overline{c_1}\overline{c_2}\overline{c_3} + \overline{c_1}\overline{c_2}c_3 + \overline{c_1}c_2\overline{c_3} + \overline{c_1}c_2c_3$ with the don't cares (unused codes) $c_1c_2\overline{c_3} + c_1c_2c_3$ is c_1.

Currently MIS-MV does not handle the output encoding problem. Its approach, though, can be extended to handle a symbolic minimization procedure similar to what is explained in section 4.1.2, and therefore to obtain a solution also to this problem.

Comparison of Different Methods

Programs such as MUSTANG, JEDI and PESTO rely only on the two-level representation of the symbolic cover to extract a similarity measure between the context in which each pair of symbols appear. This measure is used to drive a greedy embedding algorithm that tries to keep similar symbols close in the encoded boolean space. This has clearly only a weak relation with

the final objective (minimum cost implementation of a boolean network), and it makes an exact analysis of the algorithm performance on benchmark examples hard. Still it must be said that the implementation of PESTO stands out as a very skillful one, to point that this program is currently the best achiever especially on large examples.

Some improvement can be seen in MUSE, that uses a one-hot encoding for both input and output symbols, and then performs a multi-level optimization. In this way at least some of the actual potential optimizations can be evaluated, and their gain can be used to guide the embedding, but there is no guarantee of optimality in this approach, and the output encoding problem is again solved with a similarity measure.

Full multi-level multiple-valued optimization (MIS-MV) brings us closer to our final objective, because all potential optimizations can in principle be evaluated. The complexity of the problem, though, limits this potentiality to an almost greedy search, as in MISII.

Still we do not have a complete solution to the encoding problem for multi-level implementation because:

1. We need to improve our estimate of the final cost to be used in multi-level multiple-valued optimization.

2. The problem of optimal output encoding must be addressed directly.

The algorithms described in this section, though, can and have been successfully used, and the path towards an optimal solution is at least clearer than before.

4.1.4 Experimental Results

We report some comparisons among available state assignment programs based on the techniques discussed in the previous sections. For the experiments we used the MCNC '89 set of benchmark FSM's.

The Two-level Case

We report one set of experiments that compare programs for two-level state assignments.

Table 4.1 summarizes the results obtained running the algorithms of NOVA [147], KISS [92] and random state assignments. The results of NOVA were obtained running ESPRESSO [114] to obtain the input constraints and the symbolic minimizer of NOVA built on top of ESPRESSO to obtain the mixed input/output constraints, NOVA to satisfy the constraints on the codes of the states and of the

symbolic inputs (if any), and ESPRESSO again to obtain the final area of the encoded FSM. The best result of the different options of NOVA was shown in the table. The results of KISS were obtained running ESPRESSO to obtain the input constraints, KISS to satisfy the constraints on the codes of the states and of the symbolic inputs (if any), and ESPRESSO again to obtain the final area of the encoded FSM. The areas under random assignments are the best and the average of a statistical average of a number of different (number of states of the FSM + number of symbolic inputs, if any) random state assignments on each example. The final areas obtained by the best solution of NOVA average 20% less than those obtained by KISS, and 30% less than the best of a number of random state assignments. NOVA can use any number of encoding bits greater than or equal to the minimum. The best results of NOVA on the benchmark of Table 4.1 have been obtained with a minimum encoding length, but this is not always the case. KISS uses a code-length sufficient to satisfy all input constraints. Since it satisfies the constraints by an heuristic algorithm it does not always achieve the minimum necessary code-length.

Notice that the lower bound provided by symbolic minimization is often larger than the best upper bound achieved by encoding the FSM's, even though the available programs model only partially the effects of output encoding. This means that output encoding is more important than input encoding on the quality of final results.

Comparisons for some of the approaches mentioned above [124, 39] have not been carried out for the lack of an available implementation.

The Multi-level Case

We report a set of experiments that correlate good two-level state assignment to the corresponding multi-level logic implementation, comparing against an estimation-based multi-level encoding algorithm.

Table 4.2 reports the number of literals after running through the standard boolean optimization script in the multi-level logic synthesis system MISII [12] with encodings obtained by NOVA, MUSTANG [36], JEDI [77] and random state assignments. In the case of NOVA only the best minimum code-length two-level result was given to MISII . MUSTANG was run with *-p*, *-n*, *-pt*, *-nt* options and minimum code-length. JEDI was run with all available options and minimum code-length [76]. In all cases ESPRESSO was run before MISII. The final literal counts in a factored form of the logic encoded by NOVA average 30% less than the literal counts of the best of a number of random state assignments. The best (minimum code-length) two-level results of MUSTANG, and JEDI versus the

example	random		KISS			NOVA		
	b-area	a-area	#bits	#cubes	area	#bits	#cubes	area
bbara	616	649	5	26	650	4	24	528
bbsse	1089	1144	6	27	1053	4	29	957
bbtas	165	215	3	13	195	3	8	120
beecount	285	293	4	11	242	3	10	190
cse	1947	2087	6	45	1756	4	45	1485
dk14	720	809	9	24	550	6	25	500
dk15	357	376	6	17	391	5	17	289
dk16	1826	1994	12	55	2035	7	54	1188
dk17	320	368	6	19	361	5	17	272
dk27	143	143	4	9	117	4	7	91
dk512	374	418	7	18	414	5	17	289
donfile	1200	1360	12	24	984	5	28	560
ex1	3120	3317	7	42	2436	6	37	2035
ex2	798	912	6	31	744	5	27	567
ex3	342	387	6	18	432	4	17	306
ex5	324	358	5	15	315	4	14	252
ex6	810	850	5	24	792	3	25	675
iofsm	560	579	4	16	448	4	15	420
keyb	3069	3416	8	47	1880	5	48	1488
mark1	760	782	5	19	779	4	17	646
physrec	1677	1741	5	34	1564	4	33	1419
planet	4896	5249	6	89	4539	6	86	4386
s1	3441	3733	5	81	2997	5	63	2331
sand	4278	4933	6	95	4655	6	89	4361
scf	19650	21278	8	140	18760	7	137	17947
scud	2262	2533	6	71	2698	3	62	1798
shiftreg	132	132	3	6	72	3	4	48
styr	5031	5591	6	91	4186	5	94	4042
tbk	5040	6114	na	na	na	5	57	1710
train11	221	241	6	10	230	4	9	153
TOTAL	65453	72002			na			51053
%	100	110			na			77

Table 4.1: Comparison of FSM's encoding for two-level implementation

best (minimum code-length) two-level results of NOVA are also reported. Notice that in the case of MUSTANG and JEDI the run that achieved the minimum number of cubes is not necessarily the same that achieved the minimum number of literals. In the case of NOVA only the best two-level result was fed into MISII, so the data reported refer to the same minimized cover. Even though NOVA was not designed as a multi-level state-assignment program, its performances compare successfully with MUSTANG. Among the three programs, the best literal counts are often given by JEDI. These data show that a state assignment that gives a good two-level implementation provides a good starting point for a multi-level implementation, but it does not match the quality reached by algorithms specialized for multi-level implementations. Early claims in [151, 152, 150] that two-level tools were good enough also for multi-level implementations reflected mainly a temporary lack of good tools for multi-level implementations.

We report two kinds of experiments to verify the validity of MIS-MV as *input encoder*:

- Compare the relative importance of the various multi-valued optimization steps.
- Compare MIS-MV with some existing *state assignment* programs, such as JEDI [77], MUSE [42], MUSTANG [36] and NOVA [147]. Notice that we want to compare only the input encoding algorithms of these programs and so we need to "shut off" all effects due to the encoding of the output part, captured by purpose (these programs embody also heuristics for the output encoding problem) or by chance. Therefore we replaced the codes returned by each program in the present state *only*, while the next state was simply replaced by one-hot codes.

  The experiments were conducted as follows:

- A single simplified boolean script (using *simplify* only once) was used both for multi-valued and binary valued optimization.
- The script was run twice in all cases.
- MIS-MV:

  1. ESPRESSO was run on the unencoded machine.
  2. All or part of the first script was run in MIS-MV's multi-valued mode.
  3. The inputs were encoded, using the simulated annealing algorithm.
  4. The remaining part of the first script and the second script were run in binary-valued mode.

example	JEDI	MUSTANG	NOVA	JEDI	MUSTANG	NOVA	random
	#cubes	#cubes	#cubes	#lit	#lit	#lit	#lit
bbara	24	25	24	57	64	61	84
bbsse	30	31	29	111	106	132	149
bbtas	9	10	8	21	25	21	31
beecount	12	12	10	39	45	40	59
cse	52	48	45	200	206	190	274
dk14x	29	32	26	106	117	98	164
dk15x	19	19	17	67	69	65	73
dk16x	64	71	52	225	259	246	402
donfile	33	49	28	76	160	88	193
ex1	48	55	44	250	280	215	313
ex2	35	36	27	122	119	96	162
ex3	19	19	17	66	71	76	83
keyb	52	58	48	140	167	200	256
mark1	17	19	17	66	76	86	116
physrec	39	37	33	132	159	150	178
planet	93	97	86	547	544	560	576
s1	57	69	63	152	183	265	444
sand	105	108	96	549	535	533	462
scf	147	148	137	812	791	839	890
scud	57	83	62	127	286	182	222
shiftreg	4	4	4	0	2	0	16
styr	100	112	94	508	546	511	591
tbk	57	136	57	278	547	289	625
train11	11	10	9	27	37	43	44
TOTAL	1113	1288	1033	4678	5394	4986	6407
%	107	124	100	93	108	100	130

Table 4.2: Experiments on FSM's encoding for two and multi-level implementation

- JEDI, MUSE, MUSTANG and NOVA:

  1. Each program was run in *input oriented* mode ("-e i" for JEDI, "-e p" for MUSE, "-pc" for MUSTANG and "-e ih" for NOVA) to generate the codes.
  2. The symbolic input was encoded.
  3. ESPRESSO was run again, using the invalid states as don't cares.
  4. The script was executed twice.

We performed seven experiments on each machine, four using JEDI, MUSE, MUSTANG and NOVA, and three using MIS-MV. The experiments on MIS-MV differed in the point of the script where the symbolic inputs were encoded (MIS-MV can carry on the multi-level optimizing operations on a multiple-valued network or on the encoded binary-valued network):

1. At the beginning. At this point, both MIS-MV and NOVA extract the same face constraints by multiple-valued minimization. The two programs get different results because of the different face constraints satisfaction strategies. MIS-MV satisfies the face constraints with a simulated annealing algorithm that minimizes the literal count of a two-level implementation. The cost function is computed by calling ESPRESSO and counting the literals. NOVA satisfies the input constraints with a heuristic deterministic algorithm that minimizes the number of product-terms of a two-level implementation.
2. After *simplify*, to verify multiple-valued boolean resubstitution.
3. After algebraic optimization (*gkx, gcx,* . . .), to verify the full power of MIS-MV.

Table 4.3 contains the results, expressed as factored form literals.

4.2 Relation of State Assignment to Other Optimization Steps

In this section we mention very briefly some issues in the interaction of state assignment (and encoding in general) to other steps of sequential synthesis.

4.2.1 State Assignment and State Minimization

State assignment interacts with the other traditional steps of sequential synthesis. Consider FSM decomposition, i.e., the process of replacing an FSM by a network of interconnected

example	JEDI	MUSE	MUSTANG	NOVA	best MIS-MV	beginning	simplify	algebraic optimization
bbara	96	99	96	106	84	84	84	85
bbsse	125	126	148	151	131	130	132	131
bbtas	34	36	37	32	31	35	31	31
beecount	56	60	65	70	56	62	56	58
cse	189	192	208	214	195	191	199	195
dk14	96	102	108	98	79	97	79	81
dk15	65	65	65	65	68	65	68	69
dk16	254	244	314	351	247	225	247	261
dk17	63	58	69	58	62	58	62	64
dk27	30	29	34	38	27	27	27	27
dk512	73	73	78	93	68	70	68	69
donfile	132	131	195	186	123	127	123	123
ex1	256	239	252	246	232	240	232	237
ex2	176	169	197	167	144	143	144	154
ex3	87	96	98	98	82	82	86	82
ex4	71	72	73	84	72	90	74	72
ex5	79	79	80	83	69	67	69	69
ex6	93	92	90	98	84	85	85	84
ex7	87	84	100	94	78	89	79	78
keyb	186	180	203	195	146	186	172	146
lion	16	16	14	16	16	16	16	16
lion9	55	55	61	43	38	40	38	38
mark1	94	92	89	105	92	90	94	92
mc	32	30	30	32	30	35	30	30
modulo12	58	72	77	71	71	71	71	71
opus	83	70	88	90	70	87	70	74
planet	453	511	538	551	466	512	466	473
s1	339	291	377	345	249	335	253	251
s1a	262	195	264	253	214	217	214	225
s8	50	52	47	48	48	52	48	48
sand	556	498	519	542	509	523	509	529
shiftreg	24	25	34	35	24	24	24	24
styr	427	418	460	501	438	442	438	473
tav	27	27	27	27	27	27	27	27
TOTAL	4724	4578	5135	5186	4370	4624	4415	4487

Table 4.3: Multi-level input encoding comparison

FSM's, preserving the sequential behavior. One can see state assignment as producing an FSM decomposition: there is a component FSM of two states (1 memory element) for each encoding bit, and each component FSM depends on the the state of the other components. Connections between state assignment and FSM decomposition have been considered in [34, 37, 6, 5].

4.2.2 State Assignment and State Minimization

A sequential behavior may be represented by many different STG's, and different STG's of the same behavior may lead to different logical implementations. This makes elusive the goal of obtaining the best implementation of a given sequential behavior. We demonstrate with an example the problem.

Consider FSM's M_1 (left) and M_2 (right):

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

FSM M_2 is a state minimized version of FSM M_1. An encoding of M_2 is: $s_1 = 00, s_2 = 01, s_3 = 10$ and a corresponding minimum encoded implementation of M_2 is:

```
000 011
100 010
-01 101
-10 000
```

This implementation could not have been obtained by encoding M_1, it was necessary instead to obtain first a different STG representation of the same behavior by means of state minimization. So one could think that by doing state minimization and then state assignment the best implementation could be obtained. It is not always so, as it was recognized long ago by Hartmanis and Stearns, who gave in [55] an example of an FSM whose best implementation has fewer product-terms than the best implementation obtained after state minimization of the original machine. Therefore in order to get a minimum implementation one should merge the steps of state minimization and state assignment. We will see, when discussing generalized prime implicants, how the introduction of symbolic Boolean relations allows doing the two steps at the same time, for CSFSM's. Even this last technique will not allow to explore all possible STG representations of a given sequential behavior, but if the original STG is redundant it allows to choose a reduced STG in such a way to optimize the state assignment step.

We will mention later that by using symbolic relations some cases of the interaction of state minimization and state assignment can be modeled exactly, but with little hope of practical solutions. Recently Calazans [17] proposed an heuristic algorithm to use information about compatible states of ISFSM's while doing state assignment.

4.2.3 State Assignment and Testability

Unate state assignments to guarantee testability by construction were proposed first in [140]. The logic to compute the outputs and the encoding of the next state is said to be unate in a given state variable, if the output and next state functions can be expressed as sums of products where the given variable appears either uncomplemented or complemented, but not both. In [111] a case was made for a variation of unate encoding called half-hot encoding that may allow sometimes savings in the number of columns of the encoded PLA. Half-hot encodings have exactly half the total number of state variables set to 1. The penalty on the number of necessary product terms was not addressed. The issue of encoding for testable implementations of small area using (k, p) codes was addressed recently in [83]. (k, p) codes have length p with exactly k bits set to 1 and they result in unate realizations of the encoded FSM. Information on compatibility between states was also used in the state assignment phase.

Chapter 5

Symbolic Minimization

5.1 Introduction

The optimization of logic functions performed on the Boolean representation depends heavily on the encoding chosen to represent the symbolic variables.

The cost function that estimates the area optimality of an encoding depends on the target implementation: two-level or multi-level or field-programmable gate arrays (FPGA's). The cost of a two-level implementation is the number of product-terms or the area of a programmable logic array (PLA). A commonly used cost of a multi-level implementation is the number of literals of a technology-independent representation of the logic. FPGA's come in different architectures with associated costs. Other optimization objectives may be related to power consumption, speed and testability. It may even be the case that the objective is a correctness requirement, as is race-freeness in state assignment of asynchronous circuits.

The following optimal encoding problems may be defined:

(A) Optimal encoding of inputs of a logic function. A problem in class A is the optimal assignment of *opcodes* for a microprocessor.

(B) Optimal encoding of outputs of a logic function.

(C) Optimal encoding of both inputs and outputs (or some inputs and some outputs) of a logic function.

(D) Optimal encoding of both inputs and outputs (or some inputs and some outputs) of a logic function, where the encoding of the inputs (or some inputs) is the same as the encoding of the

outputs (or some outputs). Encoding the states of a finite state machine (FSM) is a problem in class D since the state variables appear both as input (present state) and output (next state) variables. Another problem in class D is the encoding of the signals connecting two (or more) combinational circuits.

Here we concentrate on problems in class D for optimal two-level implementations. In particular we will refer mostly to the problem of encoding FSM's, since there is no loss of generality and they are of great practical interest.

We will build on the paradigm started by [92]. It involves optimizing the symbolic representation (symbolic minimization), and then transforming the optimized symbolic description into a compatible two-valued representation, by satisfying encoding constraints (bit-wise logic relations) imposed on the binary codes that replace the symbols. This approach guarantees an upper bound on the size of the encoded symbolic function provided all the encoding constraints are satisfied. Encoding via symbolic minimization may be considered a three step process. The first phase consists of multiple-valued optimization. The second step is to extract constraints on the codes of the symbolic variables, which, if satisfied, guarantee the existence of a compatible Boolean implementation. The third step is assigning to the symbols codes of minimum length that satisfy these constraints, if the latter imply a set of non-contradictory bit-wise logic relations.

When the target implementation is two-level logic, the first step may consist of one or more calls [92, 91] to a multiple-valued minimizer [114], after representing the symbolic variables with positional cube notation [139, 114]. Then constraints are extracted and a constraints satisfaction problem is set up.

Using the paradigm of symbolic minimization followed by constraints satisfaction, the most common types of constraints that may be generated [92, 91, 39, 116] are four. The first type, generated by the input variables, are *face-embedding* constraints. The three types generated by the output variables are *dominance*, *disjunctive* and *disjunctive-conjunctive* constraints. Each face-embedding constraint specifies that a set of symbols is to be assigned to one *face* of a binary n-dimensional cube and no other symbol should be in that same face. Dominance constraints require that the code of a symbol covers bit-wise the code of another symbol. Disjunctive constraints specify that the code of a symbol must be expressed as the bit-wise disjunction (*oring*) of the codes of two or more other symbols. Disjunctive-conjunctive constraints specify that the code of a symbol must be expressed as the bit-wise disjunction (*oring*) of the bit-wise conjunction (*anding*) of the codes of two or more other symbols.

The presentation is organized as follows. In Section 5.2 we present the encoding problem for optimal two-level implementations. In Section 5.3 the new symbolic minimization algorithm is described, while procedures for symbolic reduction and symbolic oring are explained, respectively, in Section 5.4 and in Section 5.5. Section 5.6 analyzes some ordering schemes. In Section 5.7 mention is made of the algorithms used for checking encodeability. An example is demonstrated in Section 5.8, and experiments are reported in Section 5.9, with final conclusions drawn in Section 5.10.

5.2 Encoding for Two-level Implementations

5.2.1 Multi-valued Minimization

Advances in the state assignment problem, reported in [93, 11, 92], made a key connection to multiple-valued logic minimization, by representing the states of a FSM as the set of possible values of a single multiple-valued variable. A multiple-valued minimizer, such as [114], can be invoked on the symbolic representation of the FSM. This can be done by representing the symbolic variables using the positional cube notation [139, 114]. The effect of multiple-valued logic minimization is to group together the states that are mapped by some input into the same next-state and assert the same output. To get a compatible boolean representation, one must assign each of the groups of states obtained by MV minimization, (called face or input constraints) to subcubes of a boolean k-cube, for a minimum k, in a way that each subcube contains all and only all the codes of the states included in the face constraint. This problem is called face embedding problem.

It is worth mentioning that the face constraints obtained through straightforward symbolic minimization are sufficient, but not necessary to find a two-valued implementation matching the upper bound of the multi-valued minimized cover. As it was already pointed out in [91], for each implicant of a minimal (or minimum) multi-valued cover, one can compute an *expanded implicant*, whose literals have maximal (maximum) cardinality and a *reduced implicant* whose literals have minimal (minimum) cardinality. By bit-wise comparing the corresponding expanded and reduced implicant, one gets *don't cares* in the input constraint, namely, in the bit positions where the expanded implicant has a 1 and the reduced implicant has a 0. The face embedding problem with *don't cares* becomes one of finding a cube of minimum dimension k, where, for every face constraint, one can assign the states asserted to vertices of a subcube that does not include any state

not asserted, whereas the *don't care* states can be put inside or outside of that subcube. One can build examples where the presence of *don't cares* allows to satisfy the input constraints in a cube of smaller dimension, than it would be possible otherwise.

5.2.2 Symbolic Minimization

Any encoding problem, where the symbolic variables only appear in the input part, can be solved by setting up a multiple-valued minimization problem followed by satisfaction of the induced face constraints. However, the problem of state assignment of FMS's is only partially solved by this scheme, because the encoding of the symbolic output variables is not taken into account (e.g. the next state variable). Simple multiple-valued minimization disjointly minimizes each of the on-sets of the symbolic output functions, and therefore disregards the sharing among the different output functions taking often place when they are implemented by two-valued logic. We will see now more powerful schemes to deal with both input and output encoding.

In [91, 147] a new scheme was proposed, called *symbolic minimization*. Symbolic minimization was introduced to exploit bit-wise dominance relations between the binary codes assigned to different values of a symbolic output variable. The fact is that the input cubes of the onset of a dominating code can be used as don't cares for covering the input cubes of the onset of a dominated code. The core of the approach is a procedure to find useful dominance (called also covering) constraints between the codes of output states. The translation of a cover obtained by symbolic minimization into a compatible boolean representation defines simultaneously a face embedding problem and an output dominance satisfaction problem. Notice that any output encoding problem can be solved by symbolic minimization. Symbolic minimization was applied also in [115], where a particular form of PLA partitioning is examined, by which the outputs are encoded to create a reduced PLA that is cascaded with a decoder.

However, to mimic the full power of two-valued logic minimization, another fact must be taken into account. When the code of a symbolic output is the bit-wise disjunction of the codes of two or more other symbolic outputs, the on-set of the former can be minimized by using the on-sets of the latter outputs, by "redistributing" the task of implementing some cubes. An extended scheme of symbolic minimization can therefore be defined to find useful dominance and disjunctive relations between the codes of the symbolic outputs. The translation of a cover obtained by extended symbolic minimization into a compatible boolean representation induces a face embedding, output dominance and output disjunction satisfaction problem.

(1)	10	st1	st2	11
(2)	00	st2	st2	11
(3)	01	st2	st2	00
(4)	00	st3	st2	00
(5)	10	st2	st1	11
(6)	10	st3	st1	11
(7)	00	st1	st1	- -
(8)	01	st3	st0	00
(9)	11	st1	st1	10
(10)	11	st3	st3	01
(11)	11	st0	st0	11

(1')	-0	st1,st2	st2	11
(2')	0-	st2,st3	st2	00
(3')	10	st2,st3	st1	11
(4')	00	st1	st1	- -
(5')	01	st3	st0	00
(6')	11	st1,st0	st1	10
(7')	11	st0,st3	st3	01

Figure 5.1: Covers of FSM-2 before and after symbolic minimization

In Figure 5.1, we show the initial description of a FSM and an equivalent symbolic cover returned by an extended symbolic minimization procedure.

The reduced cover is equivalent to the original one if we impose the following constraints on the codes of the states.

Product terms (1'), (3') and (4') are consistent with the original product terms (5) and (7) if we impose $code(st1) > code(st2)$. In a similar way, product terms (2') and (5') are consistent with the original product term (8) if we impose $code(st0) > code(st2)$. The product terms (1') and (2') yield also the face constraints $face(st1, st2)$ and $face(st2, st3)$, meaning that the codes of $st1$ and $st2$ ($st2$ and $st3$) span a face of a cube, to which the code of no other state can be assigned. The previous face and dominance constraints together allow to represent the four original transitions (1), (2), (3), (4) by two product terms (1') and (2').

Product term (3') is equivalent to the original transitions (5) and (6) and yields the face constraint $face(st2, st3)$. This saving is due to a pure input encoding join effect.

Finally the product terms (6'), (7') represent the original transitions (9), (10) and (11). The next state of (11) is $st0$, that does not appear in (6') and (7'). But, if we impose the disjunctive constraint $code(st0) = code(st1) \vee code(st3)$, i.e., we force the code of $st0$ to be the bit-wise or of the codes of $st1$ and $st3$, we can redistribute the transition (11) between the product terms (6') and (7'). The product terms (6') and (7') yield also the face constraints $face(st1, st0)$ and

(1")	-0	0-	00	11
(2")	0-	-0	00	00
(3")	10	-0	01	11
(4")	00	01	01	- -
(5")	01	10	11	00
(6")	11	-1	01	10
(7")	11	1-	10	01

Figure 5.2: Encoded cover of FSM-2

$face(st0, st3)$; together with the previous disjunctive constraint they allow the redistribution of transition (11).

We point out that if we perform a simple MV minimization on the original description we save only one product term, by the join effect taking place in transition (3').

An encoding satisfying all constraints can be found and the minimum code length is two. A solution is given by $st_0 = 11, st_1 = 01, st_2 = 00, st_3 = 10$. If we replace the states by the codes in the minimized symbolic cover, we obtain an equivalent Boolean representation that can be implemented with a PLA, as shown in Figure 5.2. Note that we replace the groups of states in the present state field with the unique face assigned to them and that product term (2") is not needed, because it asserts only zero outputs. Therefore the final cover has only six product terms.

5.2.3 Completness of Encoding Constraints

An important question is whether the constraints described earlier are sufficient to explore the space of all encodings. More precisely, the question is: find the class of encoding constraints such that by exploring all of them one is guaranteed to produce a minimum encoded implementation. Of course exploring all the encoding constraints of a given class may be impractical, but if the answer to the previous question is affirmative, one has characterized a complete class that can lead in line-of-principle to an optimal solution. This would make more attractive an heuristic that explores the codes satisfying the constraints of such a class.

Theorem 5.2.1 *Face and disjunctive constraints are sufficient to obtain a minimum two-level implementation of a state-minimized FSM if the minimum implementation has as many hardware states as there are symbolic states.*

Proof: Consider an FSM F. Let the codes that produce a minimum implementation of the FSM be given, together with the best implementation C (here minimum or best refers to the smallest cardinality of a two-level cover). Suppose that the product-terms of the minimum encoded implementation C are all prime implicants. Consider each cube of C. Its present state part will contain the codes of one or more states and it will translate into a face constraint. Its next state part will correspond to the code of a symbolic state (using the hypothesis that there are as many hardware states as symbolic states). Consider now each minterm of the original FSM F. It will be covered in the input part (proper input and present state) by one or more cubes of C; this will translate into a disjunctive constraint whose parent is the next state of the minterm and whose children are the next states of the covering cubes of C.

The face constraints and disjunctive constraints so obtained are necessary for a set of codes to produce such a minimum implementation, when they are replaced in the original cover and then the cover is minimized. But are they sufficient ? There may be many sets of codes that satisfy these constraints. Is any such set sufficient to obtain a minimum cover ? The answer if yes, if after that the set of codes is replaced in the original FSM, an exact logic minimizer is used. Indeed, if this set of codes satisfies the encoding constraints, by construction they make possible to represent the minterms of the original FSM cover by the cubes of the minimum cover C. Therefore an exact logic minimizer will produce either C or a different cover of the same cardinality as C [1]. ■

Theorem 5.2.2 *Face and disjunctive-conjunctive constraints are sufficient to obtain a minimum two-level implementation of a state-minimized FSM.*

Proof: If there are as many hardware states as there are symbolic states the previous result applies. If the best implementation has more hardware states than symbolic states, one must introduce disjunctive-conjunctive constraints. The reason is that it is not anymore always true that the next state of a cube $c \in C$ corresponds to the code of a symbolic state. Suppose that the next state of a cube c is not the code of a symbolic state. c cannot be a minterm in the input part, otherwise, since we suppose that C contains only prime implicants, the next state of c must be exactly the code

[1] The hypothesis that the FSM is state-minimized guarantees that the minimum implementation does not have fewer hardware states than there are symbolic states.

of the state of the symbolic minterm in F to which c corresponds. So c must contain more than one minterm in the input part, say w.l.o.g. that c contains exactly two minterms m_1 and m_2, each corresponding to a symbolic minterm of the care set of F. If the symbolic minterms corresponding in F to c_1 and c_2 assert next states s_1 and s_2, the next state of c must be the intersection of the codes of s_1 and s_2 (for sure the next state of c must be dominated by the intersection of the codes of s_1 and s_2, but we suppose that c is a prime implicant and that it contains exactly minterms m_1 and m_2 of the care set, so we can say that the next state of c is exactly the intersection of the codes of s_1 and s_2).

Therefore for each symbolic minterm m_s in F one defines a disjunctive-conjunctive constraint enforcing that the code of the next state of m_s is a disjunction of conjunctions, where each disjunct is contributed by one of the cubes of C that contain the input part of the minterm corresponding to m_s, and for each such cube c_{m_s} the conjuncts are the codes of the next states asserted by all the care set minterms that c_{m_s} contains. The rest of the reasoning goes as in the previous theorem. ■

Disjunctive-conjunctive constraints were introduced for the first time in [39], as the constraints induced by generalized prime implicants. Our derivation shows that they arise naturally when one wants to find a complete class of encoding constraints. In our symbolic minimization algorithm we used as the class of encoding constraints face constraints, dominance constraints and disjunctive constraints. Dominance constraints are not necessary, but they have been considered useful in developing an heuristic search strategy. We did not use disjunctive-conjunctive constraints in the heuristic procedure presented here.

5.3 A New Symbolic Minimization Algorithm

5.3.1 Structure of the Algorithm

In this section a new more powerful paradigm of symbolic minimization is presented. An intuitive explanation of symbolic minimization as proposed in [91] and enhanced in [147] has been given in Section 5.2. To help in highlighting the differences of the two schemes, the one in [147] is summarized in Figure 5.3.

The new scheme of symbolic minimization features the following novelties.

- Symbolic oring. Disjunctive constraints are generated corresponding to the case of transitions of the initial cover implicitly expressed by other transitions in the encoded two-level

1. Input data cover C with q symbolic outputs,
   optional binary outputs,
   empty acyclic graph G,
   and empty cover FinalP
   Output is the graph G and the minimal cover FinalP
2. On_k = on-set implicants of k-th output symbol
   with the corresponding binary outputs unchanged
3. Repeat Steps 4 through 9 q times
4. i = select a symbol
5. $Dc_i = \cup On_j$,
   for all j for which there is no path from vertex i
   to vertex j in G
6. $Off_i = \cup On_j$,
   for all j for which there is a path from vertex i
   to vertex j in G
7. MB_i = minimize(On_i, Dc_i, Off_i)
8. M_i = implicants of MB_i
   that are in the on-set of symbol i
9. $G = G \cup \{(j, i)$ such that M_i intersects $On_j\}$
   $P = P \cup MB_i$
10. FinalP = minimize(P)

Figure 5.3: Old Symbolic Minimization Scheme

representation, because of the oring effects in the output part.

- Implementability. Product-terms are accepted in the symbolic cover, only when they yield satisfiable encoding constraints.

- Symbolic reduction. Symbolic minimization is iterated until an implementable cover is produced. A symbolic reduction procedure guarantees that this always happens.

At last, codes satisfying the given encoding constraints are generated. The accuracy of the synthesis procedure can be measured by the fact that the cardinality of the symbolic minimized cover is very close to the cardinality of the original encoded FSM minimized by ESPRESSO [11]. This will be shown in the section of results.

We introduce the following abbreviations useful in the description of the algorithm:

- $IniCov = (Fc, Dc, Rc)$ is the initial cover of a 1-hot encoded FSM, where Fc, Dc and Rc are, respectively, the on-set, dc-set and off-set of the 1-hot encoded FSM.

- Ns is the set of next states of a FSM. Fc_{ns}, Dc_{ns} and Rc_{ns} are the set of product-terms asserting ns, respectively, in Fc, Dc and Rc, $\forall ns \in Ns$.

- On_{ns}, $Dcare_{ns}$ and Off_{ns} are, respectively, the on-set, dc-set and off-set of next state ns, $\forall ns \in Ns$, On_{ns}.

- On_{bo}, Dc_{bo} and Off_{bo} are, respectively, the on-set, dc-set and off-set of the binary output functions.

- $PartCov = (OnCov, DcCov, OffCov)$ is the cover of a fragment of a 1-hot encoded FSM, where $OnCov$, $DcCov$ and $OffCov$ are, respectively, the on-set, dc-set and off-set of the given fragment.

- $Cons_{ns}$ is the set of input and output constraints yielded by symbolic minimization of Fc_{ns}, $\forall ns \in Ns$. The sets $Cons_{ns}$ are cumulated in $Cons$.

- $ExpCov_{ns}$ and $RedCov_{ns}$ are, respectively, a maximally expanded and a maximally reduced minimized cover of Fc_{ns}, $\forall ns \in Ns$. The sets $ExpCov_{ns}$ and $RedCov_{ns}$ are cumulated, respectively, in $ExpCov$ and $RedCov$.

At the each step of the symbolic minimization loop a new next state ns is chosen by the procedure *SelectState*, described in Section 5.6. The goal is to determine a small set of

multiple-valued product-terms that represent the transitions of Fc_{ns}. The procedure *SymbOring*, described in Section 5.5, determines Or_{ns}, the transitions of Fc_{ns} that can be realized by expanding some product-terms in the current $RedCov$ and choosing the expansions in the interval ($RedCov$, $ExpCov$). This expansion operation yields updated encoding constraints (here also disjunctive constraints are generated) that must be imposed to derive an equivalent two-level implementation. The rest of Fc_{ns} is minimized, putting in its off-set the on-sets of all states selected previously [2]. The minimization is done calling ESPRESSO, without the final *make_sparse* step. This produces $ExpCov_{ns}$, a maximally expanded minimized cover. Calling the ESPRESSO procedure *mv_reduce* on $ExpCov_{ns}$ produces $RedCov_{ns}$, a maximally reduced minimized cover. The reduced minimized cover $RedCov_{ns}$ yields new encoding constraints $Cons_{ns}$.

If it turns out that the constraints in $Cons_{ns}$ are not compatible with the constraints already in $Cons$, a *SymbReduce* procedure is invoked to redo the minimizations of Fc_{ns} and produce covers that yield encoding constraints compatible with those currently accepted in $Cons$. In Section 5.4, where *symb_reduce* is described, it is shown that this always happens, i.e. this symbolic reduction step always produces an implementable symbolic minimized cover of Fc_{ns}. The compatible constraints $Cons_{ns}$ are added to $Cons$ and the new accepted covers $ExpCov_{ns}$ and $RedCov_{ns}$ are added, respectively, to $ExpCov$ and $RedCov$. Finally, codes satisfying the encoding constraints in $Cons$ are found and replaced in the reduced symbolic minimized cover $RedCov$. The resulting encoded minimized cover $EncRedCov$ is usually of the same cardinality as the cover obtained by replacing the codes in the original symbolic cover and then minimizing it with ESPRESSO. $EncRedCov$ can be minimized again using ESPRESSO to produce a cover $MinEncRedCov$, that rarely has fewer product-terms than $EncRedCov$. These statements will be supported by results in the experimental section. To check the correctness of this complex procedure a verification is made of $MinEncRedCov$ against $EncIniCov$. A non-equivalence of them signals an error in the implementation.

The outlined procedure is shown in Figure 5.4. The routines with initial letter in the lower case are directly available in ESPRESSO (not necessarily with the same name and syntactical usage), while the routines with initial letter in the upper case are new and will be described in the following sections.

Proposition 5.3.1 *The algorithm of Figure 5.4 generates an implementable symbolic cover.*

Proof: By construction a product term is added to the symbolic cover, only if it carries constraints

[2] This is not required: one should put only those states that ns covers.

procedure symbolic(Fc, Dc, Rc) {
  do { /* repeat until all next states are selected */
    /* Sel is a set of currently selected states */
    ns = SelectState($Ns - Sel$); $Sel = Sel \cup ns$
    /* Or_{ns} are the transitions of Fc_{ns} expressed by oring */
    (Or_{ns}, $ExpCov$, $RedCov$, $Cons$)
      = SymbOring($IniCov$,$ExpCov$,$RedCov$,$Cons$)
    /* $OnCov$ are the transitions to be covered */
    $OnCov$ = $Fc_{ns} - Or_{ns}$
    /* add the on-sets of states previously selected to the off-set */
    $OffCov$ = $\bigcup_{i \in Sel-ns} On_i$
    /* add binary output off-set */
    $OffCov$ = $OffCov \cup Off_{bo}$
    /* everything else (including Or_{ns}) is in dc-set */
    $DcCov$ = complement($OnCov$,$OffCov$)
    /* invoke espresso with no makesparse */
    $ExpCov_{ns}$ = espresso($OnCov$,$DcCov$,$OffCov$)
    $RedCov_{ns}$ = mv_reduce($ExpCov_{ns}$,$DcCov$)
    $Cons_{ns}$ = Constraints($IniCov$,$ExpCov_{ns}$,$RedCov_{ns}$)
    if (ConstraintsCompatible($Cons$,$Cons_{ns}$) fails)
      ($ExpCov_{ns}$,$RedCov_{ns}$,$Cons_{ns}$) =
        SymbReduce($IniCov$,$PartCov$,$ExpCov_{ns}$,$RedCov_{ns}$,$Cons$,$Cons_{ns}$)
    $ExpCov$ = $ExpCov \cup ExpCov_{ns}$
    $RedCov$ = $RedCov \cup RedCov_{ns}$
    $Cons$ = $Cons \cup Cons_{ns}$
  } while (at least one state in $Ns - Sel$)
  $Codes$ = EncodeConstraints($Cons$)
  $EncRedCov$ = Encode($RedCov$, $Codes$) /* encode symbolic min. cover */
  $MinEncRedCov$ = minimize($EncRedCov$)
  $EncIniCov$ = Encode($IniCov$, $Codes$) /* encode initial FSM */
  $MinEncIniCov$ = minimize($EncIniCov$)
  if (verify($MinEncRedCov$, $EncIniCov$) fails) ERROR
}

Figure 5.4: New Symbolic Minimization Scheme

on the codes that are compatible with the constraints of all the symbolic cubes cumulated up to then. Therefore one guarantees that the symbolic cover is always implementable at any stage of its construction. ■

5.3.2 Slice Minimization and Induced Face and Dominance Constraints

The procedure *Constraints* computes the face and dominance constraints induced by a pair of minimized covers $(RedCov_{ns}, ExpCov_{ns})$ with respect to the original cover Fc. For each product-term $pexp \in ExpCov_{ns}$ there is a companion product-term $pred \in RedCov_{ns}$ obtained from $pexp$ by applying to it the multiple-valued reduce routine of ESPRESSO. For each pair of product-terms $(pred, pexp) \in (RedCov_{ns}, ExpCov_{ns})$ one gets the implied face constraint by considering the 1-hot representation of the input part. For each position k in the input part of the 1-hot representation of $pred$ and $pexp$, opposite bits yield a don't care in the face constraint and equal bits yield the common care bit in the face constraint. Face constraints are generated for all symbolic input variables, including proper symbolic inputs, if any.

Dominance constraints are computed by determining, for each product-term $pred \in RedCov$, the transitions of the original cover Fc that $pred$ intersects in the input part. The next states that these transitions assert must cover the next state of $pred$, for the functionality of the FSM to be maintained. Notice that currently we compute only the dominance constraints implied by the product-terms in $RedCov$. Computing them both for $RedCov$ and $ExpCov$ (as we do in the case of input face constraints with the notion of don't care input constraints), would allow to explore a larger part of the solution space. This is not currently done, because it would make the constraint satisfaction problem more complex.

Oring constraints are generated only in the *SymbOring* procedure described in Section 5.5. In Figure 5.5 the pseudo-code of *Constraints* is shown.

5.4 Symbolic Reduction

The procedure *SymbReduce* is invoked to set up a series of new minimizations that produce an implementable minimized cover of $OnCov$. This is required when a set of constraints $Cons_{ns}$ incompatible with those in $Cons$ are obtained at a certain iteration in the loop of *symbolic*. When this happens, it means that we cannot minimize the current $OnCov$ (with the current $DcCov$) in one shot, because the minimization process would merge multiple-valued product-terms in such a way

```
/* face and dominance constraints induced by (RedCov_ns, ExpCov_ns) */
Constraints(IniCov,ExpCov_ns,RedCov_ns) {
    foreach (pair of product-terms (pred, pexp) ∈ (RedCov_ns, ExpCov_ns)) {
        foreach (position k in the 1-hot representation) {
            if (I(pred)[k] and I(pexp)[k] are opposite bits) face[k] = dc
                else face[k] = I(pred)[k]
        }
        foreach (transition t ∈ Fc) {
            /* don't intersect if t and pred assert same next state */
            if (t and pred assert different next states) {
                if (distance(I(pred),I(t)) = 0) {
                create covering constraint (nxst(t) > nxst(pred))
                }
            }
        }
    }
}
```

Figure 5.5: Derivation of face and dominance constraints

that incompatible constraints are generated. Instead we can minimize $OnCov$ by blocks and control the allowed companion dc-sets so that only compatible constraints are generated. It is evident that in the worst-case, if only one transition of $OnCov$ is minimized at a time, with an empty dc-set, we always obtain implementable product-terms. This is equivalent to perform no minimization at all. In *SymbReduce*, the transitions of $OnCov$ are partitioned into maximal sets of transitions that can be minimized together. Maximal companion dc-sets are found for each previous on-set of transitions.

The routine *SymbReduce* is divided in two steps. In the first step, a maximal subset of $Cons_{ns}$ is sought that is compatible with $Cons$. The rationale is that the companion product-terms of $ExpCov_{ns}$ and $RedCov_{ns}$ are an acceptable cover for a subset of $OnCov$. This is done in a greedy fashion. The constraints of $Cons_{ns}$ compatible with $Cons$ are saved into $AConsTmp$. A new constraint of $Cons_{ns}$ is checked for compatibility with $Cons \cup AConsTmp$. If it is compatible, it is added to $AConsTmp$, otherwise the product-term companion to the constraint is deleted from both $ExpCov_{ns}$ and $RedCov_{ns}$. The transitions of $OnCov$ not covered by the resulting $RedCov_{ns}$ are the new cover that must be minimized in such a way that only implementable multiple-valued product-terms are found. The transitions of $OnCov$ covered by the resulting $RedCov_{ns}$ are instead added to the dc-set.

In the second part, the current $OnCov$ (i.e. the part of the initial $OnCov$ left uncovered by the previous step) is minimized. The transitions of $OnCov$ that can be minimized together are saved into $OnCovTmp$. A new transition t of $OnCov$ is minimized together with $OnCovTmp$ to return both $ExpCovTMp$ and $RedCovTmp$. The implied constraints are computed in $AConsTmp$. If they are compatible with $Cons$, t is added to $OnCovTmp$. In this way one determines sets of transitions that can be minimized together. The dc-set of each such set of transitions is enlarged in a similar greedy fashion. The rationale is that one may obtain more expanded resulting product-terms useful in later stages of the algorithm. Then $ExpCov_{ns}$, $RedCov_{ns}$ and $Cons_{ns}$ are updated, respectively, with the saved accepted sets $ExpCovTmp$, $RedCovTmp$ and $AConsTmp$. This is iterated until all transitions of $OnCov$ are minimized.

The outlined procedure is shown in Figures 5.6 and 5.7. The routines with initial letter in the lower case are directly available in ESPRESSO (not necessarily with the same name and syntactic usage), while the routines with initial letter in the upper case are new.

```
/* PartCov is the triple (OnCov,DcCov,OffCov) */
procedure SymbReducePart1(IniCov,PartCov,ExpCov_ns,RedCov_ns,Cons,Cons_ns) {
    /* choose greedily a maximal subset of compatible constraints */
    /* pt(c) is a product-term companion to constraint c */
    AConsTmp is empty
    foreach (constraint c ∈ Cons_ns) {
        if (ConstraintsCompatible(Cons,AConsTmp,c) succeeds) {
            AConsTmp = AconsTmp ∪ c
        } else {
            ExpCov_ns = ExpCov_ns − pt(c) /* pt(c) ∈ ExpCov_ns */
            RedCov_ns = RedCov_ns − pt(c) /* pt(c) ∈ RedCov_ns */
        }
    }
    Cons_ns = Cons_ns ∪ AconsTmp
    foreach (transition t in OnCov) {
        /* if the product-terms in RedCov_ns cover t */
        if (sharp(t, RedCov_ns) returns empty) {
            OnCov = OnCov − t
            DcCov = DcCov + t
        }
    }
}
```

Figure 5.6: Symbolic reduction - Part1

procedure SymbReducePart2($IniCov$,$PartCov$,$ExpCov_{ns}$,$RedCov_{ns}$,$Cons$,$Cons_{ns}$) {
do { /* piece-wise minimizations of what left in $OnCov$ */
$OnCovTmp = \emptyset$; $DcCovTmp = \emptyset$
/* choose greedily a maximal on-set */
foreach (transition t in $OnCov$) {
$OffCovTmp$ = complement($OnCovTmp \cup t$, $DcCovTmp$)
/* invoke espresso with no makesparse */
$ExpCovTmp$ = espresso($OnCovTmp \cup t$,$DcCovTmp$,$OffCovTmp$)
$RedCovTmp$ = mv_reduce($ExpCovTmp$,$DcCovTmp$)
$AConsTmp$ = Constraints($IniCov$, $ExpCovTmp$, $RedCovTmp$)
if (ConstraintsCompatible($Cons$,$AConsTmp$) succeeds) {
$OnCovTmp = OnCovTmp \cup t$
$OnCov = OnCov - t$
$SaveExpCovTmp = ExpCovTmp$; $SaveRedCovTmp = RedCovTmp$
$SaveAConsTmp = AConsTmp$
}
}
/* choose greedily a maximal dc-set of previous on-set */
foreach (transition t in $DcCov$) {
$OffCovTmp$ = complement($OnCovTmp$, $DcCovTmp \cup t$)
/* invoke espresso with no makesparse */
$ExpCovTmp$ = espresso($OnCovTmp$,$DcCovTmp \cup t$,$OffCovTmp$)
$RedCovTmp$ = mv_reduce($ExpCovTmp$,$DcCovTmp$)
$AConsTmp$ = Constraints($IniCov$, $ExpCovTmp$, $RedCovTmp$)
if (ConstraintsCompatible($Cons$,$AConsTmp$) succeeds) {
$DcCovTmp = DcCovTmp \cup t$
$SaveExpCovTmp = ExpCovTmp$; $SaveRedCovTmp = RedCovTmp$
$SaveAConsTmp = AConsTmp$
}
}
$Cons_{ns} = Cons_{ns} \cup SaveAConsTmp$
$ExpCov_{ns} = ExpCov_{ns} \cup SaveExpCovTmp$;
$RedCov_{ns} = RedCov_{ns} \cup SaveRedCovTmp$
} while (at least one transition in $OnCov$)
}

Figure 5.7: Symbolic reduction - Part2

5.5 Symbolic Oring

In two-level logic minimization of multi-output functions the fact of sharing cubes among single outputs reduces the cardinality of the cover. When minimizing symbolic logic to obtain minimal encodable two-level implementations, one should detect the most profitable disjunctive constraints so that - after encoding - sharing of cubes is maximized. In Section 5.3 an example was given where *oring* in the output part accounts for most savings in the minimum cover. In the symbolic minimization loop presented in Section 5.3, $SymbOring$ is invoked to that purpose.

The goal of the procedure *SymbOring* is to determine a subset (if it exists) of the transitions of Fc_{ns} that can be realized using the product-terms of the partial minimized symbolic cover $(ExpCov, RedCov)$. If so, that subset is moved from the on-set to the dc-set of the cover to minimize in the current step. The procedure is heuristic because it handles a transition of Fc_{ns} at a time and it introduces some approximations with respect to an exact computation. For each transition t of Fc_{ns} the following algorithm decides whether t can be realized using or modifying product-terms in $RedCov$. Here we present the main features, leaving out minor design choices.

At a certain step of the procedure *symbolic* a pair of partial covers ($ExpCov$,$RedCov$) is available. For each cube $pexp \in ExpCov$ there is a companion cube $pred \in RedCov$ (and viceversa) such that $pred$ is obtained by $pexp$ by applying to it the multiple-valued *reduce* routine of ESPRESSO. A cube $pred \in RedCov$ potentially useful to espress implicitly t must satisfy the conditions that its input part (denoted $I(pred)$) has non-empty intersection with $I(t)$ and the output part of t (denoted $O(t)$) covers $O(pred)$. All such cubes are collected in the cover $Inter(t)$. It may happen that $I(pred)$ does not intersect $I(t)$, but that $I(pexp)$ intersects $I(t)$, because in $pred$ the bit of the present state of t is lowered, while in $pexp$ it is raised. If so, one may raise temptatively also the bit in $pred$ to obtain another potentially useful cube that is added to $Inter(t)$. The product-term $pred$ raised in the present state of t is denoted by $raised(pred)_t$ [3].

The set $OrNstates(Inter(t))$ of next states of cubes in $Inter(t)$ is computed. Define $Inter(t)_S$ as the set of transitions of $Inter(t)$ with next state included in set S. In order that a disjunctive effect occurs it is necessary that, for at least two next states $s1$ and $s2$, $I(t)$ is covered both by the union of the input parts of all cubes in $Inter(t)_{s1}$ and by the union of the input parts of all cubes in $Inter(t)_{s2}$. Here covering is meant to be restricted to the next state function assumed

[3] In the current implementation p is not added to $Inter(t)$ if $I(p)$ is covered by the input part of another cube already in $Inter(t)$. The rationale is that product-terms whith a more expanded input part are preferred, because they are more likely to cover other transitions in the future. An exact algorithm should define the notion of don't-care intersecting product-terms, if one knows how to handle conditional dominance constraints.

as a single output. Suppose that $OrNstates$ has at least two elements. We determine the states s of $OrNstates$ such that the union of the input parts of the cubes in $Inter(t)_s$ covers $I(t)$, and discard the others. Moreover, in order that a disjunctive effect occurs it is necessary that, for all binary output functions, $I(t)$ is covered by the union of the input parts of all cubes in $Inter(t)$. If all previous tests are not satisfied, the attempt of expressing t by symbolic oring fails.

If the previous necessary conditions are satisfied, all subsets of elements in the set $OrNstates$ are computed in $Subset(OrNstates)$. Each such subset, denoted by or, is an oring pattern potentially useful to espress implicitly the transition t. For each oring pattern or, the procedure *OringCover* returns $OrCov(t)$, a subset of transitions of $Inter(t)_{or \cup \phi}$ (it means $Inter(t)$ restricted to next states in or or empty next state) that cover t, both in the next state output space and in the binary output spaces. Notice that *OringCover* may fail to find a cover even if it exists, because while the input space of the binary output functions can be covered by considering the whole $Inter(t)$, only a subset of it ($Inter(t)_{or \cup \phi}$) is considered by *OringCover*. Notice also that there may be many possible such covers, but only one is found. This may penalize the quality of the final results, because the computed cover may yield uncompatible constraints, while there is another cover that yields compatible constraints. We do not give the details of *OringCover*, that is based on a greedy strategy.

If a cover $OrCov(t)$ is found, one considers the modified partial minimized cover $RedCovTmp$, obtained from $RedCov$ by raising the present state bits according to what done in the generation of $Inter(t)$. Then the constraints implied by the modified cover are derived and checked for compatibility with the oring constraint or (since some product-terms of $RedCov$ have been raised in the present state, there are raised face constraints and by consequence dominance constraints must be recomputed). If the answer is positive, the transition t is implementable by oring and both $RedCov$ and $Cons$ are updated. Otherwise a new oring pattern from $Subset(OrNstates)$ is considered. When they have been all exhausted, a new transition of Fc_{ns} is taken into consideration. [4].

The outlined procedure is shown in Figures 5.8. The routines with initial letter in the lower case are directly available in ESPRESSO (not necessarily with the same name and syntactic usage), while the routines with initial letter in the upper case are new.

[4] A better alternative would be to check for constraints compatibility while building $OrCov(t)$: do not add a new product-term to the subset of $OrCov(t)$ currently accepted, if together with it, it yields infeasible constraints .

procedure SymbOring($IniCov$,$ExpCov$,$RedCov$,$Cons$) {
  foreach (transition $t \in Fc_{ns}$) {
    foreach (pair of product-terms $(pred, pexp) \in (RedCov, ExpCov)$) {
      if ($I(pred) \cap I(t)$ non-empty and $O(t) \supseteq O(pred)$) {
        $Inter(t) = Inter(t) \cup pred$
      } else {
        if ($I(pexp) \cap I(t)$ non-empty and $O(t) \supseteq O(pexp)$) {
          $Inter(t) = Inter(t) \cup raised(pred)_t$
        }
      }
    }
    compute $OrNstates(Inter(t))$
    if (at least two states in $OrNstates$) {
      foreach (next state $s \in OrNstates$)
        if ($\bigcup_{p \in Inter(t)_s} I(p) \not\supseteq I(t)$) $OrNstates = OrNstates - s$
      foreach (binary output function)
        if ($\bigcup_{p \in Inter(t)} I(p) \not\supseteq I(t)$) $OrNstates$ empty
    }
    if (at least two states in $OrNstates$) {
      generate $Subset(OrNstates)$
      foreach (element or of $Subset$) {
        $OrCov(t)$ = OringCover($Inter(t)_{or \cup \phi}$,t,$ExpCov$,$RedCov$)
        if ($OrCov(t)$ is not empty) {
          $RedCovTmp$ = Raise($RedCov$,$Inter(t)$,t)
          $ConsTmp$ = Constraints($IniCov$,$ExpCov$,$RedCovTmp$)
          if (ConstraintsCompatible($ConsTmp$,or) succeeds) {
            $Or_{ns} = Or_{ns} \cup t$
            $RedCov = RedCovTmp$
            $Cons = ConsTmp \cup or$
            goto outer foreach loop
          }
        }
      }
    }
  }
}

Figure 5.8: Symbolic oring

5.6 Ordering of Symbolic Minimization

In the procedure *symbolic* described in Section 5.3, at each cycle of the symbolic minimization loop, states are partitioned in two sets: those selected in previous iterations (Sel) and those still unselected ($Ns - Sel$). At the start of a new cycle, a new state ns is selected by the procedure *SelectState* from $Ns - Sel$ and the state partition is updated.

The transitions of the FSM are partitioned, accordingly, in the transitions asserting the states in Sel and already minimized and the transitions asserting the states in $Ns - Sel$ and not yet minimized. We observe the following facts:

1. When a new state ns is selected, the transitions asserting it cannot be used later to minimize the transitions asserting states in $Ns - Sel - ns$. Therefore if one measures how much an unselected state can help in minimizing the other unselected states by dominance ($DomGain$), the state of minimum gain should be selected first.

2. When a new state ns is selected, the transitions asserting it cannot be espressed later using the transitions asserting states in $Ns - Sel - ns$. Therefore if one measures how much the minimization of an unselected state is helped by the other unselected states by oring ($OrGain$), the state of minimum gain should be selected first.

Summarizing, the problem of the best selection of a new state can be reduced to one of measuring the dominance and oring gains and then choosing the state that minimizes their sum ($TotGain = DomGain + OrGain$).

As an example, consider that $Ns = st0, st1, st2, st3, st4, st5, st6$. Suppose that currently $st0$, $st5$, $st6$ have been already selected and that a new state must be chosen among $st1$, $st2$, $st3$, $st4$, by computing their gain and choosing the minimum. We have devised two slightly different schemes for computing the gain of a state. In the first scheme, the gain of a state, for instance $st1$, can be computed by setting up a minimization as shown in Figure 5.9 (in the figure the covers are shown for the next state functions asserted by the unselected states). After the minimization, the difference in cardinality between the resulting and original covers gives one component of the gain, $DomGain$ (associated to the dominance constraints: $st1 > st2$, $st1 > st3$, $st1 > st4$). The second component of the gain, $OrGain$ (associated to the disjunctive constraints: $st1 = st2 \vee st3 \vee st4$, $st1 = st2 \vee st3$, $st1 = st2 \vee st4$, $st1 = st3 \vee st4$), is found by computing, for each transition asserting $st1$, whether its input part is covered by the input parts of the transitions asserting at least two other unselected states, for the related next state functions and all binary output functions.

OnCov:
on-set of st2 0010000
on-set of st3 0001000
on-set of st4 0000100
OffCov:
on-set of st2 0001100
on-set of st3 0010100
on-set of st4 0011000
on-set of st0 0011100
on-set of st5 0011100
on-set of st6 0011100
DcCov:
on-set of st1 0011100

Figure 5.9: First scheme to compute the gain

In the second scheme, the gain of a state can be computed by setting up a minimization as shown in Figure 5.10 (referring again to $st1$ in the previous example). After the minimization, the difference in cardinality between the resulting and original covers gives the overall gain $TotGain$, inclusive of both the dominance and disjunctive components.

The pseudo-code in Figure 5.11 shows the first scheme to compute the gain. The second one is simpler, since it does not include explicitly the covering check to measure the oring contribution (that is implicitly taken into account by the minimization process) and it is not shown here.

5.7 Satisfaction of Encoding Constraints

The described procedures require algorithms to check satisfiability of a set of face, dominance and disjunctive constraints, and to find minimum codes that satisfy them. We used the algorithms reported in [116], to which we refer for a complete description. They are based on the notion of encoding dichotomies that are candidate encoding columns. The notion of encoding dichotomy was pioneered in [143] and the connection with satisfaction of face constraints was

OnCov:
on-set of st2 0010000
on-set of st3 0001000
on-set of st4 0000100
on-set of st1 0011100
OffCov:
on-set of st2 0001100
on-set of st3 0010100
on-set of st4 0011000
on-set of st0 0011100
on-set of st5 0011100
on-set of st6 0011100

Figure 5.10: Second scheme to compute the gain

established in [154]. Other contributions on the subject can be found in [126, 20] and more recently in [44, 45].

5.8 Symbolic Minimization by Example

In this section we clarify with an example the mechanics by which the oring effects plays an important role in the minimization of symbolic logic. Then we demonstrate our algorithm for symbolic minimization on a simple example.

5.8.1 The Oring Effect in Two-level Logic

In two-level logic minimization of multi-output functions the fact of sharing cubes among single outputs reduces the cardinality of the cover. As an example, consider the following cover of a logic function of four input and four output variables:

```
1000 0100
0100 0001
1100 0101
0001 1000
1001 1100
```

procedure SelectState($UnSel$) {
    foreach (state $st \in UnSel$) {
        $gain(st)$ = ComputeGain(st,$UnSel$)
    }
    sel = $st \in UnSel$ with minimum $gain(st)$
}
procedure ComputeGain($IniCov$,st,$UnSel$) {
    /* measure potential gains by dominance */
    $OnCov$ = $\bigcup_{i \in (UnSel - st)} Fc_i$
    $OldCard$ = #$(OnCov)$
    foreach (state $j \in UnSel - st$)
        $OffCov_j$ = $\bigcup_{i \in UnSel - j - st} On_i \cup \bigcup_{i \in Ns - UnSel} On_i$
    $OffCov$ = $(\bigcup_{j \in UnSel - st} OffCov_j) \cup Off_{bo}$
    $DcCov$ = complement($OnCov$,$OffCov$)
    /* invoke espresso with no makesparse */
    $OnCov$ = espresso($OnCov$,,$DcCov$,$OffCov$)
    $DomGain = OldCard -$ #$(OnCov)$
    /* measure potential gains by oring */
    foreach (transition $t \in Fc_{st}$) {
        foreach (state $i \in UnSel - st$) {
            $OnCov_i$ = product-terms of $OnCov$ asserting next state i
            if ($I(t) \subseteq I(OnCov_i)$ for next state and binary output functions) {
                increment $OrCount$
                if ($OrCount > 1$) { /* t can be expressed by oring */
                    increment $OrGain$
                    goto outer foreach loop
                }
            }
        }
    }
    $TotGain = DomGain + OrGain$
}

Figure 5.11: Ordering of symbolic minimization

```
0101 1001
1101 1101
0010 0010
1010 0110
0110 0011
1110 0111
0011 1010
1011 1110
0111 1011
1111 1111
```

and an equivalent minimum cover, as found by ESPRESSO:

```
---1 1000
1--- 0100
--1- 0010
-1-- 0001.
```

Consider the product term 1001 1100 that appears in the original cover. In the minimum cover, when the input cube 1001 is true, the first two product terms of the minimum cover are excited and the output part 1100 is asserted. Therefore the product term 1001 1100 is implemented by means of the product terms $---1$ 1000 and $1---$ 0100. Notice that two product terms must be in any cover to realize the following product terms of the original cover 1000 0100 and 0001 1000. Therefore a net saving of one product term (the one needed to realize 1001 1100) has been achieved in the minimum cover. We say that the product term 1001 1100 has been realized by oring or disjunctive effect (due to the semantics of the output part of a two-level implementation) or that it has been redistributed through the two product terms $---1$ 1000 and $1---$ 0100. The oring effect accounts for most savings in the minimum cover of this example.

5.8.2 A Worked-out Example of Symbolic Minimization

This subsection contains an example of symbolic minimization. The example is *shiftreg* from the MCNC suite. The symbolic cover of *shiftreg*, using the syntax of ESPRESSO, is:

```
.mv 4 1 -8 -8 1
.type fr
.kiss
0 st0 st0 0
1 st0 st4 0
0 st1 st0 1
1 st1 st4 1
```

```
0 st2 st1 0
1 st2 st5 0
0 st3 st1 1
1 st3 st5 1
0 st4 st2 0
1 st4 st6 0
0 st5 st2 1
1 st5 st6 1
0 st6 st3 0
1 st6 st7 0
0 st7 st3 1
1 st7 st7 1
```

Suppose that the ordering routine returned $st0, st4, st1, st2, st5, st3, st6, st7$ as the order in which the slices of next states must be minimized. Let each position in the 1-hot encoded notation correspond respectively to the states $st0, st4, st1, st2, st5, st3, st6, st7$. For instance 10000000 represents $st0$, while 01000000 represents $st4$. Slices including all the transitions that have the same next state are minimized in the given order. The result of each minimization is a set of symbolic cubes which realize the slice. A dc-set as specified by the theory is provided in each minimization. If terms of the dc-set having a different next state are used in a minimization, then covering constraints are introduced, together with companion face constraints (face constraints not related to output constraints can be introduced also, when transitions having the same next state are merged). Before each minimization, the algorithm figures out whether some transitions of the given slice can be realized by symbolic cubes already in the partial minimized symbolic cover, when a satisfiable oring constraint is imposed. Only the remaining transitions are kept in the onset of the slice under minimization. Whenever symbolic cubes that impose constraints on the codes are added to the cover, their consistency with respect to the constraints cumulated up to then is verified. As long as the consistency verification fails, different symbolic cubes are tried; eventually an encodeable symbolic cover is constructed. At the end codes of minimum code-length that satisfy the constraints are found and the codes are replaced in the symbolic cover and in the original FSM (it is not necessary, but convenient to do both, because don't cares can be used differently, producing covers not of the same cardinality). A final step of two-valued minimization produces a minimal encoded FSM.

- Minimization of the slice of next state $st0$.

  The onset is:

```
0 10000000 100000000
0 00100000 100000001
```

The dcset is:

```
1 11000000 100000000
- 01010010 100000000
1 00100000 111111111
- 00001101 111111111
- 11111111 011111110
```

The minimized expanded cover is:

```
- 11111111 111111110
- 00101101 111111111
```

The minimized reduced cover is:

```
- 11111111 100000000
- 00100000 000000001
```

The constraints $code(st4) > code(st0)$, $code(st1) > code(st0)$, $code(st2) > code(st0)$, $code(st5) > code(st0)$, $code(st3) > code(st0)$, $code(st6) > code(st0)$ and $code(st7) > code(st0)$ are introduced. The companion face constraints are trivial.

- Minimization of the slice of next state $st4$.

  The onset is:

```
1 10000000 010000000
1 00100000 010000001
```

  The dcset is:

```
- 01010010 010000000
0 00100000 000000001
- 00001101 111111111
- 11111111 101111110
```

  The minimized expanded cover is:

```
- 00101101 101111111
1 11111111 111111110
```

The minimized reduced cover is:

```
- 00100000 000000001
1 11111111 010000000
```

The constraints $code(st5) > code(st4)$, $code(st6) > code(st4)$ and $code(st7) > code(st4)$ are introduced. The companion face constraints are trivial.

- Minimization of the slice of next state $st1$.

  The onset is:

```
0 00010000 001000000
0 00000100 001000001
```

  The dcset is:

```
- 01000010 001000000
- 00100000 000000001
1 00010110 001000000
1 00000100 111111111
- 00001001 111111111
- 11111111 110111110
```

  The minimized expanded cover is:

```
- 00101101 110111111
- 01011111 111111110
```

  The minimized reduced cover is:

```
- 00000100 000000001
- 00010100 001000000
```

  The constraints $code(st5) > code(st1)$ and $face(st2, st3)$ are introduced.

- Minimization of the slice of next state $st2$.

  The onset is:

```
onset
0 01000000 000100000
0 00001000 000100001
```

The dcset is:

```
1 01011110 000100000
- 00100100 000000001
1 00001100 111111111
- 00000010 000100000
- 00000001 111111111
- 11111111 111011110
```

The minimized expanded cover is:

```
- 00101101 111011111
- 01001011 111111110
```

The minimized reduced cover is:

```
- 00001000 000000001
- 01001000 000100000
```

The constraints $code(st6) > code(st2)$ and $face(st4, st5)$ are introduced.

- Minimization of the slice of next state $st5$.
  The transitions of this slice are realized by oring symbolic cubes previously added to the cover, if one introduces the constraint $code(st5) = code(st4) \vee code(st1)$.

- Minimization of the slice of next state $st3$.
  One of the two transitions of this slice is realized by oring symbolic cubes previously added to the cover, if one introduces the constraint $code(st3) = code(st1) \vee code(st2)$. Consider the remaining transition.
  The onset is:

```
0 00000001 000001001
```

The dcset is:

```
1 01000011 000001000
- 00101100 000000001
1 00001001 111111111
- 00000010 000001000
- 11111111 111110110
```

The minimized expanded cover is:

```
- 00000001 111111111
```

The minimized reduced cover is:

```
- 00000001 000001001
```

The constraint $code(st7) > code(st3)$ is introduced.

- Minimization of the slice of next state $st6$.
  The transitions of this slice are realized by oring symbolic cubes previously added to the cover, if one introduces the constraint $code(st6) = code(st4) \vee code(st2)$.

- Minimization of the slice of next state $st7$.
  One of the two transitions of this slice is realized by oring symbolic cubes previously added to the cover, if one introduces the constraint $code(st7) = code(st4) \vee code(st1) \vee code(st2)$.
  Consider the remaining transition.
  The onset is:

```
onset
1 00000010 000000010
```

The dcset is:

```
- 00101101 000000001
1 00000001 111111111
- 11111111 111111100
```

The minimized expanded cover is:

```
1 00000011 111111110
```

The minimized reduced cover is:

```
1 00000010 000000010
```

No other constraint is introduced.

- Minimization of the slice of the proper binary outputs.
  The onset is:

```
- 00101101 000000001
- 00100000 000000001
- 00000100 000000001
- 00001000 000000001
```

The dcset is:

```
- 11111111 111111110
```

The minimized expanded cover is:

```
- 00101101 111111111
```

The minimized reduced cover is:

```
- 00101101 000000001
```

The constraint $face(st1, st5, st3, st7)$ is introduced.

- The final symbolic cover is:

```
- 11111111 100000000
1 11111111 010000000
- 00010111 001000000
- 01001011 000100000
- 00000001 000001001
1 00000010 000000010
- 00101101 000000001
```

Codes of the states that satisfy the previous constraints are: $code(st0) = 000$, $code(st4) = 010$, $code(st1) = 100$, $code(st2) = 001$, $code(st5) = 110$, $code(st3) = 101$, $code(st6) = 011$, $code(st7) = 111$. The minimized encoded symbolic cover is:

```
---1 1000
1--- 0100
--1- 0010
-1-- 0001
```

The minimized encoded FSM is:

```
---1 1000
1--- 0100
--1- 0010
-1-- 0001
```

5.9 Experimental Results

The algorithms described have been implemented in a program, called ESP_SA, that is built on top of ESPRESSO. We report one set of experiments that compare the results of performing state assignments of FSM's with ESP_SA and NOVA, a state-of-art tool. The FSM's come from the MCNC suite and other benchmarks. The experiments were run on a DEC 3100 work-station. Our program ESP_SA uses a library of routines described in [116] to check encodeability of constraints and produce minimum-length codes that satisfy them. Table 5.1 shows the statistics of the FSM's used. The statistics include the number of states, proper inputs and proper outputs, together with the number of symbolic produc-terms ("#cubes") of the original FSM description, the cardinality of a minimized 1-hot encoded cover of the FSM ("#1-hot") and the number of bits for an encoding of minimum length ("#bits").

In Table 5.2, data are reported for runs of ESP_SA with three different ordering options ("ord1", "ord2", "ord2n"). For each run, "#scubes" indicates the number of cubes of the cover of symbolic cubes obtained by ESP_SA, after encoding with the codes found by ESP_SA and minimization with ESPRESSO; "#cubes" indicates the number of cubes after encoding the original cover with the codes found by ESP_SA and minimization with ESPRESSO; "#bits" indicates the length of the codes found by ESP_SA.

In Table 5.3, some data related to the best of the three previous runs are reported. Under "cover", "#incomp" gives the number of pairwise incompatibilities in the final step of computing codes the satisfy the encoding constraints, and "size" gives the number of prime dichotomies. Under "calls", "#esp" gives the number of calls to ESPRESSO and "#check" gives the number of encodeability checks. Under "CPU times(sec.)", "order" gives the time in seconds for the ordering routine, "symb" gives the time for symbolic minimization, not including the time spent by the encodeability routines that is reported under "constr(enc)" ("enc" is the time spent for finding the codes satisfying the constraints at the end), while "total" sums up all the contributions.

Table 5.4 compares the results of ESP_SA with those of NOVA, a state-of-art state assignment tool, providing the number of cubes of the minimized encoded FSM ("#cubes") and the code-length ("#bits"). Of the results by NOVA, it is reported the one the minimizes the final cover cardinality (under the heading "NOVA(min.#cubes)") and the one that minimizes the final cover cardinality, if the code-length is kept to the minimum one, i e. to the logarithm of the number of states (under the heading "NOVA(min.#bits)").

example	#states	#inputs	#outputs	#cubes	#1-hot	#bits
bbara	10	4	2	60	34	4
bbsse	16	7	7	56	30	4
bbtas	6	2	2	24	16	3
beecount	7	3	4	28	12	3
cse	16	7	7	91	55	4
dk14	7	3	5	56	25	3
dk15	4	3	5	32	17	2
dk17	8	2	3	32	20	3
dk27	7	1	2	14	10	3
dk512	15	1	3	30	21	4
donfile	24	2	1	96	24	5
ex1	20	9	19	138	44	5
ex2	19	2	2	72	38	5
ex3	10	2	2	36	21	4
ex4	14	6	9	21	21	4
ex5	9	2	2	32	19	4
ex6	8	5	8	34	23	3
ex7	10	2	2	36	20	4
keyb	19	7	2	179	77	5
kirkman	16	12	6	370	61	4
lion9	9	2	1	25	10	4
maincont	16	11	4	40	27	4
mark1	15	5	16	22	19	4
master	15	23	31	86	79	4
modulo12	12	1	1	24	24	4
opus	10	5	6	22	19	4
ricks	13	10	23	51	33	4
s1	20	8	6	107	92	5
s1a	20	8	6	107	92	5
s8	5	4	1	20	14	3
saucier	20	9	9	32	30	5
scud	8	7	6	127	8	3
shiftreg	8	1	1	16	9	3
slave	10	16	29	75	46	4
train11	11	2	1	25	11	4

Table 5.1: Statistics of FSM's

example	ord1			ord2			ord2n		
	#scubes	#cubes	#bits	#scubes	#cubes	#bits	#scubes	#cubes	#bits
bbara	27	27	5	31	28	6	24	23	5
bbsse	31	31	6	26	26	7	24	24	8
bbtas	10	9	3	10	10	4	11	11	4
beecount	10	10	4	12	12	6	10	10	4
cse	58	55	7	42	42	5	42	42	5
dk14	26	27	4	27	27	4	26	26	4
dk15	17	17	4	17	17	4	17	17	4
dk17	19	17	5	19	17	5	19	19	6
dk27	7	7	5	9	8	5	7	7	5
dk512	19	18	7	18	16	9	15	15	8
donfile	26	25	12	25	25	13	26	25	12
ex1	37	36	9	42	40	9	42	40	9
ex2	34	35	10	36	32	12	30	31	9
ex3	20	18	6	21	18	7	17	17	6
ex4	14	14	5	15	15	5	14	14	5
ex5	17	16	9	18	18	6	14	13	4
ex6	25	25	4	26	25	4	26	25	4
ex7	20	20	8	20	18	4	15	15	5
keyb	75	65	9	45	46	6	47	47	5
kirkman	102	74	11	54	53	10	55	54	9
lion9	8	7	6	9	8	5	9	8	6
maincont	12	12	8	14	14	7	13	13	9
mark1	17	18	6	17	17	6	17	17	6
master	69	68	5	70	68	5	70	69	5
modulo12	22	22	11	20	20	10	22	22	11
opus	15	15	4	15	15	4	15	15	4
ricks	29	29	4	30	30	4	30	30	4
s1	62	59	6	49	44	7	49	44	7
s1a	62	61	11	61	61	13	60	60	9
s8	11	9	4	11	10	4	11	10	4
saucier	24	23	6	25	24	8	22	22	6
scud	70	63	7	68	65	8	68	65	8
shiftreg	4	4	3	4	4	3	4	4	3
slave	39	39	5	35	35	4	35	35	4
train11	10	9	5	13	12	6	10	9	5

Table 5.2: Results of ESP_SA with different ordering heuristics

example	cover		calls		CPU times (sec.)			
	#incomp	size	#esp	#check	order	symb	constr(enc)	total
bbara	38	8	96	173	7.4	12.9	0.6(0.1)	20.8
bbsse	458	168	155	46	41.6	10.4	4.3(0.8)	56.4
bbtas	9	4	30	80	1.0	0.9	0.2(0.0)	2.1
beecount	104	15	66	55	2.5	1.7	0.3(0.0)	4.5
cse	1170	629	155	80	99.9	45.6	19.6(7.9)	145.1
dk14	316	186	38	29	7.1	2.3	1.9(0.5)	11.3
dk15	256	238	17	19	1.8	0.4	1.2(0.5)	3.4
dk17	30	14	47	24	5.1	1.4	0.7(0.0)	7.2
dk27	1	2	38	30	0.9	0.6	0.1(0.0)	1.7
dk512	1	2	138	140	11.1	32.7	2.8(0.0)	46.6
donfile	17929	2701	432	1254	98.9	2044.0	143.4(117.1)	2286.3
ex1	2282	815	410	542	794.8	759.0	39.0(10.8)	1592.7
ex2	3934	826	212	1161	37.3	1493.7	28.2(21.4)	1559.2
ex3	148	14	68	52	3.4	3.7	0.7(0.1)	7.8
ex4	1048	359	122	22	15.9	5.0	3.5(2.8)	24.4
ex5	285	27	57	46	3.1	2.7	0.4(0.1)	6.2
ex6	219	16	47	29	8.8	1.7	0.9(0.1)	11.4
ex7	352	34	68	43	6.5	3.4	0.8(0.2)	10.7
keyb	967	1094	212	71	129.9	76.7	32.3(27.6)	239.0
kirkman	716	84	155	1164	1385.8	1187.5	172.8(3.6)	2746.1
lion9	26	7	86	75	5.4	2.6	0.4(0.0)	8.4
maincont	363	55	194	196	34.5	48.6	2.5(0.5)	85.5
mark1	443	112	247	155	44.7	42.8	2.1(0.8)	89.5
master	281	300	327	315	271.0	240.5	18.6(3.0)	530.1
modulo12	45	10	93	2358	4.8	227.2	1.3(0.2)	233.3
opus	312	151	68	18	6.4	1.7	0.9(0.6)	9.0
ricks	353	408	107	60	53.4	19.3	7.7(3.9)	80.4
s1	969	288	233	92	253.1	126.9	10.9(4.8)	390.9
s1a	225	67	317	639	151.3	661.8	13.0(2.8)	826.1
s8	6	4	46	103	1.7	1.3	0.3(0.0)	3.4
saucier	1401	3340	256	124	45.0	99.1	157.2(156.2)	301.3
scud	70	11	457	1011	59.4	228.9	12.3(0.3)	300.5
shiftreg	3	3	47	54	1.3	1.0	0.1(0.0)	2.5
slave	229	132	68	41	39.1	8.2	3.8(0.5)	51.1
train11	156	23	105	86	9.5	5.2	0.4(0.1)	15.1

Table 5.3: Measured parameters of ESP_SA

example	ESP_SA		NOVA(min.#cubes)		NOVA(min.#bits)	
	#cubes	#bits	#cubes	#bits	#cubes	#bits
bbara	23	5	24	4	24	4
bbsse	24	8	27	5	29	4
bbtas	9	3	8	3	8	3
beecount	10	4	10	3	10	3
cse	42	5	44	5	45	4
dk14	26	4	22	4	25	3
dk15	17	4	16	3	17	2
dk17	17	5	17	4	17	3
dk27	7	5	7	3	7	3
dk512	15	8	17	4	17	4
donfile	25	12	24	14	28	5
ex1	36	9	37	6	44	5
ex2	31	9	26	6	27	5
ex3	17	6	17	4	17	4
ex4	14	5	14	4	14	4
ex5	13	4	14	4	14	4
ex6	25	4	23	4	25	3
ex7	15	5	15	4	15	4
keyb	46	6	47	6	48	5
kirkman	53	10	52	6	77	4
lion9	7	6	8	4	8	4
maincont	12	8	16	4	16	4
mark1	17	6	17	4	17	4
master	68	5	71	4	71	4
modulo12	20	10	11	4	11	4
opus	15	4	15	4	15	4
ricks	29	4	39	4	30	4
s1	44	7	63	5	63	5
s1a	60	9	65	5	65	5
s8	9	4	9	3	9	3
saucier	22	6	25	5	25	5
scud	63	7	60	5	62	3
shiftreg	4	3	4	3	4	3
slave	35	4	35	4	35	4
train11	9	5	9	4	9	4

Table 5.4: Comparison of FSM's encodings for two-level implementation

5.10 Conclusions

We have presented a symbolic minimization procedure that advances theory and practice with respect to the seminal contribution in [91]. The algorithm described here is capable of exploring minimal symbolic covers by using face, dominance and disjunctive constraints to guarantee that they can be mapped into encoded covers. The treatment of disjunctive constraints is a novelty of this work. Conditions on the completness of sets of encoding constraints and a bridge to disjunctive-conjunctive constraints (presented in [39]) are given.

A key feature of the algorithm is that it keeps as invariant the property that the minimal symbolic cover under construction is encodeable, by means of efficient procedures that check encodeability of the encoding constraints induced by a candidate cover. Therefore this synthesis procedure has predictive power that precedent tools lacked, i.e. the cardinality of the cover obtained by symbolic minimization and of the cover obtained by replacing the codes in the initial cover and then minimizing with ESPRESSO are very close. Experiments show cases where our procedure improves on the best results of state-of-art tools.

A direction of future investigation is to explore more at large the solution space of symbolic covers by escaping from local minima using some iterated expansion and reduction scheme, as it is done in ESPRESSO. Currently the algorithm builds a minimal symbolic cover, exploring basically a neighborhood of the original FSM cover. Another issue requiring more investigation is how to predict somehow the final code-length while building a minimal symbolic cover, to trade-off product-terms vs. encoding length.

Chapter 6

Encoding Constraints

6.1 Introduction

The various techniques for exact and heuristic encoding based on multiple-valued or symbolic minimization of two-level and multi-level logic, reported in [92, 91, 115, 39, 85, 18], produce various types of encoding constraints. By encoding constraints we mean requirements on the codes to be assigned to the symbols.

A first type are *face-embedding* constraints generated by the multiple-valued input variables (*input* constraints). These constraints specify that a set of symbols is to be assigned to one *face* of a binary *n*-dimensional cube, without any other symbol sharing the same face. Given symbols a, b, c, d, e, an input constraint involving symbols a, b and c is denoted by (a, b, c). An encoding satisfying (a, b, c) is given by $a = 111, b = 011, c = 001$. Vertex 101 cannot be assigned to any other symbol.

Two other types are *dominance* and *disjunctive* constraints generated by the multiple-valued output variables (*output* constraints). A dominance constraint, denoted by $>$, e.g., $a > b$ requires that the code of a symbol bit-wise covers the code of another symbol. A disjunctive constraint specifies that the code of a symbol (the *parent* symbol) is the bit-wise disjunction, denoted by $\vee$, e.g., $a = b \vee c$, of the codes of two or more other symbols (the *children* symbols).

The minimization procedure described in [39] produces *disjunctive-conjunctive* constraints. They require that the code of a symbol is the bit-wise disjunction (denoted by $\vee$) of the *conjunctions*, denoted by $\wedge$), of the codes of two or more symbols. An example is: $(a \wedge b \wedge c) \vee (a \wedge d \wedge e) \vee (a \wedge f \wedge g) = a$. An in-depth discussion of disjunctive-conjunctive constraints is postponed to the chapter on encodeability of generalized prime implicants.

An example containing input, dominance and disjunctive constraints is: (b, c), (c, d), (b, a), (a, d), $b > c$, $a > c$, $a = b \vee d$. An encoding satisfying all constraints with minimum code length of two is $a = 11, b = 01, c = 00, d = 10$.

In this chapter, we focus on the following problems. Given a set of encoding constraints:

P-1: Determine whether the constraints are satisfiable.

P-2: Determine the binary codes that use a minimum number of bits and satisfy all the constraints.

P-3: Using a fixed number of code bits, minimize a cost function of the constraints that are not satisfiable.

Previous work on encoding constraint satisfaction has dealt mostly, but not exclusively, with input constraints. Exact algorithms and efficient heuristics (restricted to input and dominance constraints) for solving problems P-2 and P-3 are reported in [147]. An approximate solution to P-3 for input constraints based on a theory of intersecting cubes is described in [126, 43] and a solution based on simulated annealing is reported in [81]. An exact solution to P-2 for input constraints based on the notion of prime sections is described in [44, 45]. This approach seems very promising because of the claim that prime sections are fewer than prime dichotomies, the latter being the building blocks of encodings in the theory that we are going to use in this chapter. An approximate solution to P-2 and P-3 for input constraints based on a greedy strategy to find an encoding bit by bit and on an iterative method to improve the obtained solution is reported in [129]. The answer to Problem P-1 is always affirmative for input constraints only. A solution to P-1 and a heuristic algorithm to solve P-3 when both input and output dominance constraints occur are provided in [91], extending an algorithm for input constraints described in [92]. A solution to P-1, when both input and output constraints (including disjunctive constraints) are present, is described in [39], and corrected in [38]. A solution to problem P-2 based on compatible graph coloring is provided for input constraints in [154] and extended to output constraints in [20]. To date, to the best of our knowledge, no efficient algorithms exist for solving all three problems when all types of constraints occur. In most previous contributions, techniques to generate constraints and to satisfy them were intermixed. Instead, we concentrate only on the problem of satisfying encoding constraints.

We propose a framework and efficient algorithms to solve P-1, P-2 and P-3 for input and output encoding constraints. A polynomial time algorithm to answer P-1, and, exact and heuristic algorithms to solve P-2 and P-3 are provided. We solve P-3 with different cost functions, such as the number of constraints satisfied and the number of cubes or literals required in the encoded

implementation. These algorithms also handle *encoding don't cares* [91, 85] and can be easily extended to other types of constraints. We also prove the NP-completeness of problems P-2 and P-3. This result has not been shown previously, though it has been conjectured [91].

The approach used here is based on a formulation provided in [154], which in turn is related to the state assignment technique employed by Tracey in 1966 [143]. We first demonstrate the difficulty of finding codes that satisfy encoding constraints by proving it NP-complete in Section 6.2. Section 6.3 provides some definitions. In Section 6.4 an abstraction of the problem is presented. In Section 6.5 we describe a new algorithm to satisfy input constraints only. This is extended to handle input and output constraints in Section 6.6. This includes a polynomial time algorithm for checking the satisfiability of a set of encoding constraints. A heuristic algorithm is sketched in Section 6.7. Extensions of the framework to handle various types of constraints and cost functions are discussed in Section 6.8. Experimental results are provided in Section 6.9. Section 6.10 concludes the chapter.

6.2 Statement and Complexity of the Encoding Problems

In this section we formally state Problem P-2 both as a decision and an optimization problem and show that the decision (optimization) version with input constraints alone is NP-complete (NP-hard). We will show later that Problem P-1 can be solved by a polynomial time algorithm.

A few preliminary definitions are required. An *n-dimensional hypercube* (or n-cube) is a graph of 2^n vertices labeled uniquely by the integers from 0 and $2^n - 1$. An edge joins two vertices whose binary representations of their integer labels differ by exactly one bit. The minimum k-cube that contains a given subset of vertices of a n-cube ($k \leq n$) is the k-face (or smallest face) spanned by the given vertices.

Decision version of P-2:

Instance: Set of input and output constraints defined on a set of symbols S, and a positive integer k.

Question: Is there a function f from S to the vertices of a k-cube such that:

1. symbols in the same input constraint are mapped to vertices spanning a face that does not contain the image of any other symbol, and

2. the binary labels of the images of the symbols satisfy the output constraints?

Optimization version of P-2:

Instance: Set of input and output constraints defined on a set of symbols, S.

Objective: Find the minimum k-cube and a function f from S to the vertices of the k-cube such that:

1. symbols in the same input constraint are mapped to vertices spanning a face that does not contain the image of any other symbol, and
2. the binary labels of the images of the symbols satisfy the output constraints.

Answering the decision version for different dimensions repeatedly solves the optimization problem (with a polynomial number of calls to the decision procedure) and, of course, solving the optimization problem answers the decision version for all dimensions. Clearly, by assigning a weight of *1* to each constraint, P-2 can be seen as a special case of P-3. Hence, P-3 is no easier than P-2.

The decision version of P-2 with input constraints alone is defined as *face hypercube embedding*. To prove that face hypercube embedding is NP-complete a few more preliminaries are needed.

A given graph $G = (V, E)$ is a subgraph of an n-cube if there is a function mapping vertices of G into vertices of the n-cube that preserves the adjacency relations. G can be embedded in an n-cube if G is a subgraph of the n-cube. The problem of deciding whether a given graph is embeddable into an arbitrary dimension hypercube has been shown to be NP-complete [71]. It has also been proved that even the problem of deciding whether a graph can be embedded into a fixed-size hypercube is NP-complete [31]. The proof in [31] actually shows that the problem of determining whether a graph of 2^k nodes can be embedded in a k-cube is NP-complete. This result can be used to prove that face hypercube embedding is NP-complete.

Theorem 6.2.1 *Face hypercube embedding is NP-complete.*

Proof: Face hypercube embedding is in NP. Consider the set of positions P where all the codes in a given constraint agree (this set must not be empty unless a constraint involves all symbols). The codes of the symbols not in that constraint must differ in at least one position of P from the codes of the symbols in that constraint. This can checked in time linear in the product of the number of constraints, number of symbols and integer k.

Suppose k is the dimension of the hypercube into which the face constraints composed of symbols from set S must be embedded. Let us restrict face hypercube embedding to the instances where the symbols involved in the face constraints are 2^k and each face constraint involves only two symbols. For these instances it is possible to define a graph $G(V, E)$ induced by the face constraints. The set of nodes V is in correspondence with the symbols in S and there is an edge between two nodes when the two corresponding symbols are in the same face constraint. The set of face constraints can be embedded into a k-cube if and only if the companion graph is a subgraph of a k-cube. Notice that in this case the concept of face embedding reduces to the familiar notion of graph adjacency. The problem of determining whether a graph of 2^k nodes is a subgraph of a k-cube is NP-complete by reduction from 3-partition [31]. Therefore the problem of determining whether for 2^k symbols a set of face constraints each with exactly two symbols can be embedded into a k-cube is NP-complete. But this is a restricted version of face hypercube embedding and hence the latter is NP-complete. ■

6.3 Definitions

An **encoding dichotomy** (or, more simply, **dichotomy**) is a 2-block partition of a subset of the symbols to be encoded. The symbols in the left block are associated with the bit *0* while those in the right block are associated with the bit *1*. If an dichotomy is used in generating an encoding, then one code bit of the symbols in the left block is assigned *0* while the same code bit is assigned *1* for the symbols in the right block.

For example, $(s_0 s_1; s_2 s_3)$ is a dichotomy in which s_0 and s_1 are associated with the bit *0* and s_2 and s_3 with the bit *1*. This definition of dichotomy differs from the one in [143, 154], which allows the left block of a dichotomy to assume either the encoding bit *0* or *1*, and it is equivalent to the definition of **fixed dichotomy** given in [20].

A dichotomy is **complete** if each symbol appears in either block. A **completion** of a dichotomy (l, r) is a dichotomy (l', r') such that $l' \supseteq l$, $r' \supseteq r$, and each symbol appears exactly once in either l' or r'.

Two dichotomies d_1 and d_2 are **compatible** if the left block of d_1 is disjoint from the right block of d_2 and the right block of d_1 is disjoint from the left block of d_2. Otherwise, d_1 and d_2 are incompatible. Note again that this definition differs from the definition of compatibility described in [143, 154]. The **union** of two compatible dichotomies, d_1 and d_2, is the dichotomy whose left and right blocks are the union of the left and right blocks of d_1 and d_2 respectively. The union

operation is not defined for incompatible dichotomies. A dichotomy d_1 **covers** a dichotomy d_2 if the left and right blocks of d_2 are subsets respectively either of the left and right blocks, or of the right and left blocks of d_1. For example, $(s_0; s_1 s_2)$ is covered by $(s_0 s_3; s_1 s_2 s_4)$ and $(s_1 s_2 s_3; s_0)$, but not by $(s_0 s_1; s_2)$. A **prime dichotomy** of a given set of dichotomies is one that is incompatible with all dichotomies not covered by it.

A set of complete dichotomies generates an encoding as follows. Each complete dichotomy generates one column of the encoding, with symbols in the left (right) block assigned a *0* (*1*) in that column. For example, given the complete dichotomies $(s_0 s_1; s_2 s_3)$ and $(s_0 s_3; s_1 s_2)$, the unique encoding derived is $s_0 = 00, s_1 = 01, s_2 = 11$, and $s_3 = 10$.

A dichotomy **violates** an output constraint if the encoding bit generated for the symbols in the dichotomy does not satisfy the bit-wise requirement for the output constraint. A **valid dichotomy** is one that does not violate any output constraint. For example, the dichotomy $(s_0; s_1 s_2)$ violates the constraint $s_0 > s_1$, since s_0 is assigned bit *0* whereas s_1 is assigned bit *1* by this dichotomy. Hence, s_0 does not cover s_1 in this bit. The dichotomy $(s_0 s_1; s_2)$ does not violate this constraint.

6.4 Abstraction of the Problem

Satisfaction of encoding constraints may be abstracted as a *binate covering problem* (BCP) [113].

Suppose that a set $S = \{s_1, \ldots, s_n\}$ is given. The cost of 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, BCP can be defined as finding $S' \subseteq S$ that minimizes

$$\sum_{i=1}^{n} c_i x_i, \tag{6.1}$$

subject to the constraint

$$A(x_1, x_2, \ldots, x_n) = 1, \tag{6.2}$$

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. BCP is the problem of finding 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, A can be written as an array of cubes (that form a matrix with coefficients from the set $\{0, 1, 2\}$). Each variable of A denotes a column, and each sum

(or clause) denotes a row. 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.

Suppose that symbols a, b, c and three constraints $(a, b), b > c, b = a \vee c$ are given. An *encoding column* is a column vector whose i-th component is a bit (i.e. a 0 or 1) assigned to the i-th symbol. All possible encodings can be represented as sets of encoding columns. The column encodings for the example are: $c_1 = 001, c_2 = 010, c_3 = 011, c_4 = 100, c_5 = 101, c_6 = 110$, where the order of symbols in a column is a, b, c[1]. Since each symbol in a column is either assigned 1 or 0, a column partitions the symbols into a 1-block and a 0-block. For example, $c_6 = 110$ places a and b in the 1-block and c in the 0-block. For each face constraint consider the encoding dichotomies that have the symbols of the face constraint in one block, and have one of the remaining symbols in the other block [154]. In the example, this is $(ab; c)$ or $(c; ab)$. This means that by covering either $(ab; c)$ or $(c; ab)$, the face constraint (a, b) is satisfied. Add the encoding dichotomies expressing the uniqueness of the codes; these are $(a; b)$ or $(b; a)$, $(a; c)$ or $(c; a)$, $(b; c)$ or $(c; b)$. Build a table whose columns are the encoding columns and whose rows are the encoding dichotomies. A column covers a row representing an encoding dichotomy if the symbols of each block of the dichotomy are in the same partition of the column. For example, $c_6 = 110$ covers $(c; a)$ since c is set to 0 and a is set to 1, but does not cover $(a; b)$ because a and b are both set to 1. Likewise $c_6 = 110$ covers $(c; b)$. Put a 1 in entry (i, j) if column j covers row i. For each output constraint, add a row for each encoding column that cannot be chosen if that output constraint must be satisfied and put a 0 in the corresponding entry. In the example, $b > c$ yields two rows, one has a 0 in column c_1, the other has a 0 in column c_5. One could imagine more complex types of constraints that add rows carrying two or more 0's and no 1's to denote that all the columns with a 0 in them cannot be simultaneously selected. The binate table for the example is shown in Figure 6.1.

A minimum column cover of the given rows gives a minimum set of encoding columns that satisfy all given constraints. This requires the solution of a binate covering problem. However, the problem reduces to a unate covering problem when only face constraints are present [143].

[1] 000 and 111 are excluded because they do not carry useful information.

		c1	c2	c3	c4	c5	c6
a;b	b;a		1	1	1	1	
a;c	c;a	1		1	1		1
c;b	b;c	1	1			1	1
ab;c	c;ab	1					1
	b>c	0					
	b>c					0	
	b=a+c	0					
	b=a+c		0				
	b=a+c				0		
	b=a+c					0	
	?			0	0		

Figure 6.1: Satisfaction of encoding constraints using binate covering

Although BCP offers a unified framework for solving encoding constraints, the design of efficient algorithms requires exploiting specific features of the problems at hand. In the sequel we demonstrate this fact by developing exact and heuristic algorithms.

6.5 Input Constraint Satisfaction

We first present a new algorithm for satisfying input encoding constraints that, compared to previous approaches [147, 154], significantly improves the efficiency of the input encoding process.

The encoding constraint satisfaction problem is a three-step process. The first is the generation of the dichotomies that represent the face embedding constraints [154]. Each face embedding constraint generates several dichotomies, called **initial dichotomies**. The symbols that are to be on a face are placed in one block of each dichotomy representing that constraint, while the other block contains one of the symbols not on the face. Thus, for n symbols $s_1, s_2, ..., s_n$ and a face embedding constraint that requires the l symbols $s_1, s_2, .., s_l$ to be on one face, we generate $2\ (n-l)$ dichotomies each with the symbols $s_1, s_2, ..., s_l$ in one block (either left or right) and exactly one of the remaining $n-l$ symbols in the other block. Notice that initial dichotomies are generated in pairs, for instance, given the symbols s_1, s_2, s_3, s_4 and the face constraint (s_1, s_3), the

initial dichotomies $(s_1, s_3; s_2)$, $(s_2; s_1, s_3)$, and $(s_1, s_3; s_4)$, $(s_4; s_1, s_3)$ are generated. Sometimes we say that dichotomy $(s_2; s_1, s_3)$ is the **dual** of dichotomy $(s_1, s_3; s_2)$ and viceversa.

These dichotomies exactly capture the face embedding constraints. We also require that each symbol get a distinct code. This is represented by a dichotomy with one symbol in each block. When there are n symbols and no encoding constraints, the number of uniqueness constraints is $n^2 - n$; these would generate an exponential number ($2^n - 2$) of prime dichotomies. We need to add only those uniqueness constraints that are not covered by the dichotomies generated from the face-embedding constraints, because any encoding that satisfies the covering dichotomy satisfies also the covered dichotomy.

The second step of encoding is the generation of prime dichotomies from the dichotomies. [143] describes an approach similar to the process of *iterated consensus* for prime generation in two-level logic minimization [11]. However, the number of iterations required to generate all the prime dichotomies may be formidable even for small problems. Using this approach, several different compatible merges often yield the same prime dichotomy. This results in a substantial waste of computation time [154]. In Section 6.5.1, we describe a method of generating all prime dichotomies and demonstrate its effectiveness in determining an exact solution.

The final step of encoding is to obtain a cover of the initial dichotomies using a minimum number of primes. This is a classical unate covering problem and efficient branch and bound techniques, both for exact and heuristic solutions, are well known [113].

6.5.1 Efficient Generation of Prime Dichotomies

By definition, each prime dichotomy is a *maximal compatible* of the dichotomies since it is not compatible with any dichotomy that it does not cover. As in [86], an incompatibility between two dichotomies represented by the literals a and b, is written as $(a + b)$. When the product of the sum terms representing all the pairwise incompatibilities is written as an irredundant sum-of-products expression, a maximal compatible is generated as the union of those dichotomies whose literals are missing in any product term [86]. For example, assume that we wish to find the maximal compatibles for five dichotomies, a, b, c, d, e. Assume that the incompatibilities are $(a+b)(a+c)(b+c)(c+d)(d+e)$. Then the equivalent irredundant sum-of-products expression is $abd + acd + ace + bcd + bce$. The five primes are then formed by the unions of the missing literals: $\{c, e\}, \{b, e\}, \{b, d\}, \{a, e\}, \{a, d\}$.

The problem is how to efficiently derive the equivalent sum-of-products expression from

the product-of-sums expression representing the incompatibilities. In the past, this has been performed using an approach based on Shannon decomposition [153]:

$$f(x_1, \cdots, x_i, \cdots, x_n) = x_i \cdot f(x_1, \cdots, 1, \cdots, x_n) + \overline{x_i} \cdot f(x_1, \cdots, 0, \cdots, x_n)$$

Basically one splits on a variable at a time and generates recursively two subproblems. The complexity of performing the recursive Shannon expansion is exponential since a binary tree is constructed. We describe an algorithm that can generate all the primes, but only uses a linear number of operations in the size of the output.

The product-of-sums expressions previously generated have two features:

1. Each clause has exactly two literals;

2. Literals appear only in the positive phase, i.e., the function is unate.

By exploiting these properties it is possible to simplify the algorithm based on Shannon expansion. The algorithm is described in pseudo-code in Fig. 6.2. Given a product-of-sums expression, a splitting variable, *x*, is chosen. Since all clauses have exactly two literals in the positive phase, the product of all sum terms containing *x*, call it *x_expr*, after simplification, consists of two terms, the first is *x* alone and the second is the product of all the other variables in *x_expr*. Therefore a recursive call is needed only for the product of the sum-terms in the initial expression that do not contain *x*, called *reduced_expr*. The two product terms, *x_expr* and *cs(reduced_expr)*, are multiplied and single cube-containment is used to obtain the minimum sum-of-products expression. Again single cube-containment can be used to find the minimum expression since the function is unate [11].

This algorithm replaces exponential (in the number of dichotomies) calls as required in the worst-case by a Shannon expansion based approach by a linear number of them. Of course, the runtime of the algorithm is still proportional to the final number of primes, which may be exponential (in the number of dichotomies).

The example in Fig. 6.3 illustrates the complete input encoding process. A set of input constraints is shown and the corresponding initial dichotomies are derived. The maximal compatibles are generated by a procedure *cs* that recurs on the splitting variable. Variable 0 is chosen as first splitting variable. The procedure returns the minimal product *ps* of the following two expressions: the first is the product of all sum terms containing 0 (in this case simplified into 0 and 234567) and the second is the result of the recursive call of the procedure *cs* on the sum terms that do not contain 0. By minimal product it is meant that the two expressions, when available after a series of recursive

calls, are multiplied out and then single cube-containment is performed on them. Once the maximal compatibles are found, the prime dichotomies are easily obtained and a standard unate covering routine produces a minimum subset of primes that cover all given initial dichotomies. Notice that to simplify the example we have forced the symbol s_1 to be always in a right block. This reduces the number of prime dichotomies but does not affect the solution to the input encoding problem[2].

6.6 Input and Output Constraint Satisfaction

6.6.1 Output Encoding Constraints

A dominance constraint $a > b$, requires that the encoding for a bit-wise covers the encoding for b. This means that any dichotomy chosen in the final cover cannot have a in the left block while b is in the right block. Hence, any dichotomy that has this property may be deleted from consideration.

A disjunctive constraint $a = b \vee c$, implies that the encoding for symbol a must be the same as the bit-wise *or* of the encodings of b and c. This means that any dichotomy in a feasible solution must have at least one of b and c appear in the same block as a. Any dichotomy that does not possess this property may be deleted. This property is easily extended to the case where the disjunctive constraint involves more than two symbols or has nested conjunctive constraints.

A preliminary algorithm follows from the discussion above. In the first step, the dichotomies corresponding to the input constraints are generated. Next the prime dichotomies are generated using the algorithm described in Section 6.5.1; those that violate any of the dominance or disjunctive constraints are eliminated. Finally, the remaining dichotomies are used in selecting a minimum cover of all the initial dichotomies representing the input constraints. If there is at least one initial dichotomy that cannot be covered, then there is no solution.

This procedure may be used to answer two questions. The first is whether a feasible encoding exists for a set of input and output constraints. The second is to find the minimum length encoding satisfying the constraints, if it exists. An obvious drawback of this method is that many prime dichotomies may be generated but later deleted since they violate output constraints. We present an efficient algorithm that avoids the generation of useless prime dichotomies.

[2]In general, this symmetry cannot be exploited when there are both input and output constraints.

```
/* Given pairwise incompatibilities among a list of dichotomies
as a product-of-sums expression generate all prime dichotomies.
Each sum term has two literals and there are n variables,
each corresponding to a distinct initial dichotomy. */

/* Convert 2-CNF to sum-of-products expression
O(n) recursive calls */
procedure cs (expr) {
    x = splitting variable
    C = all sum terms with variable x
    reduced_expr = expr without the sum-terms in C
    x_expr = sum-of-product expression of C
    return (ps (x_expr, cs(reduced_expr)))
}

/* Obtain the product of two expressions.
expr1 has 2 terms, where the first term is a single variable */
procedure ps (expr1, expr2) {
    product_expr = product of expr1 and expr2
    result_expr = single_cube_containment (product_expr)
    return (result_expr)
}

procedure prime_dichotomy_generate (expr) {
    result = cs (expr)
    foreach (term T in result)
        missing = list of variables not in T
        new_prime_dichotomy = union of dichotomies corresponding to missing
        add new_prime_dichotomy to prime_list
    return (prime_list)
}
```

Figure 6.2: Efficient generation of prime dichotomies

$Constraints$	(s_0, s_2, s_4)	(s_0, s_1, s_4)	(s_1, s_2, s_3)	(s_1, s_3, s_4)
$Initial\ dichotomies$	$0 : (s_0 s_2 s_4; s_1)$	$1 : (s_3; s_0 s_2 s_4)$	$2 : (s_3; s_0 s_1 s_4)$	$3 : (s_2; s_0 s_1 s_4)$
	$4 : (s_0; s_1 s_2 s_3)$	$5 : (s_4; s_1 s_2 s_3)$		
	$6 : (s_0; s_1 s_3 s_4)$	$7 : (s_2; s_1 s_3 s_4)$		

$Deriving\ maximal\ compatibles\ (prime\ dichotomies)$

$cs(\ (0+2)(0+3)(0+4)(0+5)(0+6)(0+7)(1+3)(1+4)(1+5)(1+6)(1+7)(2+4)(2+5)(2+6)(2+7)(3+4)(3+5)(3+6)(4+7)(5+6)(5+7)\ )$

$ps(\ (0+234567), cs(\ (1+3)(1+4)(1+5)(1+6)(1+7)(2+4)(2+5)(2+6)(2+7)(3+4)(3+5)(3+6)(4+7)(5+6)(5+7)\ )\ )$

$ps(\ (0+234567), ps(\ (1+34567), cs(\ (2+4)(2+5)(2+6)(2+7)(3+4)(3+5)(3+6)(4+7)(5+6)(5+7)\ )\ )\ )$

$ps(\ (0+234567), ps(\ (1+34567), ps(\ (2+4567), cs(\ (3+4)(3+5)(3+6)(4+7)(5+6)(5+7)\ )\ )\ )\ )$

$ps(\ (0+234567), ps(\ (1+34567), ps(\ (2+4567), ps(\ (4+7), cs(\ (3+4)(3+5)(3+6)(5+6)(5+7)\ )\ )\ )\ )\ )$

$ps(\ (0+234567), ps(\ (1+34567), ps(\ (2+4567), ps(\ (4+7), ps(\ (3+456), cs(\ (5+6)(5+7)\ )\ )\ )\ )\ )\ )$

$ps(\ (0+234567), ps(\ (1+34567), ps(\ (2+4567), ps(\ (4+7), ps(\ (3+456), (5+67)\ )\ )\ )\ )\ )$

$ps(\ (0+234567), ps(\ (1+34567), ps(\ (2+4567), ps(\ (4+7), (35+367+456)\ )\ )\ )\ )$

$ps(\ (0+234567), ps(\ (1+34567), ps(\ (2+4567), (345+357+367+456)\ )\ )\ )$

$ps(\ (0+234567), ps(\ (1+34567), (2345+2357+2456+2367+4567)\ )\ )$

$ps(\ (0+234567), (12345+12357+12456+12367+14567+34567)\ )$

$(012345+012357+012456+012367+034567+014567+234567)$

$Maximal\ compatible\ sets$	$\{6,7\}$	$\{4,6\}$	$\{4,5\}$	$\{3,7\}$
	$\{2,3\}$	$\{1,2\}$	$\{0,1\}$	
$Prime\ dichotomies$	$(s_0 s_2; s_1 s_3 s_4)$	$(s_0; s_1 s_2 s_3 s_4)$	$(s_0 s_4; s_1 s_2 s_3)$	$(s_2; s_0 s_1 s_3 s_4)$
	$(s_2 s_3; s_0 s_1 s_4)$	$(s_3; s_0 s_1 s_2 s_4)$	$(s_0 s_2 s_4; s_1 s_3)$	
$Minimum\ cover$	$(s_0 s_2 s_4; s_1 s_3)$	$(s_2 s_3; s_0 s_1 s_4)$	$(s_0 s_4; s_1 s_2 s_3)$	$(s_0 s_2; s_1 s_3 s_4)$

Figure 6.3: Input encoding example

6.6.2 Satisfiability of Input and Output Constraints

We motivate the constraint satisfaction procedure using the example in Fig. 6.4. Given the encoding constraints, 26 initial dichotomies are obtained. Consider the initial dichotomies $(s_0; s_1s_5)$ and $(s_1s_5; s_0)$ that are generated from the face embedding constraint (s_1, s_5). Since $s_0 > s_1$, the dichotomy $(s_0; s_1s_5)$ is not allowed and is deleted from consideration. The dichotomy $(s_1s_5; s_0)$ is valid and will be used in a feasible encoding. Consider the dichotomy $(s_1; s_2s_5)$. If this dichotomy is to be expanded to a valid prime dichotomy, symbol s_0 is forced to be in the right block, since $s_0 > s_2$. Also, since $s_1 > s_3$, s_3 must be in the left block and since $s_4 > s_5$, s_4 is forced into the right block. Thus, all valid dichotomies covering this initial encoding dichotomy must cover the "raised" dichotomy $(s_1s_3; s_0s_2s_4s_5)$. Similarly, we obtain the six raised dichotomies shown. On generating the prime dichotomies from these raised dichotomies, we obtain five primes.

A dichotomy is **raised** by adding symbols into either its left or right block as implied by the output constraints. For example, the dichotomy $(s_0; s_1s_2)$ may be raised to the dichotomy $(s_0s_3; s_1s_2)$. A dichotomy is said to be **maximally raised** if no further symbols can be added into either the left or right block by the output constraints. The procedure *raise-dichotomy* in Fig. 6.6 describes an algorithm that maximally raises a dichotomy with respect to a set of output constraints.

When the problem is to determine if a set of constraints is satisfiable, we do not have to generate the prime dichotomies. Instead we use the set of maximally raised valid dichotomies, which are far fewer in number than the prime dichotomies, and we merely check if all the initial dichotomies are covered by the maximally raised and valid dichotomies.

An algorithm to check for the satisfiability of input and output constraints is shown in Figs. 6.5, 6.6 and 6.7. The following example shows why the second call to *remove_invalid_dichotomies* in Fig. 6.7 is needed. Consider the constraints $(a; bc)$, $d = b + c$ and $a > d$. After *raise_dichotomy* the following dichotomy is generated $(ad; bc)$ (by constraint $a > d$), but $(ad; bc)$ is an invalid dichotomy because it conflicts with constraint $d = b+c$. So a new pass of *remove_invalid_dichotomies* is required to delete it. Alternatively, we could suppress in Fig. 6.7 the first call to *raise_dichotomy*, and leave only the second one.

Since the raising of each dichotomy is performed in time linear in the number of symbols times the number of initial dichotomies, the running time of the algorithm is polynomial in the number of symbols and constraints.

$Face\ embedding\ constraints:$

(s_1, s_5) (s_2, s_5) (s_4, s_5)

$Dominance\ constraints:$

$s_0 > s_1$ $s_0 > s_2$ $s_0 > s_3$

$s_0 > s_5$ $s_1 > s_3$ $s_2 > s_3$

$s_4 > s_5$ $s_5 > s_2$ $s_5 > s_3$

$Disjunctive\ constraints:$

$s_0 = s_1 \vee s_2$

$Initial\ dichotomies:$

$(s_0; s_1s_5)$ $(s_1s_5; s_0)$ $(s_0; s_2s_5)$

$(s_2s_5; s_0)$ $(s_0; s_4s_5)$ $(s_4s_5; s_0)$

$(s_1; s_2s_5)$ $(s_2s_5; s_1)$ $(s_1; s_4s_5)$

$(s_4s_5; s_1)$ $(s_2; s_1s_5)$ $(s_1s_5; s_2)$

$(s_2; s_4s_5)$ $(s_4s_5; s_2)$ $(s_3; s_1s_5)$

$(s_1s_5; s_3)$ $(s_3; s_2s_5)$ $(s_2s_5; s_3)$

$(s_3; s_4s_5)$ $(s_4s_5; s_3)$ $(s_4; s_1s_5)$

$(s_1s_5; s_4)$ $(s_4; s_2s_5)$ $(s_2s_5; s_4)$

$(s_0; s_3)$ $(s_3; s_0)$

$Maximally\ raised\ dichotomies:$

$(s_1s_3; s_0s_2s_4s_5)$ $(s_2s_3; s_0s_1s_4s_5)$ $(s_2s_3s_4s_5; s_0s_1)$

$(s_0s_1s_2s_3s_5; s_4)$ $(s_2s_3s_5; s_0s_1)$

$(s_2s_3s_5; s_4)$

$Uncovered\ initial\ dichotomies:$

$(s_0; s_1s_5)$

$(s_1s_5; s_0)$

Figure 6.4: Example of feasibility check with input and output constraints

```
/* S is the set of symbols to be encoded */
procedure remove_invalid_dichotomies (D, constraints) {
    foreach (dichotomy d ∈ D)
        /* to handle dominance constraints */
        foreach (pair of symbols s, m ∈ S)
            if (s > m & s in left block & m in right block)
                delete d
        /* to handle disjunctive constraints */
        foreach (disjunctive constraint)
            if (parent in left block & at least one child in right block)
                delete d
            if (parent in right block & all children in left block)
                delete d
        /* to handle disjunctive-conjunctive constraints */
        foreach (extended disjunctive-conjunctive constraint)
            if (parent in right block & one child of each conjunction in left block)
                delete d
}
```

Figure 6.5: Removal of invalid dichotomies

```
/* d is a valid dichotomy */
procedure raise_dichotomy (d, constraints) {
    do {
        /* to handle dominance constraints */
        foreach (symbol s ∈ S)
            if (s in left block & s > m)
                insert m into left block of d
            if (s in right block & m > s)
                insert m into right block of d
        /* to handle disjunctive constraints */
        foreach (parent symbol s in a disjunctive constraint)
            if (all children in left block)
                insert s into left block
            if (all children but one child c in left block & s in right block)
                insert child c into right block
        /* to handle disjunctive-conjunctive constraints */
        foreach (parent s in a disjunctive-conjunctive constraint e)
            if (one child of each conjunction in left block)
                insert s into left block
            if (one child of all but one conjunction in left block & s in right block)
                insert all children of remaining conjunction into right block
    } while (at least one insertion within loop)
}
```

Figure 6.6: Maximal raising of dichotomies

```
procedure check_feasible (constraints) {
    I = generate_initial_dichotomies (constraints)
    D = remove_invalid_dichotomies (I, constraints)
    foreach (dichotomy d in D)
        raise_dichotomy (d, constraints)
    D = remove_invalid_dichotomies (D, constraints)
    foreach (dichotomy i ∈ I)
        if i is not covered by some d ∈ D
            return (INFEASIBLE)
    return (FEASIBLE)
}
```

Figure 6.7: Feasibility check of input and output constraints

We now prove that the feasibility algorithm is correct.

Theorem 6.6.1 *Given a set of input and output constraints, let I be the set of initial dichotomies generated from the input constraints, including all uniqueness constraints that are not already covered by an initial dichotomy. Let each valid dichotomy in I be maximally raised to obtain a set of valid dichotomies D. A dichotomy that becomes invalid on raising is deleted from D. The input and output constraints are satisfiable if and only if each $i \in I$ is covered by some $d \in D$.*

Proof: **If Part** Consider a valid maximally raised dichotomy $d = (L_1; R_1)$, where L_1 and R_1 are disjoint subsets of the symbols to be encoded. Consider a symbol $s \notin d$. There are no output constraints that either require any of the symbols in L_1 to cover s, or s to cover any of the symbols in R_1. Otherwise d is not raised maximally. Add all symbols $F = \{s : s \notin d\}$, to the right block of d. There may be output constraints among the symbols in F, but these are satisfied since all the symbols in F are inserted into the right block. Repeat the same operation of adding uncommitted symbols to the right block for all valid maximally raised dichotomies. Call the dichotomies so obtained complete, because every symbol appears in either block of each of them. A valid encoding exists by deriving the codes from any set of complete valid maximally raised dichotomies that cover all initial dichotomies in I.

Only If Part Assume that some initial dichotomy $i \in I$ is not covered by any of the dichotomies in D. It means that there is no $d \in D$ that contains block-wise i or the dual of i. At the start of *check_feasible* i was put in D, unless removed by the first call of *remove_invalid_dichotomies*. Then

it was raised to $d(i)$ and it must have been removed by the second call to *remove_invalid_dichotomies*, otherwise $d(i) \in D$ would contain block-wise i. Suppose that there exists a dichotomy p whose right and left blocks contain the right and left blocks of i. Then the conditions that caused invalidity of i or caused raising and then invalidity of $d(i)$ are satisfied also for p (it can be seen by case analysis of the conditions of *remove_invalid_dichotomies* and *raise_dichotomy*) and so either p is invalid or it can be maximally raised to an invalid $d(p)$. Therefore there can be no valid dichotomy that contains block-wise i.

Similar reasoning holds for the dual dichotomy of i (blocks are reversed), i.e. from the fact that the dual of i was initially put in D it is deduced that there can be no valid dichotomy that contains the dual of i. Therefore there can be no valid dichotomy that covers i, i.e., no feasible solution exists. ■

6.6.3 Exact Encoding of Input and Output Constraints

Once the feasibility check of a set of input and output constraints is passed, a problem is to find codes of minimum length that satisfy the constraints. If the requirement that codes are of minimum length is dropped, then it is sufficient to take the valid maximally raised dichotomies, make each of them complete by adding to the right block any state absent from the dichotomy and then choose a minimal set of complete maximally raised dichotomies that cover all initial dichotomies. By adding absent states to the right block no invalid dichotomy can be produced, since no removal or raising rule becomes applicable to the complete maximally raised dichotomies so obtained.

We will now discuss the case when codes of minimum length are wanted. An encoding column must be a complete and valid dichotomy. When there are input constraints only, the notions of valid prime dichotomies and of valid complete dichotomies coincide. In general, there are two ways of computing all valid complete dichotomies:

1. In *generate_initial_dichotomies* add all uniqueness constraints to I, as done in [116].

2. After the prime encoding dichotomies have been generated, make them complete, by adding in all possible ways the missing symbols to the right and left blocks.

We will adopt here the first option because it is more practical in this algorithmic frame.

An algorithm for satisfying both input and output constraints is shown in Fig. 6.8. Following the generation of the initial dichotomies from the input and uniqueness constraints, those that violate output constraints are deleted. The remaining dichotomies are raised maximally. Any

raised dichotomy that becomes invalid is deleted. If each of the initial dichotomies is covered by at least one of the valid and maximally raised dichotomies, all prime dichotomies are generated from the valid raised dichotomies and invalid dichotomies are removed again. Using an exact unate covering algorithm, a minimum cover of the initial dichotomies by valid prime dichotomies yields the exact solution.

The following example shows why the third call to *remove_invalid_dichotomies* in Fig. 6.8 is needed. Consider the symbols a, b, c, d, the uniqueness constraints $(a; b)$, $(b; a)$, $(a; c)$, $(c; a)$, $(a; d)$, $(d; a)$, $(b; c)$, $(c; b)$, $(b; d)$, $(d; b)$, $(c; d)$, $(d; c)$ and the disjunctive constraint $b = c + d$. The first call to *remove_invalid_dichotomies* removes $(b; c)$ and $(b; d)$. By raising, $(c; b)$ becomes $(c; bd)$ and $(d; b)$ becomes $(d; bc)$. By merging $(b; a)$ and $(c; d)$ the invalid prime dichotomy $(bc; ad)$ is obtained.

An example is given in Fig. 6.9. Notice that, given the initial dichotomies $(s_2; s_0s_1)$, $(s_0s_1; s_2)$, $(s_3; s_0s_1)$, $(s_0s_1; s_3)$, $(s_0; s_1)$, $(s_1; s_0)$, $(s_2; s_3)$ and $(s_3; s_2)$, the following are removed because they are invalid: $(s_0s_1; s_2)$ (it conflicts with $s_1 > s_2$), $(s_0s_1; s_3)$ (it conflicts with $s_0 = s_1 \vee s_3$) and $(s_0; s_1)$ (it conflicts with $s_0 > s_1$). By raising the remaining valid dichotomies one obtains the following raised dichotomies: $(s_1s_2; s_0s_3)$ (from the initial dichotomy $(s_1; s_0)$, since $s_1 > s_2$ forces s_2 into the left block and $s_0 = s_1 \vee s_3$ forces s_3 into the right block) that subsumes the valid dichotomy $(s_2; s_3)$, $(s_3; s_2s_1)$ (from $(s_3; s_2)$, since $s_1 > s_2$ forces s_1 into the right block), $(s_2; s_0s_1)$ and $(s_3; s_0s_1)$ (the last two are initial dichotomies unmodified by the raising process). Since each initial dichotomy is covered by some raised dichotomy, an encoding satisfying all constraints exists by Theorem 6.6.1. The prime dichotomies are $(s_2s_3; s_0s_1)$ (by merging $(s_2; s_0s_1)$ and $(s_3; s_0s_1)$), $(s_3; s_0s_1s_2)$ (by merging $(s_3; s_0s_1)$ and $(s_3; s_2s_1)$), $(s_1s_2; s_0s_3)$ and $(s_2; s_0s_1)$. Notice that the last dichotomy is not complete. i.e., s_3 does not appear in either block. The completions of $(s_2; s_0s_1)$ are $(s_2s_3; s_0s_1)$ and $(s_2; s_0s_1s_3)$. The first one had been already generated by merging, the second one replaces $(s_2; s_0s_1)$. Even though in the proposed algorithm in Fig. 6.8 we do not use the step of completions, but we add instead all uniqueness constraints to I, in the example we have preferred the former way for compactness of exposition.

Theorem 6.6.2 *Given a set of input and output constraints, let I be the set of initial dichotomies generated from the input constraints, including all uniqueness constraints. The algorithm shown in Fig. 6.8 generates codes of minimum length for a set of input and output constraints, if a solution exists.*

Proof: The proof is based on Theorem 6.6.1. If a solution exists, a minimum solution can be obtained

```
procedure exact_encode (constraints) {
    I = generate_initial_dichotomies (constraints)
    D = remove_invalid_dichotomies (I, constraints)
    foreach (dichotomy d ∈ D)
        raise_dichotomy (d, constraints)
    D = remove_invalid_dichotomies (D, constraints)
    foreach (dichotomy i ∈ I)
        if i is not covered by some d ∈ D
            return (INFEASIBLE)
    P = prime_dichotomy_generate (D)
    valid_primes = remove_invalid_dichotomies (P, constraints)
    mincov = minimum_cover (I, valid_primes)
    return (derive_codes (mincov))
}
```

Figure 6.8: Exact encoding constraint satisfaction

from the maximally raised and valid dichotomies by generating prime dichotomies, and then finding a minimum covering of the initial dichotomies. Notice that we require that all uniqueness constraints are in I to guarantee that no valid dichotomy is missed. It may be that a subset of the uniqueness constraints is sufficient to the purpose, but we do not explore the issue more. ■

6.7 Bounded Length Encoding

The solution of problem P-3 (*c.f.* Section 10.1) requires a fixed-length encoding that minimizes a cost function on the constraints. In practice, this problem is more relevant than problem P-2 which requires that all constraints be satisfied using a minimum number of encoding bits. The reason is the trade-off between the increased code-length and the area gain obtained by satisfying all constraints. For example, optimal encoding for finite state machines (FSM's) implemented by two-level logic may be viewed as the process of generating a set of mixed input and output constraints. Satisfying all the constraints may require an encoding whose length is greater than the minimum code-length. This translates into extra columns of the PLA, and may result in sub-optimal PLA area and performance. The same reasoning applies to multi-level logic, where

$Face\ embedding\ constraints:$

(s_0, s_1)

$Dominance\ constraints:$

$s_0 > s_1$ $\quad$ $s_1 > s_2$

$Disjunctive\ constraints:$

$s_0 = s_1 \vee s_3$

$Initial\ dichotomies:$

$(s_2; s_0 s_1)$ $\quad$ $(s_0 s_1; s_2)$ $\quad$ $(s_3; s_0 s_1)$

$(s_0 s_1; s_3)$ $\quad$ $(s_0; s_1)$ $\quad$ $(s_1; s_0)$

$(s_2; s_3)$ $\quad$ $(s_3; s_2)$

$Raised\ dichotomies:$

$(s_2; s_0 s_1)$ $\quad$ $(s_3; s_0 s_1)$ $\quad$ $(s_1 s_2; s_0 s_3)$

$(s_3; s_2 s_1)$

$Prime\ dichotomies:$

$(s_2; s_0 s_1)$ $\quad$ $(s_2 s_3; s_0 s_1)$ $\quad$ $(s_3; s_0 s_1 s_2)$

$(s_1 s_2; s_0 s_3)$

$Complete\ dichotomies:$

$(s_2; s_0 s_1 s_3)$ $\quad$ $(s_2 s_3; s_0 s_1)$ $\quad$ $(s_3; s_0 s_1 s_2)$

$(s_1 s_2; s_0 s_3)$

$Minimum\ cover:$

$(s_2 s_3; s_0 s_1)$ $\quad$ $(s_1 s_2; s_0 s_3)$

$Final\ encoding:$

$s_0 = 11, s_1 = 10, s_2 = 00, s_3 = 01$

Figure 6.9: Example of exact encoding with input and output constraints

literal counts are used instead of cubes. Therefore, logic synthesis applications require an encoding algorithm that:

- considers different cost functions; and,
- minimizes a chosen cost function for encodings of fixed length.

There are three cost functions that are useful in such applications:

- the number of constraints satisfied;
- the number of product-terms in a sum-of-product representation of the encoded constraints; and,
- the number of literals in a sum-of-product representation of the encoded constraints [85].

We illustrate the meaning and technique of computation of these cost functions with an example. Consider the following input constraints: (e, f, c), (e, d, g), (a, b, d), (a, g, f, d). To satisfy all the constraints, an encoding of 4 bits is required. A solution is $a = 1010$, $b = 0010$, $c = 0011$, $d = 1110$, $e = 0111$, $f = 1011$, $g = 1100$. Suppose instead that the code-length is fixed at 3 bits. Irrespective of which 3-bit encoding is chosen, it must be the case that one or more input constraints are not satisfied. This leads to the problem of estimating the "goodness" of each 3-bit encoding. For each input constraint I, define a Boolean function F_I whose on-set contains the codes of the symbols in the constraint and whose off-set contains the codes of the symbols not in the constraint. The unused codes are in the don't care set. For instance, given the previous encoding, the points in the on-set of $F_{(e,f,c)}$ are $(0111, 1011, 0011)$, those in the off-set are $(1010, 0010, 1110, 1100)$ while the don't care set contains the remaining unused nine codes. If constraint I is satisfied, two-level minimization of F_I yields a single product-term. If a constraint is not satisfied, there will be at least two product-terms in the minimized result. Thus, the number of product-terms after two-level minimization is a measure of the satisfaction of the input constraints. For constraints arising from encoding problems of two-level logic, this is an appropriate cost function. An algorithm based on this cost function may require a number of two-level logic minimizations. This may be approximated by a single logic minimization of a multi-output Boolean function, where each constraint is represented by a distinct output of the multiple-output function. The number of literals of a two-level implementation of the constraints can be computed in the same way; literals are counted instead of product-terms.

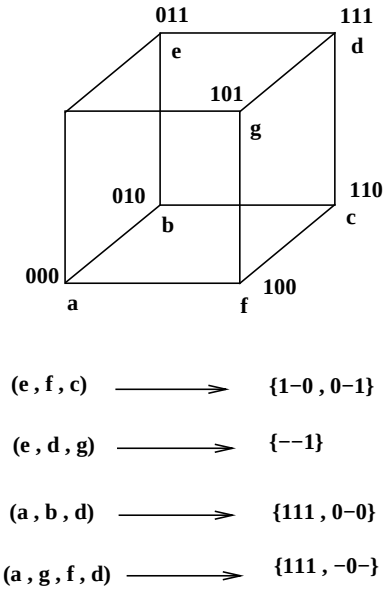


Figure 6.10: Example of cost function evaluation

In Figure 6.10, a 3-bit encoding for the previous set of constraints is shown, together with the product-terms needed to implement the encoded constraints. The given 3-bit encoding violates 3 face constraints. They are (e, f, c), (a, b, d), (a, g, f, d). 7 cubes and 14 literals are required to represent the encoded constraints.

6.7.1 Heuristic Algorithm for Input Constraints

Consider the input constraint satisfaction problem where an encoding of length c bits is desired, while minimizing the number of violated constraints. This is an exact version of problem P-3. We require a selection of prime dichotomies that must have two properties. First, the primes must ensure that each symbol gets a unique code, that is, all the uniqueness constraints must be covered by the selected primes. Second, the fewest face constraints must be violated. The only apparent way this can be done is to enumerate all $2^n - 2$ prime dichotomies (using n symbols) and then solve an exact weighted unate covering problem. This approach is clearly infeasible on all but trivial instances of P-3.

Heuristic algorithms can be easily developed within the encoding framework presented in this chapter. In this subsection we describe a heuristic algorithm based on the concept of dichotomies to solve P-3 approximately.

As indicated above, the first phase of an exact solution to problem P-3 involves the enumeration of all $2^n - 2$ prime dichotomies that exist for n symbols. This step is termed *candidate dichotomy generation* (or *candidate generation* in short). The second phase is to determine a selection of a fixed number of these encoding dichotomies that minimize the desired cost function. This is termed *selection*. While *candidate generation* is clearly exponential in the input size, the *selection* phase requires examination of a polynomial (in the code-length c) number of sets of candidate encoding dichotomies. A heuristic algorithm that avoids this enumeration of dichotomies while retaining the structure of the exact approach is detailed now.

Let $S = s_1, s_2, ..., s_n$ be a set of symbols and let D be a set of encoding dichotomies using these symbols. Consider some subset of symbols, $P = s_{p_1}, s_{p_2}, ..., s_{p_k}$. The **restricted dichotomies** of D with respect to P are the elements of the set D_P of dichotomies obtained by removing all symbols not in P from each dichotomy $d \in D$.

The algorithm has three main phases: *splitting* of a set of symbols, *merging* of restricted dichotomies and selection of the c *best* restricted dichotomies for a subset of symbols. The splitting phase is used to divide the given encoding problem into two smaller problems, each to be encoded using one less bit. Assuming that each sub-problem is solved optimally, the solution for the original encoding problem is generated by the steps of merging and selection.

Let a code of length c be desired for n symbols, $s_1, ..., s_n$. Consider a partition of the symbols into two groups $s_1, ..., s_k$ and $s_{k+1}...s_n$. Let D_1 be the $c - 1$ best dichotomies restricted to $s_1, ..., s_k$. Similarly, let D_2 be the $c - 1$ best dichotomies restricted to $s_{k+1}, ..., s_n$. Then, the candidate dichotomies for $s_1, ..., s_n$ is the set

$$D = \{(s_1...s_k; s_{k+1}...s_n)\} \cup (D_1 \times D_2) \cup (D_2 \times D_1).$$

The best selection of c dichotomies from D is used to obtain a desired encoding. By repeatedly applying this technique until each partition contains a single symbol, a bounded-length encoding is achieved following the merging and selection steps.

Splitting of symbols: We are interested in obtaining two sub-problems, each using one less code bit than the given problem does. In splitting the symbols into disjoint partitions, the fewest constraints should be violated. This is achieved by using a modification of the Kernighan-Lin [68] algorithm for partitioning[3].

[3]This step can also be performed by using the notion of incompatibility between dichotomies. The prime dichotomy that covers the maximum number of dichotomies is desired. Given the pairwise incompatibilities between dichotomies, this can be obtained by choosing the minimum cover of the pairwise incompatibilities (*cf.* Section 6.5.1). We do not

Each partition P can be considered as yielding a dichotomy, d_P. For example, the partition of n symbols into two blocks of symbols $\{s_1, ..., s_k\}$ and $\{s_{k+1}, ..., s_n\}$ gives $d_P = (s_1...s_k; s_{k+1}...s_n)$. The choice of partition P is determined by a cost function evaluation on the dichotomy d_P. For example, if the number of violated face constraints is to be minimized, then P is chosen such that the fewest face constraints are violated by d_P. If the number of literals (or cubes) is being minimized, then P is chosen such that the maximum number of restricted initial dichotomies are covered by d_P. This corresponds to minimizing the number of uncovered initial dichotomies. Thus, for the partitioning algorithm [68], the *nodes* are the symbols being partitioned and the *nets* are either face constraints or initial dichotomies.

The procedure is performed recursively on each resulting partition. Each partition again yields candidate dichotomies restricted to the subset of symbols that appear in it. When only two symbols remain, a single dichotomy that corresponds to the uniqueness constraint between them is generated.

Consider the example shown in Fig. 6.3, where an encoding of length 3 is required to minimize the number of literals in sum-of-product form. In the first step, at least four initial dichotomies must be violated by any partition. Assume that the symbols are partitioned into $P_1 = \{s_0, s_1, s_2, s_4\}$ and $P_2 = \{s_3\}$, which violates 6 of the initial dichotomies (numbered 0, 3, 4, 5, 6, 7). Further partition of the symbols in P_1 yields $P_{11} = \{s_0, s_4\}$ and $P_{12} = \{s_1, s_2\}$, which violates four of the initial dichotomies (numbered 0, 3, 6, 7 in the example).

Merging of restricted dichotomies: Here the restricted dichotomies generated from each of the sub-partitions, say P_1 and P_2, are merged to obtain a set of dichotomies that ensures unique codes for all the symbols in the merged partition, $P = P_1 \cup P_2$. Since the sets of symbols in P_1 and P_2 are disjoint, each dichotomy in P is a union of one dichotomy each from P_1 and P_2. Thus, m dichotomies for P_1 and n dichotomies for P_2 yield mn candidate dichotomies for P.

Consider partitions $P_1 = \{s_0, s_1, s_2, s_4\}$ and $P_2 = \{s_3\}$ which are to be merged for the example of Fig. 6.3. Assume that the encoding dichotomies chosen (by recursive application of this algorithm) for P_1 are $D_1 = \{(s_0s_4; s_1s_2), (s_0s_2; s_1s_4)\}$. The only choice for P_2 is $D_2 = \{(s_3;)\}$. The merged dichotomies to be considered are $D = \{(s_0s_1s_2s_4; s_3), (s_0s_3s_4; s_1s_2), (s_0s_4; s_1s_2s_3), (s_0s_2s_3; s_1s_4), (s_0s_2; s_1s_3s_4)\}$. The best encoding of length 3 is chosen from this set by the next step.

Selection of best restricted dichotomies: The objective of this final step is to choose a

employ this technique since the number of incompatibilities is often enormous. Additionally, the prime dichotomy is required to have a bounded number of symbols in each block, which further complicates the approach.

minimal number of candidate dichotomies that violate the minimum number of encoding constraints, yet covers all the uniqueness constraints. It is important to note that when the best selection of dichotomies restricted to a subset of symbols is sought, a global view of constraints (and cost function) must be employed. For example, consider the subset of symbols $P = \{p_1, ...p_k\}$ with candidate dichotomies D_p. A cover of size c_{D_p} is desired. The constraints of the entire problem are first restricted to the symbols $p_1, ...p_k$. The cost function evaluation technique mentioned in the previous section is applied to each selection of c_{D_p} dichotomies from D_p. The set that minimizes the given cost function is chosen as the best selection of restricted dichotomies.

Continuing with the example of Fig. 6.3, following the merging step described above, the 3 best dichotomies selected are $(s_0s_1s_2s_4; s_3), (s_0s_2; s_1s_3s_4)$ and $(s_0s_4; s_1s_2s_3)$. This is done by evaluating all selections of size 3 from the set D that cover all uniqueness constraints and minimize the literal count. In the general case the number of evaluations can be restricted to some fixed number to reduce the search space.

This heuristic algorithm has shown promising results and has been successfully applied to other encoding constraint satisfaction problems [97, 7].

6.8 Other Applications

In this section we illustrate that the formulation presented in Section 6.6 provides a uniform framework for the satisfaction of various other encoding problems.

6.8.1 Input Encoding Don't Cares

The notion of an encoding don't care was first described in [91], and an example of how encoding don't cares are generated in the two-level case is given in [145]. A face constraint containing symbols a, b and e and with symbols c and d as encoding don't cares is denoted $(a, b, [c, d], e)$. This constraint specifies that a, b, e must be assigned to one *face* of a binary *n*-dimensional cube, with c and d free to be included or excluded from this face, and no other symbol sharing this face. Encoding don't cares have been shown to be essential for determining good factors in deriving a multi-level implementation of a given multi-valued description [85].

A simple example shows that suboptimal solutions of P-2 are computed when input encoding don't cares are disregarded. Given the symbols $S = \{a, b, c, d, e, f\}$ and the face constraints (a, b), (a, c), (a, d), $(a, b, [c, d], e)$, a minimum length encoding uses 3 primes, e.g. $(a, b, e; d, f)$,

$(a,c,d;b,e,f)$, $(a,b,d;c,e,f)$. If the encoding don't cares are forced to be in the face constraint, i.e. $(a,b,[c,d],e)$ is replaced by (a,b,c,d,e) then a minimum length encoding uses 4 primes, e.g. $(a,b,c,d,e;f)$, $(a,b,c;d,e,f)$, $(a,c,d;b,e,f)$, $(a,b,d;c,e,f)$. In the case that the encoding don't cares are forced not to be in the face constraint, i.e. $(a,b,[c,d],e)$ is replaced by (a,b,e) a minimum length encoding uses 4 primes, e.g. $(a,b,e;c,d,f)$, $(a,b,c;d,e,f)$, $(a,d;b,c,e,f)$, $(a,c,d;b,e,f)$.

The framework described in Section 6.6 naturally handles encoding don't cares. Consider the face constraint $(s_0s_1s_3[s_5])$, which implies that s_5 may or may not be chosen to be on the same face as s_0, s_1 and s_3. Converting this constraint to initial dichotomies is simply a matter of not generating the dichotomies $(s_0s_1s_3;s_5)$ and $(s_5;s_0s_1s_3)$. The absence of these dichotomies enables s_5 to be either inside or outside the face that includes s_0, s_1 and s_3. In the presence of encoding don't cares, a prime dichotomy may be a bi-partition of a subset of the symbols. In contrast, when encoding don't cares are not used, each prime dichotomy is a bi-partition of the entire set of symbols. For instance, if we consider the set of face constraints of the previous example (a,b), (a,c), (a,d), $(a,b,[c,d],e)$, the prime dichotomies generated by the extended definition of compatibility are: $(a,b,e;f)$, $(a,b,e;d,f)$, $(a,b,e;c,f)$, $(a,b;c,d,e,f)$, $(a,c;b,d,e,f)$, $(a,d;b,c,e,f)$, $(a,b,c;d,e,f)$, $(a,c,d;b,e,f)$, $(a,b,d;c,e,f)$, $(a,b,c,d;e,f)$. A minimum cover of 3 primes can be extracted out of them, as shown before.

The algorithms described for the feasibility check and exact encoding, shown in Figures 9.1, 9.2 and **??** respectively, extend naturally to encoding don't cares. Note that the satisfiability check algorithm described in [39] cannot be easily extended to handle encoding don't cares without a significant penalty in run-time. The encoding algorithm presented in [147] also cannot be extended to handle don't cares.

6.8.2 Distance-2 Constraints

In [135, 134, 35] a condition for easy and full sequential testability requires an encoding such that the codes assigned to a selected pair of states, say a and b, must be at least distance-2 apart. This condition may be easily satisfied by selecting at least two prime dichotomies in the minimum cover, each having a and b in different blocks. Suppose that, of all the prime dichotomies, the pairs $\{p_1,p_2\}$ and $\{p_3,p_4\}$ have a and b in different blocks. At least one of the two pairs must be chosen in a final cover. This is enforced by augmenting the binate covering formulation with the clauses

$$(p_1+\overline{b}_1)(p_2+\overline{b}_1)(p_3+\overline{b}_2)(p_4+\overline{b}_2)(b_1+b_2),$$

where b_1 and b_2 are two new columns of the covering table.

6.8.3 Asynchronous State Assignment

The state assignment algorithm proposed by Tracey [143] may also be applied in performing state assignment for asynchronous state machines [74]. The basic idea is that whenever a pair of state transitions occur under the same input (so that the input values cannot be used to distinguish among them), at least one state signal must remain constant during both transitions and have a different value for each transition. This set of constant signals allows the circuit to distinguish among different transitions thus avoiding critical races. Tracey was the first to propose the concept of dichotomy as corresponding informally to the idea of a column (bit) in the binary encoding of the internal states. It distinguishes one set of states from another by a single bit in the corresponding encodings. The implementation in [74] successfully uses our exact input encoding algorithm (*cf.* Section 6.5).

6.8.4 Logic Decomposition

In [97] it is investigated the problem of decomposing a function so that the resulting sub-functions have a small number of cubes or literals. The decomposition problem is formulated as an encoding problem. In general, an input-output encoding formulation has to be employed to solve the problem. However, it is shown that for programmable gate array architectures which use look-up tables, the input encoding formulation suffices, provided one uses minimum-length codes. The unused codes are used as don't cares for simplifying the sub-functions. An average improvement of over 20% is achieved when encoding is used while performing the decomposition. The encoding is performed using the heuristic algorithm described in Section 6.7.1.

6.8.5 Logic Partitioning

In [7] the problem of encoding the communication between two logic blocks is studied. Two separate blocks of logic can communicate unidirectionally through a channel that consists of a number of communication lines. The encoding of the symbols communicated across the channel has two requirements: first, the encoding width is fixed (usually to the minimum possible width), and second, the encoding must minimize the amount of logic in the sending and receiving blocks while balancing the size of the blocks. By definition, the input encoding constraints and output encoding constraints are each taken from different blocks of logic. Consequently, balancing the size of the

blocks translates into balancing the amount of constraint satisfaction in the two sets of constraints. Since the existing constraint satisfaction algorithms do not perform constraint satisfaction balancing, only the encoding constraints generated from the receiving block are considered. The heuristic input encoding algorithm described in Section 6.7.1 is used among others.

6.8.6 Limitations of Dichotomy-based Techniques

This section has illustrated how new classes of encoding constraints, together with face and output constraints, can be accommodated in the dichotomy-based frame. It is legitimate to ask what kind of constraints cannot be naturally solved using dichotomies. Such an example of unwieldy encoding constraints are *chain* constraints [1] used to derive area-optimal finite state machine implementations that use counter-based PLA structures. State assignment in [1] consists of a step of deriving face and chain constraints and a step of satisfying them. A chain constraint requires that increasing binary numbers be assigned to the codes of the ordered sequence of states. The first element in the chain can be given any code. For instance, a chain constraint involving the ordered sequence $a, b, c, d, e, f, g, h, i$ is denoted by $(a - b - c - d - e - f - g - h - i)$ and is satisfied by the encoding $a = 0010$, $b = 0011$, $c = 0100$, $d = 0101$, $e = 0110$, $f = 0111$, $g = 1000$, $h = 1001$, $i = 1010$. For every pair of adjacent states in the chain the code of the right state is equal to the code of the left state increased by one in binary arithmetic. As an example of encoding problem with face and chain constraints, consider the face constraints (b, c), (a, b), and the chain $(d - b - c - a)$. A satisfying assignment is: $a = 00, b = 10, c = 11, d = 01$.

Even though it is possible, for a given code length, to add to the covering expression the clauses that impose the chain conditions, a straightforward solution seems to require a computationally expensive enumeration.

6.9 Results

Table 1 gives the results of using the exact encoding algorithm on a set of examples using both input and output encoding constraints. These constraints are generated using an extension of the procedure described in [91] that also generates good disjunctive constraints. The procedure has been described in Chapter 5. The procedure for generating encoding constraints ensures that the constraints are satisfiable by calling the algorithm in Figure 9.3. The number of valid prime encoding-dichotomies is shown in the third column. As seen from the table, all the examples with

Name	**# States**	**# Primes**	**# Bits**	**Time** (secs)
bbsse	16	1449	7	20
cse	16	201	7	3
dk16	27	24316	12	1050
dk512	15	35	9	1
donfile	24	673	12	17
ex1	20	2023	9	45
keyb	19	189	9	4
kirkman	16	54	11	8
master	15	972	5	4
planet	48	> 50000	*	*
s1	20	469	7	10
s1a	20	50	7	3
sand	32	2481	11	88
scf	121	> 50000	*	*
styr	30	> 50000	*	*
tbk	32	13	12	41
viterbi	68	> 50000	*	*
vmecont	32	> 50000	*	*

* indicates results not available

Larger examples were not experimented with

Table 1: Exact input and output encoding

Name	States	# Constraints	Constraints		Cubes	
			NOVA	ENC	NOVA	ENC
bbsse	16	5	3	3	12	8
cse	16	12	8	8	24	18
dk16	27	33	25	20	43	48
dk512	15	10	8	9	12	11
donfile	24	24	8	11	48	39
ex1	20	11	8	8	19	19
kirkman	16	25	9	9	58	58
planet	48	12	12	12	12	12
s1	20	14	14	14	14	14
s1a	20	14	14	14	14	14
sand	31	7	6	6	8	8
styr	30	18	14	14	29	26
scf	121	14	11	*	21	*
tbk	32	98	44	39	284	237
viterbi	68	6	6	6	6	6
vmecont	32	40	24	25	81	67

Constraints: Number of constraints to be satisfied

Constraints: Number of satisfied constraints

Cubes: Number of cubes in a two-level implementation of the constraints

NOVA: Encoding using NOVA [147], minimum code length

ENC: Heuristic encoding, minimum code length

* : Out of memory

Table 2 : Two-level heuristic minimum code length input encoding

less than 50000 primes completed in very little CPU time on a DEC 3100 workstation. In the case of *planet* there are only nine dominance constraints and no disjunctive constraints, which lead to almost no decrease in the number of primes generated from the face constraints (exponential in the worst case). In the case of *vmecont* there are only eight different face constraints (six of them have only two states), which lead to a huge number of primes being generated from the large number of un-implied uniqueness constraints. Thousands of satisfiability checks on input and output encoding constraints can be performed routinely in a matter of seconds, showing the efficiency of our algorithm. The previous approach suggested for prime generation in [154] does not complete on any of the examples.

Table 2 compares an implementation of the heuristic algorithm described in Section 6.7.1 with the best bounded-length input encoding algorithm implemented in NOVA [147] (option *-e ih*). NOVA is a state assignment program for two-level implementations, that features a variety of constraint satisfaction algorithms. The input constraints are generated by calling the two-level multiple-valued logic minimizer ESPRESSO [114]. The number of satisfied face constraints and the number of cubes in a two-level implementation of the constraints using the minimum possible length for encoding are compared in the table. While both algorithms perform comparably with regard to the number of constraints satisfied, our approach has a significant advantage with respect to the number of cubes needed to implement the input constraints in two-level form. This cost function is very important because it measures the advantage of satisfying a subset of input constraints in a fixed code-length more precisely. Our algorithm in almost all cases needs fewer cubes than the algorithm in NOVA. On the benchmark set it requires on average 13% fewer cubes and in some cases the gain is more than 20%. The number of cubes listed in Table 2 under the column NOVA, is not the same as the number of cubes of the final FSM implementation obtained by NOVA [147]. NOVA performs additional encoding tasks to approximate the input-output encoding problem that arises in FSM's. Instead, *we compare only the quality of the input encoding algorithms*. For instance, we report 284 cubes for *tbk* using NOVA and 237 cubes for our algorithm. This means that if *tbk* were to be encoded with 5 bits using only input constraint information, the encoding algorithm in NOVA would require 284 and our algorithm 237 cubes to implement the input constraints. In reality with the option *-e ih* NOVA achieves 147 cubes, because it does not limit itself to input constraints satisfaction (and with the option *-e ioh* it achieves 57 cubes, using a better model of the input-output encoding problem). A heuristic algorithm that considers partial satisfaction of a set of input and output constraints remains to be developed. In the example of *sand* only 8 cubes are reported for both algorithms, because these are the cubes needed to implement the cubes generating input constraints. However, there are many more cubes in the FSM that do not generate input constraints, and are not reflected in the table.

Table 3 compares our approach to simulated annealing for multi-level examples. Input constraints with don't cares are generated by the multiple valued multi-level synthesis program MIS-MV [85] with the number of factored form literals in the encoded implementation as cost function (in practice, the number of literals in a sum-of-product representation of the encoded constraints is used as an approximation to this cost function). Because of the presence of encoding don't cares and the cost function of literals, simulated annealing was the only other known algorithm for solving this problem. We use two sets of experiments to compare the effectiveness of our heuristic

bounded-length algorithm versus the version of simulated annealing algorithm implemented in MIS-MV. Minimum-length encoding is always used. MIS-MV is run using a script that invokes the constraints satisfaction routine six times; five times to perform a cost evaluation that drives the multi-valued multi-level optimization steps and one final time to produce the actual codes that replace the symbolic inputs [85]. Simulated annealing is called the first five times with 1 pairwise code swap per temperature point, while the last call performed 10 pairwise code swaps per temperature point. Simulated annealing does not complete on the larger examples with 10 pairwise swaps per step. These examples are marked with a † in the table, and only 4 swaps were allowed per temperature step for these examples. When using our heuristic algorithm, the full-fledged encoder is called all six times. See [85] for a detailed explanation of the scripts.

As can be seen from Table 3, our algorithm on average performs a little better than simulated annealing in terms of literal count. This is significant especially in the large examples, where it reduces the literals counts up to 10% further than simulated annealing. When our algorithm does worse, it is within 5% of the simulated annealing result. However, a significant parameter here is the amount of time taken. Simulated annealing consumes at least an order of magnitude of time (two orders or more for larger sized examples) more than our algorithm when a better quality solution is desired, i.e. using 10 swaps per step. On attempting to reduce the runtime to be comparable to our approach, a noticeable loss of optimization quality compared to our approach may be observed in the table. Further improvements to the heuristic encoding algorithm are possible.

6.10 Conclusions

This chapter has presented a comprehensive solution to the problem of satisfying encoding constraints. We have shown that the problem of determining a minimum length encoding to satisfy both input and output constraints is NP-complete. Based on an earlier method for satisfying input constraints [154], we have provided an efficient formulation of an algorithm that determines the minimum length encoding that satisfies both input and output constraints. It is shown how this algorithm can be used to determine the feasibility of a set of input and output constraints in polynomial time in the size of the input. While all previous exact formulations have failed to provide efficient algorithms, an algorithm that efficiently solves the input and output encoding constraints exactly has been described. A heuristic procedure for solving input encoding constraints with bounded code-length in both two-level and multi-level implementations is also demonstrated. In the multi-level case, only a very time-consuming algorithm based on simulated annealing was

Name	**States**	**Literals**		**Time**	
		SA	ENC	SA	ENC
bbsse	16	162	164	3017	175
cse	16	229	236	3969	234
dk16	27	336	380	27823	1523
dk512	15	82	85	2090	138
donfile	24	154	172	16265	935
kirkman	16	201	229	2621	322
master	15	392	398	2069	423
s1	20	280	304	16297	833
s1a	20	240	254	4878	241
†sand	31	763	737	1926	2332
†scf	121	*	*	*	*
†styr	30	581	608	3128	1359
†planet	48	648	639	10298	14983
†tbk	32	560	498	3774	4090
†viterbi	68	327	322	860	1013
†vmecont	32	378	364	2074	2883

SA: Simulated annealing (5 calls with 1 move per step and 1 call with 10 moves per step)

ENC: Heuristic encoding in minimum code length (6 calls)

Time SA : Time for SA; includes run time for minimization script [85]

Time ENC : Time for ENC; includes run time for minimization script [85]

†: SA does not complete in 10 hours with 10 moves per step; SA limited to 4 steps per move

*: Does not complete in 10 hours

Table 3 : Multi-level heuristic minimum code length input encoding

known before. This framework has also been used for solving a variety of encoding constraint satisfaction problems generated by other applications.

Chapter 7

Generalized Prime Implicants

7.1 Introduction

A method for exploring globally the solution space of optimal two-level encodings was proposed by Devadas and Newton in [39]. Their key contribution was the definition of Generalized Prime Implicants (GPI's), as a counterpart of prime implicants in two-level minimization.

Unfortunately, the number of GPI's is so large even for small FSM's, that in practice it is out of question to compute them and a fortiori to solve the induced covering problem for non-trivial examples.

Recently, enumeration and manipulation of very large sets have been successfully performed by representing their characteristic functions with Binary Decision Diagrams (BDD's). In many cases of practical interest these sets have a regular structure that translates into small-sized BDD's, even when an explicit representation would be impossible to compute. Here, loosely, we consider a representation as explicit if it requires space lineraly proportional to the size of the represented set.

In particular, researchers at Bull and UCB [25, 79, 53] investigated implicit computations of prime implicants of a two-valued or multi-valued function. In some examples all primes could be computed implicitly, even when explicit techniques implemented in ESPRESSO [11] failed to do so. Moreover, implicit algorithms have been designed to reduce the unate table of the Quine-McCluskey procedure to its cyclic core [29, 53], and to solve the binate covering problem associated with exact state minimization [66].

In the present work we capitalize on these algorithmic technologies to propose a complete procedure to generate and select GPI's based on implicit computations. This approach combines

techniques for implicit enumeration of primes and implicit solution of covering tables together with a new formulation of the problem of selecting an encodeable cover of GPI's. The proposed algorithms have been implemented using state assignment of FSM's as a test case. The experiments exhibit a set of medium FSM's where large GPI problems could be solved for the first time, showing that these techniques open a new direction in the minimization of symbolic logic. Since the problem of symbolic minimization is harder than two-valued logic minimization, more practical work is required to improve the efficiency of the implementation and to tie it with good heuristics to explore the solution space of encoding problems. The present contribution shows how to extract a minimal encodeable cover from a large set of GPI's, allowing - in line of principle - the exploration of all minimal encodeable covers. This advances the state-of-art of symbolic minimization, which up to now has been done with various heuristic tools [92, 147, 42, 77], often very well-tuned for their domain of application, but lacking a rigorous connection beween an exact theory and the approximations made. For instance it is noticeable that these tools, with the exception of ESP_SA, cannot predict the cardinality of the covers that they produce, while the size of a minimized encoded cover of GPI's matches the size of the cover obtained after encoding (with the same codes) and minimizing the original cover.

The presentation is organized on a number of chapters as follows. In Section 7.2 we introduce some basic definitions. In Section 7.3 we introduce GPI's. In Section 7.4 we show how generation of GPI's of a symbolic cover can be reduced to finding the prime implicants of a companion multi-valued function. The relations of GPI's to primes of encoded covers is analyzed in Section 7.5. The problem of selecting a minimum set of encodeable GPI's by reduction to unate covering is described in Section 8.1, and by reduction to binate covering is described in Section 8.2. The issue of non-determinism and GPI's is discussed in Section 8.3. A theory of encodeability of GPI's based on the new notions of raising graphs and updating sets is presented in Section 9.1. The passage to implicit algorithms is done in Sections 11.1 and 11.2. In Section 11.3 we present an implicit solution of the GPI selection problem, while Section 11.4 demonstrates on an example the implicit algorithm. The correctness of the results is verified with the method shown in Section 11.5. Implementation issues are discussed in Section 11.6. In Section 11.7 experimental results are given, while conclusions are drawn in Section 11.8.

7.2 Basic Definitions

7.2.1 Finite State Machines

A Finite-State Machine (FSM) is represented by its **State Transition Graph** (STG) or equivalently, by its **State Transition Table** (STT). A STG is denoted by a sextuple $\{I, O, S, IS, \delta, \lambda\}$, where I and O are the sets of inputs and outputs, S is the set of states and IS is the set of initial states. δ (next state function) is a mapping from $I \times S$ to S that given an input and a present state defines a next state. λ (output function) is a mapping from $I \times S$ to O that given an input and a present state defines an output. An STG where one next-state and one output for every possible transition from every state are defined corresponds to a **completely specified finite state machine** (CSFSM). An STT is a tabular representation of the FSM. Each row of the table corresponds to a single edge in the STG. Conventionally, the leftmost columns in the table correspond to the primary inputs and the rightmost columns to the primary outputs. The column following the primary inputs is the present-state column and the column following that is the next-state column.

An **incompletely specified finite state machine** (ISFSM) is one where either δ or λ or both are a relation of a restricted kind, i.e. there is at least one pair (i, s) on which either $\delta(i, s)$ or $\lambda(i, s)$ (or both) is equal to the set of all possible values, written usually in cube notation. For instance, suppose that $O = B^3$, then $i_1\ s_1\ \ s_2\ \ ---$ denotes a transition under input i_1 from s_1 to s_2 which outputs any of the possible 8 minterms in B^3; $i_1\ s_1\ \ ANY\ 01-$ denotes a transition under input i_1 from s_1 to any state in S which outputs either 010 or 011 (instead of ANY one can write $*$ or $-$). Lastly, $i_1\ s_1\ \ ANY\ \ ---$ denotes a transition under input i_1 from s_1 to any state in S which outputs any minterm in B^3; for economy of representation, one usually omits these transitions (sometimes called *missing* or *unspecified* transitions) from an FSM description. When doing state assignment, if there are more hardware states than symbolic states[1], ANY of a missing transition can be implemented by any possible hardware state. To every STG containing unspecified next-states one can construct an equivalent STG where all unspecified next states are replaced by a trap state T, as in [98]. The transitions from T under any input go to T itself and their outputs are unspecified. The new STG describes exactly the same behaviours as the old one.

[1] Suppose that 3 symbolic states are encoded with 2 bits, then there are 4 hardware states.

7.2.2 Multi-valued Functions

We review the definitions used for **multi-valued** (also known as **symbolic**) input binary-valued functions. For a more complete treatment the reader is referred to [114].

Definition 7.2.1 *Let $p_i, i = 1, \ldots, n$ be positive integers. Define $P_i = \{0, \ldots, p_i - 1\}$ for $i = 1, \ldots, n$, and $B = \{0, 1, *\}$. A multiple-valued input, binary-valued output function, f, is a mapping*

$$f : P_1 \times P_2 \times \cdots \times P_n \rightarrow B$$

The function f has n multiple-valued inputs. Each input variable i assumes one of the p_i values in P_i. The value $* \in B$ is used when the function value is unspecified (i.e., it is allowed to be either 0 or 1).

An n-input, m-output switching function can be represented by a multiple-valued function of $n + 1$ variables where $p_i = 2$ for $i = 1, \ldots, n$, and $p_{n+1} = m$. The minimization problem for multiple-output functions is equivalent to the minimization of a multiple-valued function of this form [119].

Definition 7.2.2 *Let X_i be a variable taking a value from the set P_i, and let S_i be a subset of P_i. $X_i^{S_i}$ represents the Boolean function*

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

$X_i^{S_i}$ is called a **literal** of variable X_i. If $S_i \equiv \emptyset$, then the value of the literal is always 0, and the literal is called empty. If $S_i \equiv P_i$, then the value of the literal is always 1, and the literal is called full.

Two-valued (or **binary**) functions are a special case of multi-valued functions where $P_i = \{0, 1\}$ for $i = 1, \ldots, n$. In the case of a two-valued single-output function, some notational simplification is then possible. A cube may be written as a vector on a set of variables with each position representing a distinct variable. The values taken by each position are 1, 0 or 2 (same as $-$, don't-care), signifying the true form, negated form or both of the variable corresponding to that position. A **minterm** is a cube with only 0 and 1 entries. Cubes can be classified based on the number of 2 entries. A cube with k entries or bits which take the value 2 is called a k**-cube**. A minterm thus is a 0-**cube**.

A **product term** (or **cube**) is a Boolean product (AND) of literals. A minterm or 0-cube is a product-term in which the sets of values of all literals are singletons. If a product term evaluates to 1 for a given minterm, the product term is said to **contain** (or **cover**) the minterm.

A **sum-of-products** (or **cover**) is a Boolean sum (OR) of product terms. If any product term in the sum-of-products evaluates to 1 for a given minterm, then the sum-of-products is said to contain the minterm. If a literal in a product-term is empty, the product term contains no minterms, and is called the null product (written $\emptyset$). The **on-set** of a function is the set of minterms for which the function value is 1. Likewise, the **off-set** is the set of minterms for which the function value is 0, and the **DC-set** is the set of minterms for which the function value is unspecified.

In the definitions which follow, $S = X_1^{S_1} X_2^{S_2} \cdots X_n^{S_n}$ and $T = X_1^{T_1} X_2^{T_2} \cdots X_n^{T_n}$ represent product terms.

The product term S contains the product term T ($T \subset S$) if $T_i \subset S_i$ for $i = 1 \ldots n$. The **complement** of the literal $X_i^{S_i}$ (written $\overline{X_i^{S_i}}$) is the literal $X_i^{P_i - S_i}$. The complement of the product term S ($\overline{S}$) is the sum-of-products $\bigcup_{i=1}^{n} \overline{X_i^{S_i}}$.

The **intersection** of product terms S and T ($S \cap T$) is the product term

$$X_1^{S_1 \cap T_1} X_2^{S_2 \cap T_2} \cdots X_n^{S_n \cap T_n}.$$

If $S_i \cap T_i = \emptyset$ for some i, then $S \cap T = \emptyset$ and S and T are said to be disjoint. The intersection of covers F and G is the union of $f \cap g$ for all $f \in F$ and $g \in G$. The **distance** between S and T ($distance(S,T)$) is $|\{i | S_i \cap T_i = \emptyset\}|$.

The **consensus** of S and T ($consensus(S,T)$) is the sum-of-products

$$\bigcup_{i=1}^{n} X_1^{S_1 \cap T_1} \cdots X_i^{S_i \cup T_i} \cdots X_n^{S_n \cap T_n}.$$

If $distance(S,T) \geq 2$ then $consensus(S,T) = \emptyset$. If $distance(S,T) = 1$ and $S_i \cap T_i = \emptyset$, then $consensus(S,T)$ is the single product term $X_1^{S_1 \cap T_1} \cdots X_i^{S_i \cup T_i} \cdots X_n^{S_n \cap T_n}$. If $distance(S,T) = 0$ then $consensus(S,T)$ is a cover of n terms. If the consensus of S and T is nonempty, it is the set of maximal product terms (ordered by containment) which are contained in $S \cup T$ and which contain minterms of both S and T. The consensus of two covers F and G is the union of $consensus(f,g)$ for all $f \in F$ and $g \in G$.

The **cofactor** (or **cube restriction**) of S with respect to T (S_T) is empty if S and T are disjoint. Otherwise, the cofactor is the product term

$$X_1^{S_1 \cup \overline{T_1}} \cdots X_2^{S_2 \cup \overline{T_2}} \cdots X_n^{S_n \cup \overline{T_n}}.$$

The cofactor of a cover F with respect to a product term S is the union of f_S for all $f \in F$.

An **implicant** of a function is a product term which does not contain any minterm in the off-set of the function. A **prime implicant** of a function is an implicant which is not contained by any other implicant of the function. An **essential prime implicant** is a prime implicant which contains a minterm which is not covered by any other prime implicant.

The product term S can be represented in **positional cube notation** as a binary vector in the following form:

$$c_1^0 c_1^1 ... c_1^{p_1-1} - c_2^0 c_2^1 ... c_2^{p_2-1} - c_n^0 c_n^1 ... c_n^{p_n-1}$$

where $c_i^j = 0$ if $j \notin S_i$, and $c_j^i = 1$ if $j \in S_i$. In other words, a symbolic variable that can take values from a set of cardinality n is represented in positional cube notation by an n-bit vector to denote a literal of that variable such that each position in the vector corresponds to a specific element of the set. A 1 in a position in the vector signifies the presence of an element in the literal while a 0 signifies the absence. This method of representation is commonly known as **one-hot**. By complementing the n-bit vector that represents the one-hot encoding of a symbolic variable, one gets a representation called **complemented one-hot**.

7.3 Generalized Prime Implicants

7.3.1 Definition of Generalized Prime Implicants

Multi-valued inputs and binary-valued outputs functions can be represented by multiple-valued functions where the set of binary outputs is treated as another multi-valued input variable. Positional cube notation allows also to represent any function with multi-valued input and multi-valued output variables. This is commonly done in programs like ESPRESSO-MV, when a function with symbolic inputs and outpus (e.g., an FSM) is 1-hot encoded and then minimized. But the minimization problem for functions with multi-valued input and multi-valued output variables is not known to be equivalent to the minimization of a multiple-valued function of this form. After 1-hot encoding the onsets of the minterms (values) of a symbolic output are minimized separately. To handle the minimization problem of functions with multi-valued input and multi-valued output variables the concept of generalized prime implicants has been introduced [39].

Consider a discrete (*alias* symbolic) function whose domain and range are finite sets. The previous theory of multi-valued minimization does not take into account the effect of encoding the symbolic output variables to get a minimum two-level encoded function. More precisely it does

not model the fact that after encoding the onsets of the symbolic outputs are not anymore disjoint. To overcome this limitation a concept of generalized prime implicants has been introduced in [39]. Even though the concept can be defined for functions with many symbolic inputs and many symbolic outputs, for simplicity we will restrict most of the discussion to the case of a function with binary inputs, one symbolic input variable, one symbolic output variable and binary outputs. This handles symbolic descriptions of FSM's. In what follows we will often not make a distinction between a function f and a cover that represents f.

Consider an FSM M given by a symbolic cover $f : I \times \Sigma \rightarrow \Sigma \times O$. Given an integer n and an encoding function $e : \Sigma \rightarrow B^n$, let $e(f)$ be the encoded cover of f, i.e. the cover obtained from f after replacement of the states with their codes, according to e. Consider a prime implicant $s = i\ p\ n\ o$ of the function represented by the encoded cover $e(f)$. Associate to the encoded present state field p the set of states $S_p \subseteq \Sigma$, whose codes are contained in p. Associate to the encoded next state field n the set of states $S_n \subseteq \Sigma$, whose intersection of the codes is n [2]. Both operations are well-defined. Then one can associate to s the following symbolic product-term $S = iX_p^{S_p} X_n^{S_n} o$.

Given $f : I \times \Sigma \rightarrow \Sigma \times O$, consider a symbolic product-term $S = iX_p^{S_p} X_n^{S_n} o$. S is a multi-valued input binary-valued output product-term, except that it has a multi-valued output variable X_n whose multi-valued literal does not need to be a singleton. This latter feature makes it "generalized". A question arises: what is the meaning of a such a generalized product-term ? Such a generalized product-term is a template for corresponding encoded product-terms, as the following definitions clarify.

Definition 7.3.1 *Given a set of symbols $S \subseteq \Sigma$ and an encoding function $e : S \rightarrow B^n$, let $e(s) = e(s)_1 e(s)_2 \cdots e(s)_n$ for $s \in S$. Then $\biguplus e(S)$ is the product-term $X_1^{S_1} X_2^{S_2} \cdots X_n^{S_n}$ where $S_i = \{0,1\}$ iff $\exists s, \tilde{s} \in S$ s.t. $e(s)_i = 1$ and $e(\tilde{s})_i = 0$; $S_i = \{0\}$ iff $\forall s \in S\ \ e(s)_i = 0$; $S_i = \{1\}$ iff $\forall s \in S\ \ e(s)_i = 1$.*

$\biguplus$ defines the minimum Boolean subspace of B^n spanned by the codes of the states of S.

Definition 7.3.2 *Given a set of symbols $S \subseteq \Sigma$ and an encoding function $e : S \rightarrow B^n$, let $e(s) = e(s)_1 e(s)_2 \cdots e(s)_n$ for $s \in S$. Then $\bigcap e(S)$ is the product-term $X_1^{S_1} X_2^{S_2} \cdots X_n^{S_n}$ where $S_i = \{0\}$ iff $\exists s \in S$ s.t. $e(s)_i = 0$; $S_i = \{1\}$ iff $\forall s \in S\ \ e(s)_i = 1$.*

[2] Consider the next states in f of the transitions with minterms in $i\ S_p$, the intersection of their codes must be equal to n if s is a prime implicant (an exception is the case of transitions with ANY next state and specified proper outputs), and must be $\leq n$ if s is an implicant that is not a prime.

$\bigcap$ defines the vertex of B^n obtained by bit-wise intersection of the codes of the states of S.

Definition 7.3.3 *Given a product-term* $S = iX_p^{S_p}X_n^{S_n}o$ *of a symbolic function* $f : I \times \Sigma \rightarrow \Sigma \times O$*, and an encoding function* $e : S \rightarrow B^n$*, the encoded product-term* $e(S)$ *is given by:* $e(S) = i \biguplus e(S_p) \bigcap e(S_n) o$.

Example 7.3.1 *Consider the symbolic product-term* $1-0\ \ st1, st3, st4\ \ st2, st3\ \ 1001$ *and the encoding* $e(st0) = 011$, $e(st1) = 000$, $e(st2) = 111$, $e(st3) = 100$, $e(st4) = 010$, $e(st5) = 101$. $\biguplus e(S_p = \{st1, st3, st4\})$ *is* $--0$, $\bigcap e(S_n = \{st2, st3\})$ *is* 100, $e(1-0\ st1, st3, st4\ st2, st3\ 1001)$ *is* $1-0\ \ --0\ \ 100\ \ 1001$.

Definition 7.3.4 *A generalized implicant (GI)* S *of a symbolic function* $f : I \times \Sigma \rightarrow \Sigma \times O$ *is a product-term of the form* $S = iX_p^{S_p}X_n^{S_n}o$ *such that there are an integer* n *and an encoding function* $e : \Sigma \rightarrow B^n$ *so that* $e(S)$ *is an implicant of* $e(f)$.

Definition 7.3.5 *A generalized prime implicant (GPI)* S *of a symbolic function* $f : I \times \Sigma \rightarrow \Sigma \times O$ *is a generalized implicant such that there are an integer* n *and an encoding function* $e : \Sigma \rightarrow B^n$ *so that* $e(S)$ *is a prime implicant of* $e(f)$.

It is true that for each prime implicant of an encoded FSM there is a GPI.

Theorem 7.3.1 *For each prime implicant of the Boolean function represented by an encoded cover there is at least one GPI.*

Proof: Given a prime of a Boolean function represented by an encoded cover, consider the present state subcube and find all states whose codes are contained in it, discarding those that do not correspond to a state in the symbolic cover. This gives S_p. Find in the original symbolic cover the next states of the states in S_p under the proper inputs of the prime. This gives S_n (the intersection of the codes of the states in S_n dominates the next state subcube of the prime). The proper input and output subcubes of the GPI are the same as those of the prime. ■

A similar theorem holds replacing prime implicant with implicant. The given definition does not tell us how to compute the GPI's. GPI's can be obtained by a symbolic equivalent of the consensus operation. Actually this is how they were first introduced in [39], as we will see in the next section.

Definition 7.3.6 *A GI* g_1 *covers another GI* g_2 *iff the proper input and output of* g_1 *contain, respectively, the proper input and output of* g_2*, the present state literal of* g_1 *is a superset of the present state literal of* g_2 *and the next state literal of* g_1 *is a subset of the next state literal of* g_2.

7.3.2 Generalized Prime Implicants by Consensus Operation

In old textbooks [94] it was common to represent a multiple-output function by a cover of the function consisting of a set of cubes in the common input space, with an output tag attached to each cube to specify the functions to whose onset the cube belongs. We call it functional view. Instead in the more modern relational view the outputs are treated as one more multi-valued variable [118, 114]. For instance a minterm in the relational view is a product-term in the input and output variables where each literal is a singleton; in the functional view it is a product-term in the input variables where each literal is a singleton, with an attached tag that specifies one or more output functions. Therefore a minterm in the functional view may correspond to more than one minterm in the relational view.

Generalized Implicants (GI's) extend the definition of multiple-output implicants to the case that some output variables are symbolic. In analogy to an output tag, the notion of symbolic tag has been introduced in [39]. A GI can be written as a cube with associated tags for the multiple-valued and binary-valued output functions. The tag of a cube for a multiple-valued output variable gives the output symbol to whose onset the cube belongs. We let the tag of a symbolic output variable contain more that one symbol, under the convention that - after encoding - the symbolic tag will be replaced by a cube that is the bit-wise intersection of the codes of the symbols in the tag.

Prime implicants are maximal implicants of a Boolean function. Implicants of multiple-output functions (multiple-output implicants) can cover 0-cubes in more than one output function. A multi-output prime implicant is a maximal implicant for a set of output functions. Prime implicants can be computed by the consensus method [107, 94]. Maximality of a multiple-output prime means that its input part cannot be expanded without intersecting the offset of at least one function in the output tag, nor any new function can be added to the output tag without the input part intersecting the offset of this added function. The consensus operation of two product-terms p_1 and p_2 is the largest product-term p such that p does not imply (i.e. is not contained in) either p_1 or p_2, but p implies $p_1 + p_2$. Iterative consensus consists of successive addition of derived consensus terms to a sum-of-product espression and removal of terms which are included in other terms. The iteration of this procedure yields the set of all prime implicants.

Boolean Consensus. Generation of all prime implicants in the Boolean domain by iterated consensus is the merging of k-cubes to form $(k+1)$-cubes until no new cubes are generated. $(k+1)$-cubes remove from the list of candidate primes those k-cubes that are covered by a $(k+1)$-cube. When two k-cubes are merged, the output part of the $(k+1)$-cube cannot dominate the output

parts of the k-cubes from which it was derived, since it is the conjunction of the output parts of the k-cubes. A $(k+1)$-cube removes a k-cube only if the input part of the $(k+1)$-cube covers the input part of the k-cube and the output part of the k-cube is the same as the output part of the $(k+1)$-cube.

Consensus can be extended to GI's by defining the symbolic tag of a consensus cube as the union of the symbolic tags of the merged cubes. From now we will indicate by $CONS$ the consensus operator. By the context it will be clear if it is Boolean consensus or symbolic consensus.

Example 7.3.2

$$CONS(11\ st1\ \ st0\ 01; 11\ st2\ \ st2\ 11) = 11\ st1, st2\ \ st0, st2\ 01$$

Since these two minterms (or, 0*-cubes) are distance-1 from each other in the input part, they can be merged together to form a* 1*-cube, with the binary output part of the* 1*-cube being the bitwise conjunction of the binary output parts of the individual* 0*-cubes. The symbolic output parts are merged too, and the output part of the* 1*-cube is the union of the output parts of the* 0*-cubes. If* s_0 *gets the code* 101 *and* s_1 *gets the code* 011*, then the output part of the encoded* 1*-cube is* 001*, saying that the cube* 11 $st1, st2$ *belongs to the onset of the third (and fifth) output function.*

GPI's are maximal implicants obtained after repeated applications of symbolic consensus. Consider the rule to generate GPI's of FSM's. We suppose that the proper inputs and outputs are binary, even though it would be easy to handle multiple-valued proper inputs and outputs.

Symbolic Consensus. $(k+1)$-cubes are generated by merging k-cubes until no new primes can be generated. A $(k+1)$-cube formed from two k-cubes has a next state tag that is the union of the two k-cubes' next state tags and an output tag that is the intersection of the outputs in the k-cubes' output tags. The binary inputs of the $k+1$-cube are obtained with the usual consensus rule for binary cubes. The present-state part of the $k+1$-cube is the union of the present state parts of the k-cubes. A $(k+1)$-cube cancels a k-cube only if their multiple-valued present state parts are identical or if the multiple-valued present state part of the $(k+1)$-cube contains all the symbolic states. [3] The binary input part of the $k+1$-cube must cover the binary input part of the k-cube. In addition, the next state and output tags have to be identical. A cube with a next state tag containing all the symbolic states and with a null output tag can be discarded.

[3] The rule has no Boolean domain counterpart and it is due to the fact that when replacing symbols with boolean vectors, the present state part yields an input constraint whose satisfiability depends on the encoding. Each of these GPI's is a multiple-output prime in the Boolean domain associated to an encoding where the input constraint is satisfied. Keeping all GPI's that differ only in the present state part, all multiple-output primes for all possible encodings are generated.

Proposition 7.3.1 *A GPI corresponding to a prime implicant of an encoded cover can always be obtained by symbolic consensus.*

Proof: Given a prime implicant of an encoded cover, consider the corresponding GPI (found as in the proof of theorem 7.3.1) and the minterms of the original symbolic cover that are contained in it. By performing symbolic consensus on the cover of the contained minterms one obtains exactly the corresponding GPI. ■

Viceversa, consider a product term obtained by symbolic consensus, then there is always an encoding such that the encoded GPI is a prime implicant of the encoded symbolic cover. For instance consider 1-hot encoding padded by a final 1, i.e., a 1 is added at the end of all codes.

7.3.3 Encodeability of Generalized Prime Implicants

Given a set of GPI's the goal is to realize the original symbolic cover. There are two issues here:

1. There may not exist a single encoding function that works for all GPI's of the cover and translates them into primes of the encoded initial cover.

2. The encoded cover of GPI's may not realize (yet) the encoded initial cover.

The first issue is one of encodeability, i.e., of finding codes that map a symbolic cover into a corresponding two-valued cover. The second issue is one of covering, i.e., of realizing all the behavior of the initial symbolic cover. We will now define carefully the conditions to satisfy both types of requirements. They will be phrased in terms of encoding constraints, expressing both encodeability and covering.

Suppose that a set of GPI's, P, is given. Consider a minterm m (in the primary input and present state space), of the original symbolic cover (a 0-cube is determined by a minterm in the proper input space and a present state) and say that it asserts the next state s_m. In an encoded cover m will assert the code assigned to s_m, denoted by $e(s_m)$. Suppose that GPI's $p_{m_1}, \ldots, p_{m_M}$ of those in P cover m. Minterm m asserts in P the intersection of next states in the tags of $p_{m_1}, \ldots, p_{m_M}$. In order that the cover of GPI's P be equivalent to the original FSM, each minterm must assert in P the same output as in the original FSM. Therefore the following **next-state encoding constraint** (or **minterm encoding constraint** or **consistency equation**) must be satisfied for every minterm m:

$$e(s_m) \quad = \quad \bigcup_{i=m_1}^{m_M} \quad \bigcap_{j} e(s_{i,j}) \tag{7.1}$$

where $s_{i,j}$s are the next states in the tag of the GPI p_{mi} and $e(s)$ is the code assigned to state s. $\bigcap$ corresponds to bitwise conjunction and $\bigcup$ corresponds to bitwise disjunction. The state s_m is called the **parent** and the states $s_{i,j}$ are called the **children** of the next-state constraint.

If no GPI in P covers m, then the constraint for m reduces to:

$$e(s_m) \quad = \quad \emptyset \tag{7.2}$$

Clearly this constraint (**empty next-state constraint**) is unsatisfiable, if some GPI that covers m is not added to P.

For example, if two GPI's, one with next state tag (s_1, s_2, s_3) and another with next state tag (s_1, s_4) are the ones in P covering minterm 10 s_1 s_1 11, the constraint for the minterm 10 s_1 s_1 11 would be $e(s_1) = e(s_1) \cap e(s_2) \cap e(s_3) \bigcup e(s_1) \cap e(s_4)$.

Moreover, in order that the cover of GPI's P be equivalent to the original FSM, each minterm must assert in P the same proper outputs as in the original FSM. Say that each GPI p_i has a corresponding output tag o_i and that the output tag of minterm m is o_m. Suppose as before that GPI's $p_{m_1}, \ldots, p_{m_M}$ of those in P cover m. The following **proper output covering constraint** must be satisfied for every minterm m:

$$o_m \quad = \quad \bigcup_{i=m_1}^{m_M} o_i \tag{7.3}$$

If minterms are defined as a product of multi-valued singleton literals as in Section 8.1, proper output covering constraints are satisfied iff a set of GPI's that covers every row is selected, i.e. by reduction to an ordinary unate covering problem.

Each GPI yields also an **input encoding constraint** (or **face embedding constraint**), i.e., the set of states in the multi-valued literal of the present state variable. An input encoding constraint is satisfiable if there is an encoding such that the codes of the states form a face (in the Boolean encoding space) that does not include the codes of the states absent from it. An input encoding constraint is satisfied in a given encoding if the codes of the states in it form a face (in the Boolean encoding space) that does not include the codes of the states absent from it. If it contains all states or only one state, the input constraint is trivial, since it does not impose any limitation on the encoding of the states.

Finally **uniqueness encoding constraints** impose that different codes states are assigned to different states (e.g., $e(s_i) \neq e(s_j)$, for $i \neq j$). Unless otherwise stated, we suppose that they must always be satisfied. They can be modelled in the same way as input constraints, and when not necessary we will not distinguish between the two types of constraints.

Sometimes constraints of various types are called collectively **encoding constraints**. It will be clear from the context which types of constraints are meant.

A set of of GPI's or of encoding constraints induced by them is said to be **encodeable** or **feasible** or **satisfiable** if there is an assignment of states to codes (Boolean vectors) such that each constraint is satisfied, according to the definition of satisfaction of its specific type of constraint. Such an assignment is called an **encoding**.

The selection of a minimum set of GPI's that satisfies both the (next state and input) encoding constraints and the (proper output) covering constraints can be modelled as a table covering problem (either a constrained unate covering or binate covering problem). This reduction will be fully developed in Section 8.1.

The tag of a GPI may contain from one to all the states. If one generates only GPI's whose tag has a cardinality less than a given bound, one has an approximate algorithm for the state assignment problem. By setting the bound to 1, a disjoint minimization problem is defined, equivalent to approximating state assignment as an input encoding problem as in [92]. By setting the bound to less than the number of states, one can trade-off quality of the solution vs. running time.

7.3.4 Sufficiency of GPI's

The problem of obtaining the minimum two-level representation of a function can be reduced to one of finding the minimum number of prime-implicants covering all the minterms. The same holds true for symbolic functions by means of GPI's, with the caveat that the chosen GPI's must be encodeable. Thus, if one selects a minimum set of encodeable GPI's that cover all the minterms, this is a minimum solution of the state assignment problem for two-level implementations. It is a solution because of encodeability, i.e., enforcing the consistency equations makes sure that each minterm asserts the same output both in the original and in the GPI cover (and so in the encoded cover). It is also a minimum solution, as the following theorem shows.

Theorem 7.3.2 *A minimum cardinality symbolic cover of an FSM can be made up exclusively of GPI's.*

Proof: We suppose that no cube of the cover has a next state tag containing all the symbolic states and a null output tag, otherwise it can be dropped and the cover would not be minimal. Assume that we have a minimum cardinality solution with a cube c_1 that is not a GPI. Let the tag of c_1 be the T. We know that a GPI p_1 exists such that

1. the binary input part of p_1 covers the binary input part of c_1;

2. p_1 and c_1 have same present state part;

3. p_1 and c_1 have same next state and binary outputs tags.

Replacing c_1 with p_1 will not change the cardinality of the cover. The only question is whether the set of GPI's so obtained is encodeable. We show now that it is the case. The generalized implicants (GI's) of the given cover are encodeable by hypothesis. The constraints of the GPI's of the new cover are the same as those of the given GI's, except for the minterms in $p_1 - c_1$. For each minterm in $p_1 - c_1$ we add new disjuncts to its consistency equation. Each disjunct is a conjunction of symbols each of which is a next state originally asserted by the minterm, because when generating GPI's (e.g. p_1) we take the union of the next states tags of the merged GI's. One of the merged GI's must cover the minterm and a GI covers a minterm only if it includes in its next state tag the next state that the minterm asserts. Since each added disjunct contains the next state asserted by the minterm, whatever encoding satisfies the old consistency equation, it satisfies also the new consistency equation. Notice that the input constraints of the GPI's of the new cover coincide with those of the given GI's, because the GPI cancellation rule requires the same present state part to delete a GI [4]. Therefore any encoding that satisfies the given GI's satisfies also the GPI's of the new cover and therefore encodeability is preserved. ■

7.4 Reduction of GPI's Computation to MV Primes Computation

The next question is how to compute efficiently GPI's. In [39] it is shown how to reduce the computation of GPI's to the computation of the primes of a multiple-valued function obtained by transformation of the given FSM. We will generalize the transformation to the case of ISFSM's and prove the correctness of the reduction.

This reduction is of great interest because it allows to exploit existing efficient algorithms for prime generation [114, 53]. We will describe briefly in Section 11.2 efficient algorithms for generation of large sets of primes and report on their application to this problem.

[4] GPI cancellation when the present state part of the cancelling cube is full preserves encodeability because it actually relaxes input constraints.

```
-0 st0 st0 01    -0 100 100 01    -0 100 011 10    01 100 000 11
11 st0 st0 00    11 100 100 00    11 100 011 11    0- 010 000 01
01 st0 st1 --    01 100 010 00    01 100 101 00    -1 010 000 01
0- st1 st1 1-    0- 010 010 10    0- 010 101 00    01 001 111 11
11 st1 st0 0-    11 010 100 00    11 010 011 10
10 st1 st2 10    10 010 001 10    10 010 110 01
1- st2 st2 11    1- 001 001 11    1- 001 110 00
00 st2 st1 10    00 001 010 10    00 001 101 01
01 st2 ANY --
```

Figure 7.1: Covers of FSM *leoncino*

7.4.1 An Example

Fig. 7.1 shows on left a symbolic cover of an example of ISFSM, *leoncino*, that will be used throughout the exposition of GPI minimization. It is an ISFSM because there are some don't cares in the proper output part and one unspecified next state, denoted by ANY. In the tabular format, it is customary to omit transitions which have the next state and all proper outputs unspecified. The input variables of this symbolic function are the proper inputs and the present state; the output variables are the next state and the proper outputs.

An FSM can be interpreted as a multiple-valued function by representing both the present state and the next state with 1-hot encoding. For instance, use ESPRESSO with the keywords: *.mv 5 2 -3 -3 2, .type fr, .kiss*. The meaning is that the given FSM is a function with 5 multiple-valued variables, two of which are binary, two 3-valued and one 2-valued. Type fr specifies that a cube is in the offset of an output variable where a 0 appears [5].

The one-hot encoded representation of the onset, offset and dcset of *leoncino* are the second, third and fourth cover from left, respectively, of Fig. 7.1. The cover of the onset and offset are read directly from the input (since type fr is specified). By complementing the union of the covers of the onset and offset, a cover of the dcset is obtained [6]:

[5] As a matter of fact, ESPRESSO treats n binary output variables as one n-valued input variable; moreover, a s-valued next state variable and an n-valued proper output variable are replaced by one $s+n$-valued variable. In the example, the function has 4 multiple-valued variables, two of which are binary, one 3-valued and one 5-valued.

[6] Complementation is performed only with respect to the proper inputs and present state universe.

7.4.2 Definition of the Transformation

We will exhibit a multi-valued function whose primes are the GPI's of the FSM *leoncino*, modulo a post-processing step.

To do that define a function, called **companion function** of the symbolic function, with 4 multiple-valued variables, two of which are binary, one 3-valued and one 8-valued. We represent the companion function by a **companion cover** of the symbolic cover, constructed as follows. Transform the cover of the onset of the original function by transforming each cube into a **companion cube** in the following way:

1. represent with complemented 1-hot encoding the next state;

2. insert the complemented 1-hot encoding of the present state between the next state and the proper outputs.

Transform the cover of the dcset of the original function by transforming each cube into a **companion cube** in the following way:

1. insert the complemented 1-hot encoding of the present state between the next state and the proper outputs.

The transformed cover of the onset of the symbolic function is:

```
-0 100 01101101
11 100 01101100
01 100 10101100
0- 010 10110110
11 010 01110100
10 010 11010110
1- 001 11011011
00 001 10111010
```

The transformed cover of the dcset of the symbolic function is:

```
01 100 00001111
0- 010 00010101
-1 010 00010101
01 001 11111011
```

Finally, the companion function is the function represented by the companion cover obtained by joining the transformed covers of the onset and dcset of the symbolic function:

```
.mv 4 2 3 8
-0 100 01101101
11 100 01101100
01 100 10101100
0- 010 10110110
11 010 01110100
10 010 11010110
1- 001 11011011
00 001 10111010
01 100 00001111
0- 010 00010101
-1 010 00010101
01 001 11111011
```

In the next section we will show that the primes of this function are in 1-1 correspondence with the GPI's of the original FSM, modulo an easy post-processing step that deletes some primes. The primes of the companion MV function are shown in Fig. 7.2.

Some primes can be removed because they do not correspond to GPI's. Primes of one of the two following types are removed:

1. Primes that are covered by another prime, with full present state part and with the same next state and output tags. It is always better to select the covering prime since it induces no face constraint and covers the same minterms in the next state and output spaces.

2. Primes with full next state tag and null output tag. Since the next state tag is full, after encoding, it would be replaced by the intersection of all the codes, that is the all zero code, for any encoding. Therefore such a prime would not contribute to cover any minterm in next state spaces, nor in the output spaces (null output tag).

Fig. 8.3 shows the set of 26 GPI's obtained after post-processing.

7.4.3 Correctness of the Transformation

Theorem 7.4.1 *The computation of GPI's can be reduced to the computation of the primes of the companion multivalued function (MV primes) followed by a post-processing step that cancels 1) any MV prime contained by an MV prime with coinciding next state and output tags and whose present state part contains all the symbolic states and 2) any MV prime with a next state tag containing all the symbolic states and with a null output tag.*

```
.mv 4 2 3 8
.p 39
 0- 010 10110111
 01 001 11111011
 1- 001 11011011
 -1 001 11011011
 01 100 10101111
 01 110 10100111
 01 101 10101011
 01 011 10110011
 01 111 10100011
 -0 100 01101101
 0- 001 10111010
 -- 001 10011010
 0- 011 10110010
 0- 100 00101101
 11 010 01110101
 -1 010 00110101
 0- 110 00100101
 10 010 11010110
 -0 010 10010110
 10 011 11010010
 -0 011 10010010
 1- 100 01101100
 -- 100 00101100
 10 101 01001001
 11 011 01010001
 -1 011 00010001
 1- 010 01010100
 -- 010 00010100
 11 110 01100100
 -1 110 00100100
 1- 110 01000100
 -- 110 00000100
 0- 101 00101000
 1- 101 01001000
 -- 101 00001000
 1- 011 01010000
 -- 011 00010000
 0- 111 00100000
 1- 111 01000000
```

Figure 7.2: GPI's of FSM *leoncino* before post-processing

Proof: In the course of the proof we will refer to a symbolic cover (symbolic product-term) and an MV cover (MV product-term) as companion of each other if they are obtained by means of the previous transformation.

One must prove that for every GPI there is a prime of the function (modulo a post-processing step) and viceversa. In the sequel, unless otherwise stated, we will call MV primes those left after the post-processing step applied to the set of primes of the MV function. In [39] the rules for consensus and cancellation originally defined for binary cubes (e.g., in [94]) were extended to symbolic cubes. We call them GPI consensus and GPI cancellation. GPI's are defined as the fixed point of the computation that takes an initial symbolic cover and iteratively applies to it GPI consensus and cancellation. Primes of the companion MV function can be computed in different ways. They can be found as the fixed point of the computation that takes an initial MV cover and iteratively applies to it MV consensus and cancellation. We suppose that both fixed-point computations proceed as follows:
Start with the initial cover. For each pair of cubes in the cover, repeat until the cover does not change:

1. compute their consensus;
2. apply cancellation to the consensus cubes;
3. add the consensus cubes to the cover, unless their are cancelled by a cube already in the cover;
4. cancel any other cube covered by a consensus cube.

We show that at each step (and at fortiori at the end) of both fixed-point computations, performed respectively on the symbolic and MV cover, two companion covers are maintained.

We are now going to describe carefully and contrast consensus and cancellation in both domains.

GPI consensus. GPI consensus forms a $k+1$-cube from two k-cubes that either have

1. same binary-valued parts and different present state part; or
2. unidistant binary-valued parts and same present state part.

Merging two k-cubes forms a $k+1$-cube that has a next state tag that is the union of the two k-cubes' next state tags and an output tag that is the intersection of the outputs in the k-cubes' output tags. The binary inputs of the $k+1$-cube are obtained with the usual consensus rule for binary cubes. The

present-state part of the $k+1$-cube is the union of the present state parts of the k-cubes. Example of GPI consensus in case 1:

$$CONS(00\ st0\ \ st0\ 01; 00\ st2\ \ st1\ 10) = 00\ st0, st2\ \ st0, st1\ 00$$

Example of GPI consensus in case 2:

$$CONS(10\ st2\ \ st2\ 11; 00\ st2\ \ st1\ 10) = -0\ st2\ \ st2, st1\ 10$$

MV consensus. Consider two MV cubes $S = {X_1}^{S_1}{X_2}^{S_2}...{X_n}^{S_n}$ and $T = {X_1}^{T_1}{X_2}^{T_2}...{X_n}^{T_n}$. The intersection of S and T is the product-term

$$\bigcup_{1}^{n} {X_1}^{S_1 \cap T_1}{X_2}^{S_2 \cap T_2}...{X_n}^{S_n \cap T_n}$$

which is the largest product term contained in both S and T. If $S_i \cap T_i = \phi$ for some i, then $S \cap T = \phi$ and S and T are said to be disjoint. The distance between S and T equals the number of empty literals in their intersection. The consensus of S and T is the sum-of-products

$$\bigcup_{1}^{n} {X_1}^{S_1 \cap T_1}...{X_i}^{S_i \cup T_i}...{X_n}^{S_n \cap T_n}.$$

If the distance of S and T is ≥ 2 then their consensus is empty. If the distance of S and T is 1 and $S_i \cap T_i = \phi$, then their consensus is the single product-term

$${X_1}^{S_1 \cap T_1}...{X_i}^{S_i \cup T_i}...{X_n}^{S_n \cap T_n}.$$

If the distance of S and T is 0, then their consensus is a cover of n terms. Summarizing, MV consensus forms one cube from two MV cubes that have distance 1, and $k+2$ cubes - if k is the number of binary inputs - from two two MV cubes that have distance 0. Example of MV consensus of unidistant cubes:

$$CONS(00\ 100\ \ 01101101; 00\ 001\ \ 10111010) = 00\ 101\ \ 00101000$$

Example of MV consensus of 0-distant cubes:

$$CONS(00\ 110\ \ 00100101; 00\ 101\ \ 00101000) =$$

$$00\ 100\ \ 00100000, 00\ 100\ \ 00100000, 00\ 111\ \ 00100000, 00\ 100\ \ 00101101$$

Notice that the transformation rule for cubes in the onset ensures that the next state in the output field has a complemented 1-hot encoding so that MV intersection of encoded next states has the

same effect as GPI union of next state tags. For the same reason, the transformation rule for cubes in the dcset *does not* complement the 1-hot encoding of the next state in the output field. The following facts account for the asymmetry. There are two types of cubes in the dcset. The first type is generated by transitions with next state ANY, e.g.,: 01 $st2$ ANY $--$ (01 001 11111011). The fact that the next state is encoded by 111 means that the cube carries no information about the next state. When this cube is merged with other cubes at distance ≤ 1, it does not add any information to the next state, same as when taking consensus of the companion GPI's. The second type is generated by transitions with a specified next state and some unspecified proper outputs. These transitions generate a pair of cubes, one in the dcset and one in the onset, as follows: $0-$ $st1$ $st1$ $1-$, corresponding to $0-$ 010 00010101 in the dcset and $0-$ 010 10110110 in the onset.

When the onset and the dcset are joined, these two cubes are merged into one cube that corresponds to the original transition where all unspecified proper outputs have been set to 1, in agreement with the fact that the GPI's computed starting from onset f and dcset d coincide with the GPI's computed starting from onset $f + d$ and empty dcset (as it is true in general for the primes of a boolean function).The example shows the cube resulting from merging and the corresponding transition: $0-$ 010 10110111 ($0-$ $st1$ $st1$ 11).

GPI cancellation. A $k + 1$ cube cancels a k-cube if one of the following is true:

1. The binary input part of the $k + 1$-cube covers the binary input part of the k-cube.
2. They have the same present state part, and the next state and output tags are identical.
3. The present state part of the $k + 1$-cube contains all the symbolic states and the next state and output tags are identical.

The last case is part of the post-processing step in the MV domain. In addition, a cube with a next state tag containing all the symbolic states and with a null output tag is cancelled. This case too is part of the post-processing step in the MV domain, except when a MV prime has also a full present state part (then the present state field in the output part is all 0's).

MV cancellation. An MV cube contains another MV cube if the parts of the former contain the corresponding parts of the latter. An MV cube cancels another MV cube if it contains it. Notice that the present state field in the output part has been introduced to avoid MV cancellation when there is strict containment between present state parts, as shown here where the upper MV cube 10 011 11010010 (10 $st1, st2$ $st2$ 10) does not cancel the lower one 10 010 11010110 (10 $st1$ $st2$ 10), consistently with the GPI cancellation rule.

GPI consensus to MV consensus. Suppose that GPI consensus applies to two symbolic cubes. Does MV consensus apply to their companion MV cubes ? If so, does GPI consensus result in a symbolic cube whose MV companion is equal to the MV cube obtained by MV consensus ?

a Suppose that the two symbolic cubes have the same binary-valued parts and different present state part. The companion MV cubes may have distance 1 or distance 0.

a1 If the companion MV cubes have distance 1, GPI consensus works as MV consensus. Example of GPI consensus and companion MV cubes:

$$CONS(00\ st0\ \ st0\ 01; 00\ st2\ \ st1\ 10) = 00\ st0, st2\ \ st0, st1\ 00$$

$$CONS(00\ 100\ \ 01101101; 00\ 001\ \ 10111010) = 00\ 101\ \ 00101000$$

a2 If the companion MV cubes have distance 0, GPI consensus generates 1 consensus cube, while MV consensus generates $k + 2$ consensus cubes, if k is the number of proper binary inputs. But $k + 1$ consensus cubes are cancelled and the only one left is the companion cube of the symbolic consensus cube. Example of GPI consensus and companion MV cubes:

$$CONS(00\ st0, st1\ \ st0, st1\ 01; 00\ st0, st2\ \ st0, st1\ 00) = 00\ st0, st1, st2\ \ st0, st1\ 00$$

$$CONS(00\ 110\ \ 00100101; 00\ 101\ \ 00101000) =$$

$$00\ 100\ \ 00100000, 00\ 100\ \ 00100000, 00\ 111\ \ 00100000, 00\ 100\ \ 00101101$$

The first two terms are absorbed by the two original MV cubes, the third one is the companion MV cube of the result of GPI consensus. The fourth term is cancelled by another MV cube companion of a symbolic cube created by GPI consensus. In this example it is cancelled by $0-\ \ 100\ \ 00101101$, an MV prime whose companion symbolic cube is $0-\ \ st0\ \ st0, st1\ \ 01$.

b Suppose that the two symbolic cubes have unidistant binary-valued parts and same present state part. In this case the companion MV cubes have distance 1 and GPI consensus works as MV consensus.

MV consensus to GPI consensus. Suppose that MV consensus applies to two MV cubes. Does GPI consensus apply to their companion symbolic cubes ? If so, does MV consensus result in a MV cube whose symbolic companion is equal to the symbolic cube obtained by GPI consensus ?

1 Suppose that the two MV cubes have distance 1.

1a If they differ in a binary input, MV consensus works as GPI consensus

1b If they differ in the present state part, MV consensus works as GPI consensus.

1c If they differ in the output part, GPI consensus does not apply, while MV consensus does. But the MV consensus cube is cancelled by an already existing cube, so the net effect is the same in both cases

2 Suppose that the two MV cubes have distance 0. Apparently GPI consensus and MV consensus behave differently, but the same reasoning as in case **a2** of the analysis of GPI consensus to MV consensus shows that the net effect is the same.

GPI cancellation to MV cancellation. Suppose that GPI cancellation applies between two symbolic cubes. Does MV cancellation apply to their companion MV cubes ? Yes. GPI cancellation applies only when two symbolic cubes have the same present state part and the next state and output tags are identical. Obviously a containment relation is satisfied by binary-valued inputs. The companion MV cubes satisfy the same containment relation and MV cancellation applies too.

MV cancellation to GPI cancellation Suppose that MV cancellation applies between two MV cubes. Does GPI cancellation apply to their companion symbolic cubes ? There are cases when MV cancellation applies, but GPI cancellation does not. But they happen only when cancelling the last cube generated by MV consensus between cubes with distance 0. This MV cube has no symbolic companion and therefore the net effect is the same, as argued in case **a2** of the analysis of GPI consensus to MV consensus. Example of GPI consensus and companion MV cubes:

$$CONS(10\ st0, st1\ \ st0, st2\ 00; 10\ st0, st2\ \ st0, st2\ 01) = 10\ st0, st1, st1\ \ st0, st2\ 00$$

$$CONS(10\ 110\ \ 01000100; 10\ 101\ \ 01001001) =$$

$$10\ 100\ \ 01000000, 10\ 100\ \ 01000000, 10\ 111\ \ 01000000, 10\ 100\ \ 01001101$$

In this example the fourth term is cancelled by the MV cube $-0\ \ 100\ \ 01101101$. The companion symbolic cubes are respectively $10\ st0, st1, st2\ \ st0, st2\ 01$ and $-0\ st0\ \ st0\ 01$. Notice that the symbolic cube companion of the cancelling cube does not cancel the symbolic cube companion of the cancelled cube. But this cancellation in the GPI domain is not required because the companion symbolic cube of the cancelled MV cube is not generated by GPI consensus. ■

7.4.4 Definition of a Max-Min Family of Transformations

The transformation in Section 7.4.2 produces a function whose primes correspond to the GPI's, after a pruning step is applied to them. It is of practical interest to define functions whose primes correspond to a subset of the GPI's, in order to generate a part of the GPI's, when the whole set cannot be built or manipulated.

We are going now to define a family of such transformations. We remind that the onset of the companion function defined in Section 7.4.2 is obtained from the onset of the symbolic FSM by transforming each cube in the following way:

1. represent with complemented 1-hot encoding the next state;

2. insert the complemented 1-hot encoding of the present state between the next state and the proper outputs.

The dcset of this new function is obtained from the previous dcset by transforming each cube in the following way:

1. insert the complemented 1-hot encoding of the present state between the next state and the proper outputs.

Notice the key step of inserting the complemented 1-hot encoding of the present state between the next state and the proper outputs. This step avoids cancellation of cubes whose present state literal is included properly in another cubes' present state literal, since the former cube might be necessary for encodeability reasons. But suppose that, instead than inserting the complemented 1-hot encoding of the present state, we insert any literal that is contained in it. The effect is that some unwanted cube cancellation can take place, and therefore that we will get a proper subset of the GPI's. In the extreme limit we can replace the complemented 1-hot encoding of the present state with an empty cube and this will make possible the most of cancellation, producing the smallest subset of GPI's definable in this way. We call **maximal transformation** the one presented in Section 7.4.2 and **minimal transformation** the one with an empty subcube. The *Max-Min* family of transformations includes any transformation that for *any* cube in the original cover inserts between the next state and the proper outputs *any* literal included between the complemented 1-hot encoding of the present state and the empty literal. This proves the next statement.

Proposition 7.4.1 *Each transformation in the Max-Min family generates a subset of GPI's.*

7.5 Relation between GPI's and Primes of Encoded FSM's

In this section we demonstrate by examples the relation between GPI's and primes of encoded FSM's.

7.5.1 Minimum Cover of Encoded FSM and Minimum Cover of Encoded GPI's

We analyze the following two experiments:

1. Given a satisfying encoding, replace the codes in the FSM and minimize it (without *makesparse*, to obtain a minimum cover of primes).

2. Given a corresponding set of GPI's, replace the codes in the GPI's and minimize the resulting cover.

The two covers are the same, up to exceptions explained by the theory.

Encode the FSM *leoncino* with the following codes: $e(st0) = 00, e(st1) = 10, e(st2) = 11$. The encoded FSM is [7]:

```
-0 00 00 01
11 00 00 00
01 00 10 --
0- 10 10 1-
11 10 00 0-
10 10 11 10
1- 11 11 11
00 11 10 10
01 11 -- --
-- 01 -- --
```

A minimum cover of primes of the encoded FSM is:

```
-01- 1010
01-- 1011
-00- 0001
101- 1110
1--1 1111
```

2. The GPI's in the minimum encodeable solution are:

[7] One can omit the last two cubes and specify $.typefr$, which tells to ESPRESSO to put the unspecified input minterms in the dcset of all outputs.

```
3  1- st2          st2          11
5  01 st0,st1,st2  st1          11
6  -0 st0          st0          01
16 10 st1,st2      st2          10
17 -0 st1,st2      st1,st2      10
11 11 st1          st0          01
18 1- st0          st0          00
```

The encoded GPI's in the minimum encodeable solution are:

```
3  1- 11           11           11
5  01 --           10           11
6  -0 00           00           01
16 10 1-           11           10
17 -0 1-           10           10
11 11 10           00           01
18 1- 00           00           00
```

Add cube $--01\ ----$ (dcare conditions on 01, that is the code of $st3$, a state introduced as an artifact of encoding):

```
1- 11           11           11
01 --           10           11
-0 00           00           01
10 1-           11           10
-0 1-           10           10
11 10           00           01
1- 00           00           00
-- 01           --           --
```

A minimum cover of primes of the encoded GPI's is:

```
1--1 1111
01-- 1011
101- 1110
-11- 0001
-00- 0001
-01- 1010
```

This coincides with the previous minimum cover of primes of the encoded FSM, except for the cube $-11-\ 0001$, explained by all zeroes effect (see discussion in subsection 8.1 before). The previous transformations are summarized by Figure 7.3.

7.5.2 Primes of Encoded FSM vs. Primes of Encoded GPI's

GPI's can be seen as templates of the primes of every encoded FSM. The following two experiments clarify the statement:

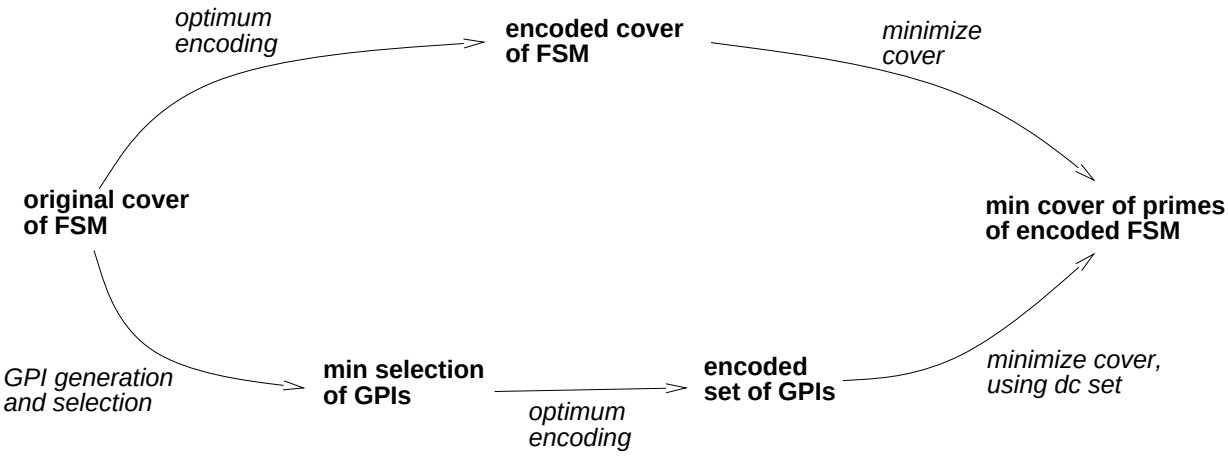


Figure 7.3: The circle of encodings

1. If one takes the primes of an encoded FSM and extracts the underlying GPI's, one gets a subset of the GPI's.

2. If one takes all the GPI's, encodes them with a given encoding and then raises them to primality in the encoding space (by removing the encoded GPI's that are not primes and expanding them with the appropriate dcset), one gets the primes of the encoded FSM (with the same encoding).

We illustrate the previous statements with examples. The previous transformations are summarized by Figure 7.4.

1. **From primes of the encoded FSM to GPI's.**

   The primes of the previous encoded FSM are:

```
1--1 1111
-1-1 1111
--01 1111
01-- 1011
---1 1010
0-10 1011
101- 1110
0-1- 1010
-01- 1010
-11- 0001
0-0- 0001
-00- 0001
```

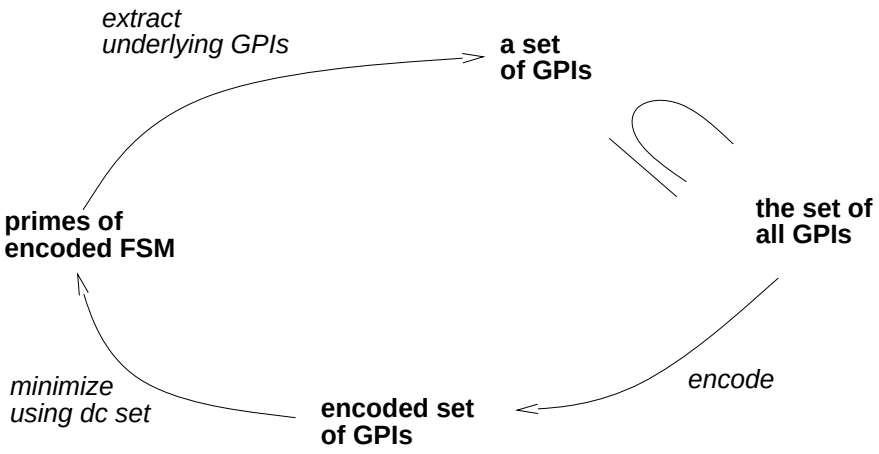


Figure 7.4: The circle of primes

```
0--0 0001
```

The companion MV cubes are:

```
1- st2,st3          st2 11
-1 st2,st3          st2 11
-- st3              st2 11
01 st0,st1,st2,st3  st1 11
-- st2,st3          st1 10
0- st1              st1 11
10 st1,st2          st2 10
0- st1,st2          st1 10
-0 st1,st2          st1 10
-1 st1,st2          st0 01
0- st0,st3          st0 01
-0 st0,st3          st0 01
0- st0,st1          st0 01
```

The corresponding GPI's are [8]:

```
3  1- st2          st2          11
4  -1 st2          st2          11
5  01 st0,st1,st2  st1          11
8  -- st2          st1,st2      10
1  0- st1          st1          11
16 10 st1,st2      st2          10
9  0- st1,st2      st1          10
17 -0 st1,st2      st1,st2      10
```

[8] There is no GPI for $--\ st3\ st2\ 11$ since it belongs to the don't care set of $st3$.

```
22 -1 st1,st2     st0,st1,st2 01
10 0- st0         st0,st1     01
6  -0 st0         st0         01
13 0- st0,st1     st0,st1     01
```

2. **From encoded GPI's to primes of the encoded FSM.**

The encoded GPI's are:

```
1  0- 10          10          11
2  01 11          --          11
3  1- 11          11          11
4  -1 11          11          11
5  01 --          10          11
6  -0 00          00          01
7  0- 11          10          10
8  -- 11          10          10
9  0- 1-          10          10
10 0- 00          00          01
11 11 10          00          01
12 -1 10          00          01
13 0- -0          00          01
14 10 10          11          10
15 -0 10          10          10
16 10 1-          11          10
17 -0 1-          10          10
18 1- 00          00          00
19 -- 00          00          00
21 11 1-          00          01
22 -1 1-          00          01
23 11 -0          00          00
24 -1 -0          00          00
25 0- --          00          00
26 1- --          00          00
```

Notice that there may be GPI's that cannot be encoded. For instance, the encoding of the MV literal $st0, st2$ of 20 : 10 $st0, st2$ $st0, st2$ 01 would be $--$, that includes also $st1$.

Notice that to establish a 1-1 correspondance with the primes of the encoded FSM, it is necessary to find the primes of the encoded GPI's, because some encoded GPI's subsume some other encoded GPI's, e.g. [9],

[9] GPI 14 and GPI 16 were kept, because it could be that no selection of GPI's that satisfies the input constraint $st1, st2$ is the smallest one, so that the smallest selection of encodeable GPI's would not include GPI 16, but might include GPI 14.

```
14 10 st1      st2         10
16 10 st1,st2  st2         10
14 10 10       11          10
16 10 1-       11          10
```

Notice also that, before computing the primes, one must add to the encoded GPI's the following cube $--01\ \ ----$, that specifies as don't care for all output functions the input minterms with present state 01, introduced as an artifact of the encoding. In general, one computes the primes of the cover that includes the encoded GPI's and all the global don't care minterms related to encoded present states not corresponding to symbolic present states of the original FSM. The primes of the encoded GPI's coincide with the primes of the encoded FSM and are:

```
1--1 1111
-1-1 1111
--01 1111
01-- 1011
---1 1010
0-10 1011
101- 1110
0-1- 1010
-01- 1010
-11- 0001
0-0- 0001
-00- 0001
0--0 0001
```

Now let us work out the example choosing a 1-hot encoding: $e(st0) = 100$, $e(st1) = 010$, $e(st2) = 001$.

1. **From primes of the encoded FSM to GPI's.**

   The primes of the encoded FSM and the corresponding GPI's are:

```
--11- 11111 dcset
--1-1 11111 dcset
---11 11111 dcset
01--1 11111 01 st2         -           11  gpi 2
--000 11111 dcset
0100- 11111 01 st2         -           11  gpi 2
1---1 00111 1- st2         st2         11  gpi 3
-1--1 00111 -1 st2         st2         11  gpi 4
01--- 01011 01 st0,st1,st2 st1         11  gpi 5
0--1- 01011 0- st1         st1         11  gpi 1
1-00- 00111 1- st2         st2         11  gpi 3
-100- 00111 -1 st2         st2         11  gpi 4
```

```
0-0-0 01011 0- st1          st1          11  gpi 1
-01-- 10001 -0 st0          st0          01  gpi 6
0-0-- 01010 0- st1,st2      st1          10  gpi 9
0---1 01010 0- st2          st1          10  gpi 7
----1 00010 -- st2          st1,st2      10  gpi 8
11-1- 10001 11 st1          st0          01  gpi 11
-0-00 10001 -0 st0          st0          01  gpi 6
0-1-- 00001 0- st0          st0,st1      01  gpi 10
-10-- 00001 -1 st1,st2      st0,st1,st2  01  gpi 22
-1-1- 00001 -1 st1          st0,st1      01  gpi 12
0---0 00001 0- st0,st1      st0,st1      01  gpi 13
100-- 00110 10 st1,st2      st2          10  gpi 16
10-1- 00110 10 st1          st2          10  gpi 14
-00-- 00010 -0 st1,st2      st1,st2      10  gpi 17
-0-1- 00010 -0 st1          st1,st2      10  gpi 15
--00- 00010 -- st2          st1,st2      10  gpi 8
1-1-- 10000 1- st0          st0          00  gpi 18
110-0 10001 11 st1          st0          01  gpi 11
10-0- 00001 10 st0,st2      st0,st2      01  gpi 20
11--0 10000 11 st0,st1      st0          00  gpi 23
1--00 10000 1- st0          st0          00  gpi 18
```

2. **From encoded GPI's to primes of the encoded FSM.**

The encoded GPI's are:

```
1  0- 010          010          11
2  01 001          ---          11
3  1- 001          001          11
4  -1 001          001          11
5  01 ---          010          11
6  -0 100          100          01
7  0- 001          010          10
8  -- 001          000          10
9  0- 0--          010          10
10 0- 100          000          01
11 11 010          100          01
12 -1 010          000          01
13 0- --0          000          01
14 10 010          001          10
15 -0 010          000          10
16 10 0--          001          10
17 -0 0--          000          10
18 1- 100          100          00
19 -- 100          000          00
```

```
20 10 -0-          000          01
21 11 0--          000          01
22 -1 0--          000          01
23 11 --0          100          00
24 -1 --0          000          00
25 0- ---          000          00
26 1- ---          000          00
```

The following global dcare minterms are added:

```
   --000           ---          --
   ---11           ---          --
   --11-           ---          --
   --1-1           ---          --
```

The primes of the encoded GPI's are:

```
--11- 11111
--1-1 11111
---11 11111
01--1 11111
--000 11111
0100- 11111
1---1 00111
-1--1 00111
01--- 01011
0--1- 01011
1-00- 00111
-100- 00111
0-0-0 01011
-01-- 10001
0-0-- 01010
0---1 01010
----1 00010
11-1- 10001
-0-00 10001
0-1-- 00001
-10-- 00001
-1-1- 00001
0---0 00001
100-- 00110
10-1- 00110
-00-- 00010
-0-1- 00010
--00- 00010
1-1-- 10000
```

```
110-0 10001
10-0- 00001
11--0 10000
1--00 10000
```

They coincide with the primes of the encoded FSM.

Summarizing, we point out that each GPI corresponds to various primes in different encoded FSM's, for instance, the GPI $3 \quad 1- \quad st2 \quad st2 \; 11$ corresponds to the following primes:

1. $1- -1 \;\; 1111$ in FSM encoded by $00, 10, 11$
2. $1- -0 \;\; 1011$ in FSM encoded by $01, 11, 10$
3. $1- - - 1 \;\; 00111$ in FSM encoded by $100, 010, 001$
4. $1- 0\,0- \;\; 00111$ in FSM encoded by $100, 010, 001$

In the last case it is noticeable that the same GPI corresponds to two different primes in the same encoded FSM (only one of them is needed for covering purposes, they differ in minterms of the don't care set of every output function). The number of GPI's is not only much smaller than the total number of primes over all encoded FSM's, but it may be even smaller than the number of primes of one encoded FSM, as the case of 1-hot encoding shows [10].

7.5.3 An Analysis Procedure

Given a symbolic FSM and an encoding (from which one derives the corresponding minimized encoded FSM), it may be of interest to study the encoding from the point-of-view of GPI analysis. For instance, if an encoding produces a very small cover, the analysis will reveal how the symbolic cover was mapped into such a compact representation. The previous discussion on the relation between GPI's and primes of a minimized encoded FSM can be put to use in devising a procedure that analyzes an encoding. Here we sketch the main steps. More specific information could be extracted to drive an intelligent heuristic search of a small encodeable cover of GPI's.

1. Encode and minimize the FSM, making sure that a cover of primes is returned [11].
2. Compute the set of GPI's.

[10] Notice that GPI's are the MV primes of the companion MV function, after post-processing.
[11] For instance, with ESPRESSO disable the step of *makesparse.*

3. Match the primes of the encoded minimized cover with the corresponding GPI's. To do this, given a prime, consider the present state subcube and find all the states included in it, then take away all hardware states that do not correspond to a state in the symbolic cover. As a result we have the proper input subcube and the set of present states. It is a fact that there is a unique GPI that has the same input subcube and the same set of states in the present state literal. It is the (only) one which corresponds to the given prime.

4. Derive the consistency equations of the given set of GPI's (for each minterm of the symbolic FSM, and for each GPI that covers it in the input part, add one term to the consistency equation of that minterm).

5. Derive the face constraints and check that they are satisfied.

As an example, consider the following encoded and minimized realization of the FSM *leoncino*:

```
-01- 1010
01-- 1011
-00- 0001
101- 1110
1--1 1111
```

The corresponding GPI's are:

```
17 -0 st1,st2     st1,st2     10
5  01 st0,st1,st2 st1         11
6  -0 st0         st0         01
16 10 st1,st2     st2         10
3  1- st2         st2         11
```

1. GPI 17 covers minterms 12,13,14,16;

2. GPI 5 covers minterms 1,2,3,4;

3. GPI 6 covers minterms 6,9,10;

4. GPI 16 covers minterms 7,8,20,21;

5. GPI 3 covers minterms 14,15,16,17,18,19;

6. minterms 5 and 11 are not implemented because st0 has zero code.

Chapter 8

Minimization of GPI's

8.1 Reduction of GPI Minimization to Unate Covering

Given all the GPI's, one must select a minimum encodeable subset of them that covers each minterm of the original FSM in the next state variables and in the proper output variables asserted by the minterm.

An approach reduces the problem to unate covering with encodeability and it has been proposed in [39]. A reduction to binate covering, where encodeability is translated into binate clauses, has been outlined in [133, 132]. Here we introduce the two approaches and discuss their respective merits. We start with reduction of GPI minimization to unate covering.

In [39] it is summarily proposed a modification of unate covering to solve the problem of selecting a minimum encodeable set of GPI's. Here we present a more complete version of it, clarifying issues arising in the case of state assignment. We will illustrate the discussion with the example *leoncino* shown in Fig. 7.1.

Minterms of the example. Minterms are product-terms where each literal is the characteristic function of a singleton. The minterms generated by the symbolic cubes of the previous cover are shown in Fig. 8.1. A $-$ means an empty next state tag. Given the semantics of ANY, no minterm is contributed by transition 01 $st2$ ANY $-$ $-$. It follows that no related encodeability constraint will be generated, ensuring that ANY of a missing transition is implemented by any possible hardware state. This is more than having all symbolic next states as possible, instead all hardware next states are possible (this is a point never mentioned in the literature). If we have multiple next states (non-deterministic FSM's), the minterm equations will have more choices. Notice that a minterm like 11 $st0$ $st0$ 00 does not need to be implemented (i.e., it is not in the onset of the

```
-0 st0 st0 01: 1  00 st0 -   01          00 100 00001
               2  00 st0 st0 00          00 100 10000
               3  10 st0 -   01          10 100 00001
               4  10 st0 st0 00          10 100 10000
11 st0 st0 00: 5  11 st0 st0 00          11 100 10000
01 st0 st1 --: 6  01 st0 st1 00          01 100 01000
0- st1 st1 1-: 7  00 st1 -   10          00 010 00010
               8  00 st1 st1 00          00 010 01000
               9  01 st1 -   10          01 010 00010
               10 01 st1 st1 00          01 010 01000
11 st1 st0 0-: 11 11 st1 st0 00          11 010 10000
10 st1 st2 10: 12 10 st1 -   10          10 010 00010
               13 10 st1 st2 00          10 010 00100
1- st2 st2 11: 14 10 st2 -   10          10 001 00010
               15 10 st2 -   01          10 001 00001
               16 10 st2 st2 00          10 001 00100
               17 11 st2 -   10          11 001 00010
               18 11 st2 -   01          11 001 00001
               19 11 st2 st2 00          11 001 00100
00 st2 st1 10: 20 00 st2 -   10          00 001 00010
               21 00 st2 st1 00          00 001 01000
01 st2 ANY --:
```

Figure 8.1: Minterms of FSM *leoncino*

```
1  00 100 11101101   00 st0 -   01                00 100 00001
2  00 100 01101100   00 st0 st0 00                00 100 10000
3  10 100 11101101   10 st0 -   01                10 100 00001
4  10 100 01101100   10 st0 st0 00                10 100 10000
5  11 100 01101100   11 st0 st0 00                11 100 10000
6  01 100 10101100   01 st0 st1 00                01 100 01000
7  00 010 11110110   00 st1 -   10                00 010 00010
8  00 010 10110100   00 st1 st1 00                00 010 01000
9  01 010 11110110   01 st1 -   10                01 010 00010
10 01 010 10110100   01 st1 st1 00                01 010 01000
11 11 010 01110100   11 st1 st0 00                11 010 10000
12 10 010 11110110   10 st1 -   10                10 010 00010
13 10 010 11010100   10 st1 st2 00                10 010 00100
14 10 001 11111010   10 st2 -   10                10 001 00010
15 10 001 11111001   10 st2 -   01                10 001 00001
16 10 001 11011000   10 st2 st2 00                10 001 00100
17 11 001 11111010   11 st2 -   10                11 001 00010
18 11 001 11111001   11 st2 -   01                11 001 00001
19 11 001 11011000   11 st2 st2 00                11 001 00100
20 00 001 11111010   00 st2 -   10                00 001 00010
21 00 001 10111000   00 st2 st1 00                00 001 01000
```

Figure 8.2: Extended representation of the minterms of FSM *leoncino*

next state variable) if $st0$ is assigned the all zeroes code.

We defined a companion MV function whose primes (modulo a post-processing step) are the GPI's of the original symbolic function. The product terms of the companion MV function that correspond to the minterms of the original symbolic function are shown in Fig. 8.2 [1]. The cover on the right shows the minterms represented with 1-hot encoding (and with the augmenting state set in the output part removed). We call the representation on the left extended representation and the one on the right reduced representation.

GPI's of the example. The GPI's of *leoncino* are shown in Fig. 8.3. The cover on the right shows the GPI's represented with 1-hot encoding (and with the augmenting state set in the output part removed). We call the representation on the left extended representation and the one on the right reduced representation.

The covering tables of the example. Now we can compute the covering table whose

[1] These cubes are not minterms of the companion function because the output variable has been augmented with one more state set and the states in the output variable are represented with complemented 1-hot encoding.

```
1  0- 010 10110111   0- st1         st1         11   0- 010 01011
2  01 001 11111011   01 st2         -           11   01 001 00011
3  1- 001 11011011   1- st2         st2         11   1- 001 00111
4  -1 001 11011011   -1 st2         st2         11   -1 001 00111
5  01 111 10100011   01 st0,st1,st2 st1         11   01 111 01011
6  -0 100 01101101   -0 st0         st0         01   -0 100 10001
7  0- 001 10111010   0- st2         st1         10   0- 001 01010
8  -- 001 10011010   -- st2         st1,st2     10   -- 001 01110
9  0- 011 10110010   0- st1,st2     st1         10   0- 011 01010
10 0- 100 00101101   0- st0         st0,st1     01   0- 100 11001
11 11 010 01110101   11 st1         st0         01   11 010 10001
12 -1 010 00110101   -1 st1         st0,st1     01   -1 010 11001
13 0- 110 00100101   0- st0,st1     st0,st1     01   0- 110 11001
14 10 010 11010110   10 st1         st2         10   10 010 00110
15 -0 010 10010110   -0 st1         st1,st2     10   -0 010 01110
16 10 011 11010010   10 st1,st2     st2         10   10 011 00110
17 -0 011 10010010   -0 st1,st2     st1,st2     10   -0 011 01110
18 1- 100 01101100   1- st0         st0         00   1- 100 10000
19 -- 100 00101100   -- st0         st0,st1     00   -- 100 11000
20 10 101 01001001   10 st0,st2     st0,st2     01   10 101 10101
21 11 011 01010001   11 st1,st2     st0,st2     01   11 011 10101
22 -1 011 00010001   -1 st1,st2     st0,st1,st2 01   -1 011 11101
23 11 110 01100100   11 st0,st1     st0         00   11 110 10000
24 -1 110 00100100   -1 st0,st1     st0,st1     00   -1 110 11000
25 0- 111 00100000   0- st0,st1,st2 st0,st1     00   0- 111 11000
26 1- 111 01000000   1- st0,st1,st2 st0,st2     00   1- 111 10100
```

Figure 8.3: GPI's of FSM *leoncino*

	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
1						x			x		x															
2						x			x		x					x									x	
3						x									x											
4						x								x	x	x										x
5														x	x								x	x		x
6				x					x		x				x									x	x	
7	x							x				x		x												
8	x							x			x		x												x	
9	x			x				x																		
10	x			x				x		x	x												x		x	x
11										x	x												x	x	x	x	
12												x	x	x	x												
13												x	x	x	x												
14			x					x						x	x												
15			x													x											
16			x					x						x	x		x										
17			x	x				x																		
18			x	x																						
19			x	x				x																		
20							x	x	x																	
21							x	x	x																	

Figure 8.4: Covering table of FSM *leoncino*

columns are GPI's and whose rows are minterms. One can use either the extended or the reduced representation for the GPI's and minterms. The extended representation has the advantage that column dominance, that requires same present state literal and next state tag (or next state tag of the dominating column as a subset of the next state tag of the dominated column), can be done simply by checking containment of the representations: a GPI (column) covers a minterm (row) iff the GPI contains the minterm. Notice that by construction the tag of a GPI may contain a superset of the next states in the tag of a covered minterm, but not a subset. When it contains a proper superset, the encodeability check tells whether the next state of the minterm can be produced by a column (or set of columns). The resulting table is shown in Fig. 8.4. The second column does not any intersect any row because it corresponds to a GPI that covers only points in the don't care set of the original function (from the unspecified transition 01 $st2$ – 11).

We call *next-state minterms* the minterms that assert a next state and *output minterms* the

minterms that assert a proper output. The *next-state minterms* insure that the correct next state is produced for a given input. The *output minterms* insure that the correct proper outputs are produced for a given input. The two types of minterms differ in the definition of when they are covered. Output minterms are covered as long as a GPI that contains them is selected. A row corresponding to a next-state minterm may require more than one column to be covered (*i.e.* to satisfy its encoding constraint) because each column may contribute only part of the next state (given that a GPI asserts as next state the conjunction of the codes of the states in its tag). Indeed if the tag of a column c is a proper superset of the tag of an intersected row r, then c might not be sufficient to cover r.

Each next-state minterm yields a constraint (or consistency equation) where the code of the next state is set equal to the disjunction of the conjunctions of the codes of the next states in the tags of the selected GPI's that cover the minterm. These output constraints have a special feature: the next state on the left side appears in all the conjunctions on the right side. This fact will be exploited to establish properties of the covering algorithm and to simplify the algorithm to check the satisfiability of constraints. Moreover, each GPI contributes an input constraint (the present states in its input part), albeit sometimes a trivial one.

The previous table cannot be used as an input to a covering routine because of the noted difference between next-state and output minterms. For instance, one cannot perform row-dominance between two rows of different kinds; e.g., in the previous table one cannot say that row 2 is eliminated by row 1, because row 2 is a next-state minterm and, even if row 1 (an output minterm) is covered, row 2 may still be unsatisfied after selecting one column that covers row 1 (in other words, the encoding constraint of row 2 may be unsatisfiable, given the current selection of columns). We will see that row dominance cannot be performed also between two rows each of which corresponds to a next state minterm. A way to handle the problem is to split the table into two tables: the (proper) *output table* and the *next-state table*. They have the same columns, the rows of the former are the output minterms and the rows of the latter are the next-state minterms.

It is possible to apply column dominance to the *combined* tables, if we restrict the ordinary definition of column dominance. The ordinary definition is that a column dominates another one if the former covers at least as many rows as the latter. *Restricted* column dominance holds iff ordinary column dominance holds, the two columns have the same present state and the next state tag of the dominating column is a subset (proper or not) of the next state tag of the dominated column. The reason is that such a dominating column covers at least as many output and next state minterms as the dominated column and contributes to the consistency equation of each covered next state minterm a term that bitwise dominates or is equal to the one contributed by the dominated

	0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2
	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
1	x x x x
3	x x
7	x x x x x
9	x x x x
12	x x x x
14	x x x x
15	x x
17	x x x x
18	x x x x
20	x x x x

Figure 8.5: Output covering table of FSM *leoncino*

column. For instance, if the tag of the dominating column is $\{st1, st2\}$ and the tag of the dominated column is $\{st1, st2, st3\}$, then $e(st1).e(st2) \geq e(st1).e(st2).e(st3)$, whatever encoding $e(.)$ is given to $st1, st2, st3$ [2]. Notice that restricted column dominance arises because of the next state table. Column dominance must be applied to the combined tables to guarantee the optimality of the solution.

In the cancellation rule of the consensus procedure to compute GPI's there is a condition that the next state tag of the cancelling GI must be equal to the next state tag of the cancelled GI. This is more restrictive than the condition for restricted column dominance requiring that the next state tag of the dominating column must be a subset (proper or not) of the next state tag of the dominated column. An interesting question is when it happens that a column covers at least as many rows of another column and its next state tag is a *proper* subset of the next state tag of the other column.

The output table is shown in Fig. 8.5. This table defines an ordinary unate covering problem. Here row dominance can be performed without conditions. Restricted column dominance can be applied to the combined tables. As a lower bound one can use the maximal independent set. This bound is looser than in standard unate covering because even if a solution can be found of cardinality equal to the lower bound, it may not satisfy the next state constraints.

The next-state table is shown in Fig. 8.6. This table defines a constrained unate covering problem. This table is covered iff some columns are selected that satisfy the encoding constraints (next state constraints, input constraints and uniqueness constraints). The next state constraints are

[2] In [39] the "same present state" condition is overlooked.

	0 1	0 2	0 3	0 4	0 5	0 6	0 7	0 8	0 9	1 0	1 1	1 2	1 3	1 4	1 5	1 6	1 7	1 8	1 9	2 0	2 1	2 2	2 3	2 4	2 5	2 6
2						x				x			x							x						x	
4						x												x	x	x						x
5																		x	x				x	x		x
6					x					x			x							x					x	x	
8	x							x					x		x		x								x	
10	x				x			x				x	x									x			x	x	
11											x	x									x	x	x	x		x
13														x	x	x	x									x
16			x					x								x	x			x						x
19			x	x				x													x	x				x
21							x	x	x								x								x	

Figure 8.6: Next-state covering table of FSM *leoncino*

a consistency equation for each row; to satisfy a consistency equation it is necessary that a column intersecting the related row is selected (*covering* problem), but in general it is not sufficient because of the interaction with the input and uniqueness constraints (*constrained* covering problem). Given the structure of the encoding constraints, they can always be satisfied by adding more columns to a given selection that solves the ordinary covering problem. [3]. Every input and uniqueness constraint yields a set of initial encoding dichotomies [116]. For each initial encoding dichotomy there is a companion one, where the same blocks of states appear moved from left to right and viceversa. Only one of two companion encoding-dichotomies must be satisfied. Next state constraints can be viewed as deleting encoding dichotomies. A removed encoding dichotomy is said to be *unsatisfied*, otherwise it is *satisfied*. We will show that by selecting enough columns that cover rows responsible of cancelling an encoding dichotomy, the latter can be satisfied. The goal is to choose the minimum number of columns such that the encoding constraints are satisfied (a necessary condition is that the next state table is covered). Row dominance is meaningless in the next-state table. Consider the example:

	r_1	r_2
c_1	x	x
c_2	x	

Even though column c_1 covers rows r_1 and r_2, we may have to choose also column c_2 to avoid

[3] Only input and uniqueness constraints were generated in the output table, so satisfiability of encoding constraints is always guaranteed there.

that the next state constraint of row r_1 removes an encoding dichotomy. Therefore removing row r_1 as row dominated by row r_2 would not guarantee a correct solution of the original problem. A new lower bound will be later defined, based on a maximal independent set of violated encoding dichotomies (similar to the notion of disjoint violations in [39]).

A solution of the example. Let us select a set of encodeable GPI's that cover the output and next state tables. The output table can be covered by choosing columns 3,5,6,17. In the next-state table two rows ($m5$ and $m11$) are not covered by columns 3,5,6,17; we choose column 18 to cover row 5 and column 11 to cover row 11. At this point the next state constraints are:

```
m2:  00 100  00 st0  st0 = st0
m4:  10 100  10 st0  st0 = st0
m5:  11 100  11 st0  st0 = st0
m6:  01 100  01 st0  st1 = st1
m8:  00 010  00 st1  st1 = st1.st2
m10: 01 010  01 st1  st1 = st1
m11: 11 010  11 st1  st0 = st0
m13: 10 010  10 st1  st2 = st1.st2
m16: 10 001  10 st2  st2 = st2 + st1.st2 (= st2)
m19: 11 001  11 st2  st2 = st2
m21: 00 001  00 st2  st1 = st1.st2
```

Only $m8$, $m13$ and $m21$ have non-trivial next state constraints. The only non-trivial input constraint is $(st1, st2)$, from column 17.

We now check if the previous constraints are satisfiable. The initial encoding dichotomies are: $(st1st2; st0)$, $(st0; st1st2)$, $(st1; st2)$, $(st2; st1)$. Next state constraint $st1 = st1.st2$ (from both $m8$ and $m21$) eliminates $(st2; st1)$; the reason is that this encoding dichotomy corresponds to an encoding bit where $st2$ is assigned 0 and $st1$ is assigned 1, but the disjunctive constraints $m8$ and $m21$ force $st1$ to be assigned 0, if $st2$ is assigned 0. For the same reason, next state constraint $st2 = st1.st2$ (from $m13$) eliminates $(st1; st2)$. Since both $(st1; st2)$ and $(st2; st1)$ are eliminated ($st1$ cannot be distinguished from $st2$) by $m8$, $m13$ and $m21$, a column that covers at least one of $m8$, $m13$, $m21$ is selected: e.g., we choose column 16 that covers row $m13$ (but does not cover $m8$ and $m21$). The previous constraints remain the same, except for the following update: $m13:\ 10\,010\ \ 10\,st1\ \ st2 = st2 + st1.st2\ (= st2)$. Notice that a new column could introduce a new input constraint, but column 16 does not. If we check again satisfiability, we notice that $(st1; st2)$ is not anymore removed by $m13$ and so we have an encodeable selection of GPI's that solves our original problem. An encoding that satisfies all constraints with a minimum code length is: $enc(st0) = 00, enc(st1) = 10, enc(st2) = 11$.

The GPI's in the final solution, together with the corresponding encoded GPI's, are:

```
3  1- st2          st2          11
   1- 11           11           11
5  01 st0,st1,st2  st1          11
   01 --           10           11
6  -0 st0          st0          01
   -0 00           00           01
16 10 st1,st2      st2          10
   10 1-           11           10
17 -0 st1,st2      st1,st2      10
   -0 1-           10           10
11 11 st1          st0          01
   11 10           00           01
18 1- st0          st0          00
   1- 00           00           00
```

By assigning to $st0$ the all zeroes code GPI 18 is not needed. It should be the case that also GPI 11 is not needed, because it covers only minterm 11 $st1$ in 11 $st1$ $st0$ 00, but it is not so. The reason being that GPI 11 is chosen to cover next state $st0$ in minterm 11 $st1$; it happens that GPI 11 when raised to primality expands also to the onset of a proper output, so that, when encoding GPI 11 with 00 for $st0$, it is not recognized that GPI 11 is useless. This motivates a later discussion on the necessity of repeating the minimization procedure to model the all zeroes code effect.

The all zeroes code issue. If a next state is encoded with all zeroes then the minterms with that next-state do not need to be covered by a GPI; in terms of the original FSM, one does not implement the product-terms with a next-state encoded with all zeroes and proper outputs all zeroes. This fact is not modelled by GPI's. For instance, the following minterms of the original FSM do not need to be implemented if $st0$ is assigned the code 00: 11 $st0$ $st0$ 00 and 11 $st1$ $st0$ 0$-$.

Knowing beforehand that those two minterms do not need to be realized may change the best solution. The only known way to cope with this problem is to repeat the previous procedure up to $N + 1$ times, if N is the number of next states; once as before [4], and then once for each next state, dropping from the original cover all the minterms producing a given next-state (called *reserved* state) and all zeroes as proper outputs. If all minterms belong to the onset of at least one binary-valued output, then there is no advantage in using an all-zero code and so only one covering must be made. If an all-zero code is already reserved, when at the end codes of minimum length that satisfy the encoding constraints of the optimal solution must be determined, one must take into account that the all-zero code cannot be used anymore. Suppose that the only set of codes of minimum length that satisfy the encoding constraints require a state to have the all-zero code. Then

[4] We do not know whether the best solution has at all a code with all zeroes.

we can add an encoding bit, setting it to 1 for all the codes but the reserved state that gets a 0. The encoding constraints will be satisfied by these new codes, with a penalty of one more encoding bit. Notice that even if we reach the conclusion that one of the states in a given optimal selection of k GPI's requires the all-zero code, there may be another selection of k GPI's where this does not happen. To find this other selection we should replace the current satisfiability check with a routine that tells whether the encoding constraints are satisfiable without using the all-zero code; in the worst-case, this can be achieved by exploring all codes that satisfy those constraints. But we are allowed to assign to the reserved state the all-zero code (not needed by the remaining states) without adding an encoding bit, if the reserved state was taken into account in the input and uniqueness constraints. In this way we are guaranteed to optimize also the secondary cost function (number of encoding bits). In practice this would be too expensive to compute, so we will only minimize the primary cost function (number of product-terms), adding one more encoding bit, when it is needed to handle the issue of the all-zero reserved code.

Summing up towards an exact algorithm. The problem is to select a minimum set of GPI's that cover the output table and satisfy the encoding constraints of the next state table. One can explore the space of solutions by solving the output table first and then computing its minimum extension to a solution of the next state table. This procedure is well-defined because of the following result:

Proposition 8.1.1 *Any solution of the output table can be extended to a global solution.*

Proof: Take the original FSM, replace each cube (asserting a next state) by a GPI that contains it and has the same present state literal and next state. This GPI exists because the rule for cancelling GI's requires the same present state literal and next state and, moreover, such a GPI is never cancelled by column domination because its next state tag cannot be a subset of another tag. This gives an upper bound on the number of GPI's necessary to cover the next-state table. These GPI's are compatible with the input constraints of any selected set of GPI's that covers the output table. The reason is that the suggested way of covering the next-state table yields only trivial output constraints (of the form $a = a + a.b + a.c + ...$) and whatever input constraints there are, they can always be satisfied (in the worst-case by 1-hot encoding). ■

The minimum of all such solutions solves exactly the original problem. In other words, for a given solution of the output table, we find the minimum set of GPI's which extends it to a solution of the next state table. This is the current best solution. One then goes back to the unate covering problem and finds a second solution to it, that in turn will be extended optimally to satisfy

the next state table, and so on, until an optimal solution to the global problem is found. When back to the unate covering problem we use as best current solution the best global one, not the best solution of the unate problem. Therefore if there is a solution of the output table, worse than the previous best solution of the output table, but such that it can be extended to a better global solution, it can be found. This guarantees that a global optimum is reached. At the end, when a minimum solution of the original problem has been found, codes of minimum length that satisfy the encoding constraints of the optimal solution must be determined.

In the output table we must solve an ordinary unate covering problem, for which well-known algorithms exist [114]. In the next state table we must solve a constrained covering problem: choose a minimum number of columns such that all encoding constraints are satisfied. An exact algorithm can be designed using a branch-and-bound scheme as for table covering. It is also helpful to maintain the same model of the problem as a table with a set of columns (GPI's) and rows (minterms). At each step a new column is chosen that extends the current partial solution to one that satisfies more the related encoding constraints. A key operation of the algorithm is to check whether a set of selected GPI's satisfies the related encoding constraints. If so, we have a complete solution, otherwise a new GPI must be selected and the feasibility check applied again. New criteria must be defined for selection of a branching column and for computing a lower bound. This constraint satisfaction problem can also be solved by a variant strategy in two steps: ordinary unate covering of the next state table, and then selection of more GPI's to satisfy the encoding constraints. In this variant the strategy of exploring the solution space is modified to favour choosing first GPI's that cover at least once every minterm of the next state table.

8.1.1 Exact Selection of an Encodeable Cover of GPI's

Figure 8.7 shows an exact algorithm to find a minimum selection of GPI's that is a cover of the original FSM and that is encodeable. The procedure is patterned on the branch-and-bound algorithm used to find an exact solution to unate covering [66].

Theorem 8.1.1 *The algorithm of Figure 8.7 finds a minimum cardinality selection of GPI's that is a cover of the original FSM and that is encodeable.*

Proof: The goal is to select columns of P to cover the output minterms and satisfy the encoding constraints induced by the next state minterms. The latter goal requires that for each next state minterm one or more GPI's are chosen so that the encoding constraints are satisfied. For this purpose at first columns are chosen until the next state minterms are satisfied, as certified by $encodeable$,

then a call to $mincov$ (a unate solver) selects a set of additional GPI's to complete the covering of the output minterms (if needed). This is done for each new solution to the next state minterms problem, i.e. each partial solution is extended optimally to a complete solution. The algorithm has the same control structure as the branch-and-bound procedure designed to solve exactly unate covering. Differences are:

1. In the table reduction step column dominance is restricted and row dominance is disallowed. Both restricted column dominance and detection of essentials are performed on the complete set of minterms to guarantee correctness.

2. The procedures to check encodeability ($encodeable$ [5]), and to compute a lower bound ($lbound$) and a branching column ($select\_column$) are specific to the problem. Designs of these procedures will be presented after that encodeability of GPI's will have been discussed in depth.

3. After invoking $mincov$ the current solution is bounded away, if the cost of the new complete solution is worse than the current upper bound.

The algorithm explores in the worst-case all solutions. At the beginning it reduces correctly the global matrix. It handles first the next state minterms and whenever it has found a new partial solution that satisfies all the encoding constraints, it extends it optimally to a complete solution. The bounding mechanism is the same as in the case of unate covering. It relies on a global upper bound, while a lower bound is computed only by considering the next state minterms. This weakens the lower bound, but guarantees correctness. Also the branching column is computed only by considering the next state minterms. The procedure $mincov$ is invoked on the output minterms table, after output minterms covered as a side-effect by the current partial solution are removed. The best solution in this table is found using a unate table solver. The current complete solution is compared against the upper bound [6]. ■

Notice that when dealing with next state minterms there is no notion of a covered minterm, but we speak instead of satisfied dichotomies, as it will be seen in detail later. Therefore next state rows are not deleted until all encoding constraints are satisfied.

[5] It replaces the simpler check that all rows of the matrix have been covered.

[6] When solving exactly unate covering, if the new lower bound is less than the upper bound and the table is empty, it means that a better solution has been found. Here, if $encodeable$ succeeds, we must compare again the complete solution with the upper bound, because in the previous comparison the new lower bound was not yet (in general) a complete solution, since the output minterms had not been covered yet (in general).

procedure exact_gpi_selection$(P, M_n, M_o, G, lbound, ubound)$ {

/* restricted column dominance, empty columns and essentials */

$P\_dom = restricted\_dominated\_columns(P, M_n \cup M_o)$

$P = P - P\_dom$

$G_e = essential(P, M_n \cup M_o)$

if $((cost(G_e) + cost(G)) \geq ubound)$ return$(\emptyset)$

else $G = G \cup G_e$

/* find lower bound from here to final solution by independent set */

$indep = lbound(P, M_n)$

/* make sure the lower bound is monotonically increasing */

$lbound\_new = MAX(card(G) + card(indep), lbound)$

/* bounding based on no better solution possible */

if $(lbound\_new \geq ubound)\ best\ = \emptyset$

/* check for new best solution */

else if $(encodeable(G, M_n))$ { /* new 'best' solution at current level */

$M_o = M_o - M_o.G$

$best_o = mincov(P, M_o, G, lbound\_new, ubound)$

if $((cost(best_o) + cost(G)) \geq ubound)\ best = \emptyset$

else $best = G \cup best_o$

} else { /* no more reductions: split and recur */

$pick = select\_column(P, M_n)$

/* branching column in the covering set */

$best1 = exact\_gpi\_selection(P - pick, M_n, M_o, G \cup pick, lbound\_new, ubound)$

/* update the upper bound if a better solution is found */

if $(best1 \neq \emptyset$ and $ubound > card(best1))\ ubound = card(best1)$

/* no branching if heuristic covering */

if $(best1 \neq \emptyset$ and $heuristic\_covering)$ return$(best1)$

/* no branching if lower bound matched */

if $(best1 \neq \emptyset$ and $card(best1) == lbound\_new)$ return$(best1)$

/* branching column not in the covering set */

$best2 = exact\_gpi\_selection(P - pick, M_n, M_o, G, lbound\_new, ubound)$

$best = solution\_choose\_best(best1, best2)$

}

return$(best)$

}

Figure 8.7: Exact selection of GPI's

8.1.2 Approximate Selection of an Encodeable Cover of GPI's

The bottleneck of the proposed exact selection algorithm is likely to be the very large number of branchings to guarantee exactness of the solution. Since the number of branchings is a (complex) function of the number of GPI's, one could try to restrict branching by generating or keeping only a subset of the GPI's. For instance a simple-minded heuristic would be to generate only the GPI's that have in the next state tag a number of states not larger than the logarithm of the number of states of the FSM. Another shortcut in the exact algorithm would be to stop at the first solution. In general an exact solution should make its quality more noticed in the case of next state intensive problems, i.e., state assignment problems whose final result depends strongly on the realization of the next states logic.

A different family of heuristics, presented in this section, starts with the complete set of GPI's, but selects a solution greedily, instead of bulding the full (branch-and-bound) computation tree. A fine tuning is required to trade-off efficiency vs. running time vs. ease of implementation.

Fig. 8.8 shows an approximate algorithm to find a selection of GPI's that is a cover of the original FSM and that is encodeable. The algorithm is approximate because it finds only one partial solution (that covers all minterms of the FSM) by invoking $unate\_encoding$ and then extends it to a complete solution by selecting greedily new GPI's needed to make the first selection encodeable. Since output and next state minterms together are fed to a standard unate table solver, "prohibited" table reductions may be carried on. In the step to extend the solution, minterms and GPI's that have been incorrectly discarded may be taken back in the solution, if needed. In opposition to the exact algorithm presented previously, this algorithm covers first optimally the output minterms and then extends greedily the partial solution to handle also the next state minterms. One could say that it is geared more towards output intensive problems. The exact and heuristic algorithms take the opposite view about covering next state minterms first vs. covering output minterms first. The following considerations discuss the issue.

1. In order to cover the output minterms first in the exact algorithm, one should modify a standard unate solver to restrict the table reduction operations. This was considered undesirable from an implementative point of view.

2. In the heuristic algorithm we did not take care of the next state minterms first, based on the expectation that it would have been less efficient than taking care of the output minterms first. The expectation is justified by the fact that we have an high quality unate table solver, not

procedure approx_gpi_selection(P, M_n, M_o) {

$G(i', p', n') = unate\_encoding(P, M_n + M_o)$

$G'(i', p', n') = P(i', p', n') - G(i', p', n')$

$unsat\_FID(x, y) = 1$

while $(unsat\_FID(x, y) \neq \emptyset)$ {

$GPI\_selected(i', p', n') = select\_column(G', M_n)$

$G(i', p', n') = G(i', p', n') + GPI\_selected(i', p', n')$

$G'(i', p', n') = G'(i', p', n') - GPI\_selected(i', p', n')$

$unsat\_FID(x, y) = encodeable(G, M_n)$

}

return(G)

}

Figure 8.8: Approximate selection of GPI's

likely to be matched in efficiency by a selector of encodeable GPI's. Experiments will assess the validity of this choice.

The simplified description of the algorithm highlights that it does a greedy search, by showing that after a call to $unate\_encoding$, one GPI at a time is chosen until the problem is solved. There is no backtacking to improve the solution (and no usage of a lower bound).

8.2 Reduction of GPI Minimization to Binate Covering

The encodeability check for a set of GPI's, given a bound on the number of encoding bits, was already formulated in [39] as a Boolean satisfiability problem.

The idea has been advanced further in [133, 132], to cast the whole problem of selecting a minimum encodeable cover of GPI's, for a fixed code-length, as a binate covering problem. An implementation has been described in [19]. A binate covering problem asks for the minimum solution of a formula written as a POS. Each literal in the POS can be chosen in the positive or negative phase in the solution and the cost of a solution is the sum of the cost of literals chosen in the positive phase, in the hypothesis that each literal has associated a cost (usually the cost is 1). In our case, the literals are the GPI's and the bits of the codes of the states; the cost of a GPI is 1 and the cost of a bit is 0. Choosing a literal of a GPI in positive phase corresponds to

selecting that GPI in the cover. Choosing a literal of a bit in positive or negative phase corresponds to setting it to 1 or to 0 in the encoding. In a sense, this reduction to binate covering lumps a genuine table covering problem (selecting a cover of GPI's) with a satisfiability problem (finding codes that satisfy constraints). Apparently this is appealing because everything is solved in a unique algorithmic frame, but the disadvantage is that a good algorithm for table covering may not be a good algorithm for satisfiability.

We will illustrate the reduction to binate covering using the same example *leoncino*. Suppose that we encode the states $st0$, $st1$ and $st2$ with 2 bits. The encoding bits are e_{01}, e_{02}, e_{11}, e_{12}, e_{21}, e_{22}, where e_{ij} is the j-th bit of the code of state i. We denote $e(sti)$ the code of state sti. We are going to build a binate table whose columns are the GPI's (denoted by g_i for $i = 1, \ldots, 26$) and the encoding bits (e_{ij}, $i = 0, 1, 2$, $j = 1, 2$). In our example there are 32 columns. The rows are clauses which state the conditions under which GPI's can be chosen to cover the minterms and an encoding compatible with them exists. There are clauses that express that next-state and output minterms are covered; other clauses represent input constraints induced by GPI's; finally, other clauses insure that a unique encoding is determined. We will now survey in detail each type of clauses.

The GPI's selected in the final cover must assert the same next state and proper outputs asserted by each minterm in the FSM. So we have clauses for both conditions.

For each next-state minterm, for all GPI's that cover it, we impose that the code of the next-state of the minterm is equal to the the disjunction of the conjunction of the next-states in the tags of the selected GPI's. Basically we read the next-state table and write-down an equation for each row. The big difference is that each row of the unate next-state table gives rise to *many* rows in the binate table, as the example shows.

$$
\begin{aligned}
e(st0) &= e(st0)(g_6 + g_{10}e(st1) + g_{13}e(st1) + g_{19}e(st1) + g_{24}e(st1)) \\
e(st0) &= e(st0)(g_6 + g_{18} + g_{19}e(st1) + g_{20}e(st2) + g_{26}e(st2)) \\
e(st0) &= e(st0)(g_{18} + g_{19}e(st1) + g_{23} + g_{24}e(st1) + g_{26}e(st2)) \\
e(st1) &= e(st1)(g_5 + g_{10}e(st0) + g_{13}e(st0) + g_{19}e(st0) + g_{24}e(st0) + g_{25}e(st0)) \\
e(st1) &= e(st1)(g_1 + g_9 + g_{13}e(st0) + g_{15}e(st2) + g_{17}e(st2) + g_{25}e(st0)) \\
e(st1) &= e(st1)(g_1 + g_5 + g_9 + g_{12}e(st0) + g_{13}e(st0) + g_{22}e(st0)e(st2) + g_{24}e(st0) + g_{25}e(st0)) \\
e(st0) &= e(st0)(g_{11} + g_{12}e(st1) + g_{21}e(st2) + g_{22}e(st1)e(st2) + g_{23} + g_{24}e(st1) + g_{26}e(st2) \\
e(st2) &= e(st2)(g_{14} + g_{15}e(st1) + g_{16} + g_{17}e(st1) + g_{26}e(st0)
\end{aligned}
$$

$$
\begin{aligned}
e(st2) &= e(st2)(g_3 + g_8 e(st1) + g_{16} + g_{17} e(st1) + g_{20} e(st0) + g_{26} e(st0) \\
e(st2) &= e(st2)(g_3 + g_4 + g_8 e(st1) + g_{21} e(st0) + g_{22} e(st0) e(st1) + g_{26} e(st0) \\
e(st1) &= e(st1)(g_7 + g_8 e(st2) + g_9 + g_{17} e(st2)
\end{aligned}
$$

Consider the first of the previous equations. It is equivalent to two equations in SOP:

$$
\begin{aligned}
e_{01} &= e_{01}(g_6 + g_{10} e_{11} + g_{13} e_{11} + g_{19} e_{11} + g_{24} e_{11}) \\
e_{02} &= e_{02}(g_6 + g_{10} e_{12} + g_{13} e_{12} + g_{19} e_{12} + g_{24} e_{12})
\end{aligned}
$$

or, equivalently:

$$
\begin{aligned}
&\overline{e}_{01} + g_6 + g_{10} e_{11} + g_{13} e_{11} + g_{19} e_{11} + g_{24} e_{11} \\
&\overline{e}_{02} + g_6 + g_{10} e_{12} + g_{13} e_{12} + g_{19} e_{12} + g_{24} e_{12}
\end{aligned}
$$

They can be rewritten in POS as:

$$
\begin{aligned}
&(\overline{e}_{01} + g_6 + g_{10} + g_{13} + g_{19} + g_{24})(\overline{e}_{01} + g_6 + e_{11}) \\
&(\overline{e}_{02} + g_6 + g_{10} + g_{13} + g_{19} + g_{24})(\overline{e}_{02} + g_6 + e_{12})
\end{aligned}
$$

Notice that a possible solution of these clauses is $e_{01} = e_{02} = 0$, and in that case no GPI is needed to cover minterm $m2$. This solves the problem of the all zeroes code that requires instead a clumsy repetition of minimizations in the unate reduction in subsection 8.1. The problem of efficient conversion from SOP to POS requires that one avoids generating duplicated and subsumed clauses. The point is illustrated by the following examples. Consider the SOP $a + bc + def$. It can rewritten as the following POS:

$$(a + b + d)(a + b + e)(a + b + f)(a + c + d)(a + c + e)(a + c + f).$$

Consider the SOP $a + bc + dc$, where some literals occur in more than one disjunct (literal c). It can be rewritten as:

$$(a + b + d)(a + b + c)(a + c + d)(a + c + c).$$

Taking away duplicated and subsumed clauses one gets:

$$(a + b + d)(a + c).$$

It is reported in [19] that a distributive method, which recursively generates clauses and immediately eliminates those duplicated and subsumed, reduces very effectively the number of clauses. The clauses of a reported example went down from $631,000$ to 184. No description of the algorithm is provided in the report. The only existing documentation is the code itself, that I have not yet read.

Summarizing, each next-state minterm equation yields some clauses to be added to the binate table. For instance, the next-state equation of minterm $m2$ yields four clauses. In the worst-case, if there are m minterms, the length of the code is k, each minterm involves g GPI's and each GPI has n next-states in its tag, we have $O(m.k.n^g)$ clauses. But in practice this number can be reduced to $O(m.k.n)$ if an efficient SOP to POS conversion is in place, given that the same next-states occur in many GPI's (this is elimination of dominated rows, in the binate covering formulation).

For each output minterm, one GPI that covers it must be selected. Basically each row of the output table translates into one row of the binate table. In our example, the first four clauses of this type are:

$$(g_6 + g_{10} + g_{13})$$
$$(g_6 + g_{20})$$
$$(g_1 + g_9 + g_{15} + g_{17})$$
$$(g_1 + g_5 + g_9)$$

Some clauses must enforce that the input constraint associated to each selected GPI is satisfied. This can be translated into the logical condition that, if a GPI is selected, each state not in the face must be assigned an opposite phase with respect to the states in the face in at least one encoding column. Input constraints with only one state or with all the states are trivial and no clauses are generated for them. In our example, face constraint $(st1, st2)$ is associated to GPI's 9,16,17,21,22, $(st0, st1)$ to GPI's 13,23,24 and $(st0, st2)$ to GPI 20. For instance, the logical condition to satisfy $st1, st2$, if GPI 9 is selected, is:

$$\overline{g}_9 + \overline{e}_{01}e_{11}e_{21} + e_{01}\overline{e}_{11}\overline{e}_{21} + \overline{e}_{02}e_{12}e_{22} + e_{02}\overline{e}_{12}\overline{e}_{22}.$$

A conversion from SOP to POS must be made. But in this case it happens rarely that simplifications can be made, differently from next-state covering clauses (the only simplification that occurs here is of clauses with a literal and its negation). The experimental fact is that these clauses are a bottleneck of the binate covering approach. For instance, [19] reports the following data: from $256,000$ clauses

for an FSM of 4 states and a code of length 2, to $11,764,900$ clauses for an FSM of 8 states and a code of length 3. In the worst-case, if the states are s, the length of the code is k, and there are f states in a face constraints, the number of clauses introduced by a GPI with a non trivial face constraint is $O((s-f).(f+1)^{2*k})$ clauses.

Some clauses insure that no pair of states are assigned the same code. In our case they are $e(st0) \neq e(st1)$, $e(st0) \neq e(st1)$ and $e(st1) \neq e(st2)$. The condition $e(st0) \neq e(st1)$, i.e., that the codes of the two states differ in at least one bit, is expressed by the following SOP:

$$e_{01}\overline{e}_{11} + \overline{e}_{01}e_{11} + e_{02}\overline{e}_{12} + \overline{e}_{02}e_{12}.$$

A conversion from SOP to POS is required. In the worst-case, if the states are s and the length of the code is k, the number of clauses to insure distinct codes is $O(2^{2k}.C_2^s)$. Other clauses insure that a state is not assigned more than one code. In our example, they are:

$$(e_{01} \oplus \overline{e}_{01})(e_{02} \oplus \overline{e}_{02})(e_{11} \oplus \overline{e}_{11})(e_{12} \oplus \overline{e}_{12})(e_{21} \oplus \overline{e}_{21})(e_{22} \oplus \overline{e}_{22})$$

In the worst-case, if the states are s and the length of the code is k, the number of clauses to insure unique codes is $O(k.s)$. All together these clauses make sure that an encoding is produced.

Once the binate table has been completed, one can use any binate solver to find a solution. In practice the size of the table is too large for available tools. Since both the number of columns (roughly, the number of GPI's) and of rows (even larger than the number of columns) become quickly very large, even approaches that solve binate covering by means of a shortest path computation of the clauses represented by BDD's as in [82, 62] have been unable to solve non-trivial instances. Indeed the methods in [82, 62] may succeed in handling huge numbers of clauses, but they are still limited by the numbers of columns, which are the support variables of the required BDD's.

8.3 GPI's and Non-Determinism

8.3.1 Symbolic Don't Cares and Beyond

In [39] mention is made of symbolic don't cares. In the state assignment context, they arise when more than one next state is allowed for a transition (don't care transitions). We introduced already such a situation when the next state is ANY, i.e. any of all the states, but a more general case is when the next state can be any of a subset of states. GPI's can be generated also for this more general case. Suppose that we have a don't care transition $i_1\ s_1\ s_0/s_2\ o_1$ where the next-state

s_0/s_2 means that s_0 and s_2 are both acceptable next states. We can replace the don't care transition by two care transitions $i_1\ s_1\ s_0\ o_1$ and $i_1\ s_1\ s_2\ o_1$, and then apply the algorithm for generating GPI's as in Subsection 7.3.1. One more rule is required to handle k-cubes with identical proper inputs and present states: a k-cube cancels another k-cube, if they have identical input parts and the tag of the first is a subset of the tag of the second. The reason is that the cancelling cube covers the same input subspace in *more* next-state spaces. The encodeability condition is modified, by replacing a single next state by a disjunction of next states in the consistency equation of each don't care minterm. The next states in the disjunction are those that appear in the original don't care transition that covers the given don't care minterm. Moreover, if a GPI corresponding to some next states being asserted by a symbolic don't care minterm is selected, other GPI's corresponding to different next states being asserted by the same don't care minterm cannot be selected in the cover [7].

Don't care transitions arise naturally in FSM's, as a way to represent different STG's that describe the same sequential function. A given STG is only *one* representation of a sequential function (a collection of input-output traces). An STG can be restructured in different ways and still represent the same sequential function. It is not known apriori which of these representations is the starting point leading to the most compact representation after state assignment. A state assignment procedure optimal for a sequential function (and not only for a given representation of the function) should capture all equivalent STG's and find out which one is the best to encode. No such a procedure is currently known in the general case. Currently it is common to do state minimization first and then to perform state assignment, since an STG with minimum number of states is usually a good starting point for state assignment. But it is a suboptimal procedure, as pointed out first in [55].

Given a CSFSM, state minimization returns a unique reduced FSM (up to state renaming). State minimization of CSFSM's merges all equivalent states into one state. Since it is not known apriori the amount of merging that produces the best starting point for state assignment, one can pass to the state assignment step the mergeability information, instead of the reduced FSM. The state assignment routine is given the task to decide the merging, while assigning the codes to the states. In this way, a family of equivalent STGs, complete under state merging and state re-direction, is explored during state assignment. Such a combined state minimization and encoding procedure has been proposed in [78], in the following form:

[7] Anticipating a future discussion, we say that the selection of encodeable GPI's reduces to binate covering and encodeability when there are don't care transitions. When don't care transitions are not present, one can reduce the selection of encodeable GPI's to unate covering and encodeability.

1. Identify equivalent states and implied state pairs.
2. Modify the FSM representation by letting the next-state of any transition be any one of the equivalent states.
3. In the state assignment step, allow equivalent states to have the same code (i.e., equivalent states need not have different codes) and assign the same code to all implied state pairs of equivalent states encoded with the same code.

This way of coupling state minimization and state assignment gives rise to don't care transitions, because some transitions have more than a possible next state.

Given an ISFSM, a state minimization procedure returns an ISFSM with a minimum number of states. Another way of looking to it is that state minimization of ISFSM's takes a family of CSFSM's and returns a subfamily of CSFSM's, all characterized by having the minimum number of states. We do not know of an exact procedure that explores at the same time state minimization and state assignment of ISFSM's. In [133], the problem of mapping the implied classes into compatibles in the reduced FSM (problem of unique mapped representation) has been modelled with don't cares transitions in the reduced FSM.

The introduction of don't care transitions is a special case of symbolic relations, pioneered in [82, 78]. Symbolic relations tie together the notion of GPI (that accounts for *symbolic* in the name) with the notion of relation. It is clear that with don't care transitions we have a relation, and not anymore a function, because the output response is a number of choices and not just one. They are symbolic relations because the output response is symbolic, so GPI's are required. To solve the symbolic relation problem, the notion of GPI's is extended to the one of generalized candidate primes (GCPI's), [8] similar to the notion of a candidate prime in a boolean relation. An appropriate covering problem is then set up, whereby a minimum subset of GCPI's is selected to implement the symbolic relation. We refer the reader to the references for a detailed presentation of the theory. It suffices to say here that covering with GCPI's involves a binate table covering step, while for covering with GPI's (constrained) unate table covering suffices.

In this thesis we only consider symbolic don't cares arising with a next state ANY. They can be handled in the frame of GPI's, without a need to extend the theory to GCPI's. Before leaving the topic of symbolic don't cares, we report an example from [133] of unique mapped representation modelled using GCPI's. Consider the ISFSM given in the table.

[8] A generalized candidate prime of a symbolic relation $\mathcal{R}$ is a cube $(\mathbf{c}|\sigma) \subseteq D \times \Sigma$ such that there exists an input encoding $\xi : D \rightarrow B^r$ and an output encoding $\psi : \Sigma \rightarrow B^n$ for which $(\xi(\mathbf{c})|\psi(\sigma))$ is a prime of a mapping $\mathbf{f} \prec_{\{\xi,\psi\}} \mathcal{R}$.

	0	1
1	1,0	2,0
2	–	4,1
3	1,–	2,0
4	1,1	5,0
5	3,0	3,1

A solution with minimum number of states can be formed with the compatibles:

$$a = \{1,3\}, b = \{2,5\}, c = \{3,4\}$$

When constructing the reduced FSM there is a choice for the next state of b under input 0 [9]. So, we get:

	0	1
a	a,0	b,0
b	(a,c),0	c,1
c	a,1	b,0

The STT of the reduced FSM is:

0	a	a	0
1	a	b	0
0	b	(a,c)	0
1	b	c	1
0	c	a	1
1	c	b	0

The primes of this symbolic relation are listed in Figure 8.9. Let the encoding of a be $l_{a1}l_{a2}$. Similarly for b and c. We now proceed to derive the covering constraints.

x = 0 a

$$(\overline{l}_{a1} + c_1 + c_7 l_{b1} + c_8 + c_9 l_{c1} + c_{10} + c_{19} l_{b1} + c_{20} + c_{21} l_{c1})$$
$$(\overline{l}_{a2} + c_1 + c_7 l_{b2} + c_8 + c_9 l_{c2} + c_{10} + c_{19} l_{b2} + c_{20} + c_{21} l_{c2})$$

x = 1 a

$$(\overline{l}_{b1} + c_2 + c_7 l_{a1} + c_{11} l_{c1} + c_{12} + c_{19} l_{a1} + c_{22} l_{c1})$$
$$(\overline{l}_{b2} + c_2 + c_7 l_{a2} + c_{11} l_{c2} + c_{12} + c_{19} l_{a2} + c_{22} l_{c2})$$

x = 0 b

$$(\overline{l}_{a1} + c_3 + c_8 + c_9 l_{c1} + c_{13} l_{c1} + c_{15} + c_{16} l_{c1} + c_{20} + c_{21} l_{c1})\overline{c}_{14}$$
$$(\overline{l}_{a2} + c_3 + c_8 + c_9 l_{c2} + c_{13} l_{c2} + c_{15} + c_{16} l_{c1} + c_{20} + c_{21} l_{c2})\overline{c}_{14} +$$
$$(\overline{l}_{c1} + c_9 l_{a1} + c_{13} l_{a1} + c_{14} + c_{16} l_{a1} + c_{21} l_{a1})\overline{c}_3\overline{c}_8\overline{c}_{15}\overline{c}_{20}$$
$$(\overline{l}_{c2} + c_9 l_{a2} + c_{13} l_{a2} + c_{14} + c_{16} l_{a2} + c_{21} l_{a2})\overline{c}_3\overline{c}_8\overline{c}_{15}\overline{c}_{20}$$

[9] The solution $a = \{1\}, b = \{2,5\}, c = \{3,4\}$ has no mapping options.

c_1:	0	a	a	0
c_2:	1	a	b	0
c_3:	0	b	a	0
c_4:	1	b	c	1
c_5:	0	c	a	1
c_6:	1	c	b	0
c_7:	–	a	$a \cap b$	0
c_8:	0	$a \cup b$	a	0
c_9:	0	$a \cup b$	$a \cap c$	0
c_{10}:	0	$a \cup c$	a	0
c_{11}:	1	$a \cup b$	$b \cap c$	0
c_{12}:	1	$a \cup c$	b	0
c_{13}:	–	b	$a \cap c$	0
c_{14}:	–	b	c	0
c_{15}:	0	$b \cup c$	a	0
c_{16}:	0	$b \cup c$	$a \cap c$	0
c_{17}:	1	$b \cup c$	$b \cap c$	0
c_{18}:	–	c	$a \cap b$	0
c_{19}:	–	$a \cup c$	$a \cap b$	0
c_{20}:	0	$a \cup b \cup c$	a	0
c_{21}:	0	$a \cup b \cup c$	$a \cap c$	0
c_{22}:	1	$a \cup b \cup c$	$b \cap c$	0

Figure 8.9: Primes of the symbolic relation.

x = 1 b

$$c_4$$

x = 0 c

$$c_5$$

x = 1 c

$$(\overline{l}_{b1} + c_6 + c_{12} + c_{17}l_{c1} + c_{18}l_{a1} + c_{19}l_{a1} + c_{22}l_{c1})$$
$$(\overline{l}_{b2} + c_6 + c_{12} + c_{17}l_{c2} + c_{18}l_{a2} + c_{19}l_{a2} + c_{22}l_{c2})$$

Let us turn now our attention to the face embedding constraints. The non-trivial face embedding constraints, the related sets of primes, and the corresponding constraints are:

$a \cup b$: c_8, c_9, c_{11}

$$\overline{c}_8\overline{c}_9\overline{c}_{11} + (l_{a1} \oplus l_{c1})(l_{b1} \oplus l_{c1}) + (l_{a2} \oplus l_{c2})(l_{b2} \oplus l_{c2})$$

$a \cup c$: c_{10}, c_{12}, c_{19}

$$\overline{c}_{10}\overline{c}_{12}\overline{c}_{19} + (l_{a1} \oplus l_{b1})(l_{c1} \oplus l_{b1}) + (l_{a2} \oplus l_{b2})(l_{c2} \oplus l_{b2})$$

$b \cup c$: c_{15}, c_{16}, c_{17}

$$\overline{c}_{15}\overline{c}_{16}\overline{c}_{17} + (l_{b1} \oplus l_{a1})(l_{c1} \oplus l_{a1}) + (l_{b2} \oplus l_{a2})(l_{c2} \oplus l_{a2})$$

Finally, the disjointness constraints are given by:

$$((l_{a1} \oplus l_{b1}) + (l_{a2} \oplus l_{b2}))((l_{a1} \oplus l_{c1}) + (l_{a2} \oplus l_{c2}))((l_{b1} \oplus l_{c1}) + (l_{b2} \oplus l_{c2}))$$

The optimum encoding is given by the minimum cost assignment satisfying the product of all the constraints derived so far. Putting these constraints in POS form results in a huge number of clauses, in spite of the simplicity of the example.

8.3.2 GPI's for Decomposition

Another application of GPI's is for the decomposition of FSM's into interconnected submachines. A formulation of FSM decomposition targeting two-level logic as *symbolic-output partitioning* has been proposed in [6]. The algorithm proposed requires the generation of GPI's of submachines and the solution of a constrained covering problem. We refer the interested reader to the original paper. Here the novel aspects of this application of GPI's are outlined.

Suppose that the problem is to decompose a given FSM, M, into two interconnected FSM's, M_1 and M_2, with the objective to minimize the total number of product terms in the minimized symbolic representations of the submachines. Let the number of product terms in the prototype machine, M, after one-hot coding and two-level logic minimization be P. Let the number of product terms in the submachines M_1 and M_2 after one-hot coding and two-level logic minimization be P_1 and P_2, respectively. An optimal decomposition minimizes $P_1 + P_2$. An upper bound to $P_1 + P_2$ is P, corresponding to the case when no good decomposition can be found and so the original FSM is not decomposed.

One decides a-priori a decomposition topology and a number of submachines. Outputs can be partitioned between the various submachines. Decomposition topologies vary from a general one where each submachine knows the state of every other submachine to a parallel one, where no submachine knows the state of any other submachine. Of course a given decomposition does not need to exist. A way to find decompositions is to come up with one partition of the original states for each submachine. These partitions must satisfy some properties to induce a functionally correct decomposition. The properties depend on the chosen topology. In the case of a general decomposition it is sufficient the minimum requirement that the product of the partitions be the zero-partition; for a parallel decomposition every partition must be closed. Instead of partitions one could look for set systems [56] (states may be in more than one block) and explore a larger solution space, but it is not done in the referred project.

Suppose that one looks for a general decomposition into two submachines (it always exists). Conceptually each state in the original FSM is split into the concatenation of two companion states. Two copies of the original FSM are made, where each copy is defined on one of the two sets of companion states. Since each copy reads the state of the other, it follows that each copy sees the global present state as in the original FSM. The outputs can be distributed between the two copies. For instance, all outputs can be given to one of the two copies.

The symbolic covers of the two submachines are then minimized. A multi-valued minimization of the 1-hot encoded covers would not yield any more information, than a multi-valued minimization of the 1-hot encoded original FSM. Instead the goal here is to find a pair of valid partitions (whose product is the zero-partition) such that the minimized multi-valued covers of the two submachines, where merged states are identified, are minimum. [10]. Now enter the GPI's.

[10] Multi-valued minimization of a 1-hot encoded cover is a concise way of saying that multi-valued minimization of a symbolic cover returns a minimized multi-valued cover, that can be realized with an equivalent two-valued cover by 1-hot encoding the minimized multi-valued cover; alternatively, one can say that 1-hot encoding of a symbolic cover followed by two-valued minimization is equivalent to multi-valued minimization of a symbolic cover followed by 1-hot encoding

Suppose that we compute the set of GPI's of each submachine (this is more than computing the set of PI's of each submachine). We know that each GPI carries a next state tag whose interpretation is that the encoded GPI produces as next state the intersection of the codes of the states in the tag. Since we are computing a bound when the encoding is one-hot, the bitwise intersection of the codes of two states is null unless they have the same code. Therefore the tag of each GPI forces to merge the states in it into one. So we can use the next state tags of the GPI's to explore all possible partitions. The selection of two minimum sets of GPI's which induce valid partitions and such that the mergings forced by their tags do not conflict with the input constraints induced by their present state literals gives two submachines whose 1-hot encoded implementations have a total minimum cardinality. Different topologies induce different requirements on the selections of GPI's that yield correct decompositions. Notice that there may be codes shorter than 1-hot and still satisfying the input constraints and merging conditions, but here we are not interested in encoding the states, but in decomposing the original FSM by means of a preprocessing step. [11].

The two selected sets of GPI's define the symbolic covers of the two submachines. Each state in the present state literal of a GPI in a submachine denotes the pair of companion states of both submachines: one is the present state of the current submachine and the other state is an input from the other submachine. Each state of a submachine is replaced by the representative of the equivalence class to which it belongs. The two symbolic covers must now be encoded. Since each submachine reads as input the state of the other submachine, the state assignment routine should take into account such an interaction between the two submachines. This is an instance of state assignment of a network of FSM's, for which no good algorithm is known up to now. It is not mentioned in [6] how the problem of encoding mutually interacting FSM's has been solved in the reported experiments. It is only stated that a state-of-art state assignment tool for single FSM's (*nova*) has been somehow used.

The following example shows the main steps of a decomposition. The original FSM is:

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

of the minimized symbolic cover.

[11] If GPI's of each submachine are used without this restriction on a chosen encoding, then the GPI's of each submachine would carry the same information as the GPI's of the original FSM. GPI's are independent of an encoding (GPI's are used to find an optimal encoded cover), but here they are used with a presupposed encoding, so that here GPI minimization is equivalent to multi-valued minimization and simultaneous exploration of the partitions.

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

The two copies are:

```
                0 sa1 sb1   sa2
                1 sa1 sb1   sa3
                0 sa2 sb2   sa3
(1)             1 sa2 sb2   sa4
                0 sa3 sb3   sa3
                1 sa3 sb3   sa4
                0 sa4 sb4   sa2
                1 sa4 sb4   sa1

                0 sa1 sb1   sb2 1
                1 sa1 sb1   sb3 1
(2)             0 sa2 sb2   sb3 0
                1 sa2 sb2   sb4 0
                0 sa3 sb3   sb3 0
                1 sa3 sb3   sb4 0
                0 sa4 sb4   sb2 1
                1 sa4 sb4   sb1 1
```

They have the following minimum covers of GPI's:

```
                - 1000 (sa1 sa2 sa3)
(3)             0 1111 (sa1 sa2 sa3)
                - 0001 (sa1 sa2 sa3)
                1 0110 (sa4)

                0 1000 (sb2)       1
(4)             1 1000 (sb3 sb4)   1
                - 0110 (sb3 sb4)   0
                0 0001 (sb2)       1
                1 0001 (sb1)       1
```

Replace the present state literals in (3),(4) with a concatenation of the codes of the submachines. The covers are:

```
                - sa1 sb1  (sa1,sa2,sa3)
                0 sa1 sb1  (sa1,sa2,sa3)
(3')            0 sa2 sb2  (sa1,sa2,sa3)
                0 sa3 sb3  (sa1,sa2,sa3)
                0 sa4 sb4  (sa1,sa2,sa3)
```

```
                - sa4 sb4  (sa1,sa2,sa3)
                1 sa2 sb2  (sa4)
                1 sa3 sb3  (sa4)

                0 sa1 sb1  (sb2)     1
(4')            1 sa1 sb1  (sb3,sb4) 1
                - sa2 sb2  (sb3,sb4) 0
                - sa3 sb3  (sb3,sb4) 0
                0 sa4 sb4  (sb2)     1
                1 sa4 sb4  (sb1)     1
```

Replace each state by a representative of its equivalence class ($sa1, sa2, sa3$ are one class represented by $sa1$; $sb3, sb4$ are one class represented by $sb3$). The final symbolic covers are:

```
                - sa1 sb1  sa1
(3'')           0 sa1 sb2  sa1
                0 sa1 sb3  sa1
                - sa4 sb3  sa1
                1 sa1 sb2  sa4
                1 sa1 sb3  sa4

                0 sa1 sb1  sb2 1
(4'')           1 sa1 sb1  sb3 1
                - sa1 sb2  sb3 0
                - sa1 sb3  sb3 0
                0 sa4 sb3  sb2 1
                1 sa4 sb3  sb1 1
```

An optimal state assignment of both submachines should take into account their interactions. For instance, when encoding submachine (3") the symbolic input sb appears as state variable in submachine (4").

In summary, decomposition does not carry out the complete encoding of the states, it merely 'preprocesses' them so that the subsequent state encoding applied on the preprocessed set of states will be guaranteed to realize the decomposition with the desired topology. The decomposition problem is simpler than the classical state assignment problem since a one-hot coding has already been assumed, and the only degree of freedom is in giving the same code to the states. It is interesting to mention that state encoding can be viewed as the problem of finding an optimal decomposition of the prototype machine into as many submachines as there are state bits in the final state encoding. The number of submachines (number of bits), topology of interconnections and distribution of the proper outputs are all unknowns that an optimal state assignment decides , thereby producing an optimal decomposition.

Redecomposition of interconnected FSM's via GPI's is briefly touched upon in [5]. The claim is that one can generate the GPI's of the submachines and after some operations deduce from them GPI's of the overall FSM. These operations are described very briefly and are not clear, but the point made is that one can explore the GPI's of the overall FSM, without a need to flatten the FSM network into a lumped FSM to compute them. This corresponds to a re-encoding/re-partitioning of the initial implementation.

Chapter 9

Encodeability of GPI's

9.1 A Theory of Encodeability of GPI's

We present a theory of encodeability of GPI's based on the notion of *raising graphs* and *updating sets*. It is at the core of new algorithms for the computation of a branching column and of a lower bound to be used in a branch-and-bound scheme to find a minimum encodeable cover of GPI's.

9.1.1 Efficient Encodeability Check of GPI's

A set of selected GPI's and the original cover of the FSM yield a set of constraints: the input constraints of the GPI's, the uniqueness constraints and the next state constraints. Then one must find if the selected GPI's are encodeable, i.e., if there is an assignment of codes to states such that all associated encoding constraints are satisfied. If so, codes of minimum length that satisfy the constraints must be found in order to convert the cover of encodeable GPI's into a two-valued cover that implements the original FSM. Theory and algorithms to check satisfiability of encoding constraints have been proposed in [116], to which we refer for details. Here we review necessary definitions and theorems. Moreover, we present novel results on encodeability of GPI's that will be the basis for a new feasibility check algorithm very suitable for a BDD-based representation. We suppose that a set of GPI's and an FSM cover (therefore a set of states and a set of next state constraints) are given.

An **encoding dichotomy** (or, more simply, **dichotomy**) $i = (l; r)$ is a 2-block partition of a subset of the states to be encoded. The states in the left block are associated with the bit *0* while

those in the right block are associated with the bit *1*. If a dichotomy is used in generating codes, then a code bit of the states in the left block is assigned *0* while the same code bit is assigned *1* for the states in the right block. For example, $(ab; cd)$ is an dichotomy in which a and b are associated with the bit *0* and c and d with the bit *1*.

A face constraint yields pairs of **initial dichotomies** (**ID**). For instance, given the states a, b, c, d, e and the face constraint (abc), the initial dichotomies are $(abc; d)$, $(d; abc)$, $(abc; e)$, $(e; abc)$. Since dichotomies have a fixed *0* or *1* assignment, for each of them there is a dual one where the blocks are switched. So there is natural equivalence relation $\tilde{I}$ on the initial dichotomies I (**duality_equivalence**). Two initial dichotomies are in the same class iff they are dual of each other (the dual of the dual of dichotomy i is i). In the example there are two classes $\{(abc; d), (d; abc)\}, \{(abc; e), (e; abc)\}$. A class $\tilde{i} = \{i_L, i_R\}$ of the duality equivalence relation is called a **free initial dichotomy** (**FID**). A FID can be represented by either initial dichotomy that is in the class. Uniqueness constraints too generate initial dichotomies.

A dichotomy $i' = (l', r')$ **orderly** or **block-wise covers** another dichotomy $i = (l, r)$, noted as $i' \supseteq i$, iff $l' \supseteq l$ and $r' \supseteq r$. Notice that this definition differs from the one given in [116], where it is said that a dichotomy i_1 *covers* a dichotomy i_2 if the left and right blocks of i_2 are subsets respectively either of the left and right blocks, or of the right and left blocks of i_1. We reserve instead this unordered definition of covering to the case of a dichotomy covering a free initial dichotomy. A dichotomy $i' = (l', r')$ **unorderly covers** a FID $i = (l, r)$, noted as $i' \supseteq i$, iff $l' \supseteq l$ and $r' \supseteq r$ or $l' \supseteq r$ and $r' \supseteq l$. We will often drop the qualification and simply say "covers", when it will be clear from the context which one is meant.

A dichotomy is **complete** if each state appears in either block. A **completion** of a dichotomy $i = (l, r)$ is a dichotomy $c(i) = (c(l), c(r)$ such that $c(l) \supseteq l$, $c(r) \supseteq r$, $c(l) \cap c(r) = \emptyset$, $c(l) \cup c(r) = U(l, r)$, where U is the universe set.

A dichotomy **violates** a next-state encoding constraint if the encoding bit generated for the states in the dichotomy does not satisfy the bit-wise requirements of the next-state encoding constraint. A **valid dichotomy** is one that does not violate any next-state encoding constraint. The notion of valid and complete dichotomy coincides with the notion of prime dichotomy proposed in [116], but here we will not use the latter term since we do not rely on iterated union to generate valid and complete dichotomies.

A dichotomy is **raised** by adding states into either its left or right block as implied by the next-state encoding constraints. A dichotomy is said to be **maximally raised** if no further states can be added into either the left or right block by the next-state encoding constraints.

procedure check_feasible (*constraints*) {
    I = generate_initial_dichotomies (*constraints*)
    D = copy(I)
    foreach (dichotomy d in D)
        raise_dichotomy (d, *constraints*)
    D = remove_invalid_dichotomies (D, *constraints*)
    $\tilde{I}$ = duality_equivalence (I)
    foreach (free dichotomy $\tilde{i} = \{i_L, i_R\} \in \tilde{I}$)
        if ($\tilde{i}$ is not covered by $d(i_L) \in D$ or by $d(i_R) \in D$)
            return (*INFEASIBLE*)
    return (*FEASIBLE*)
}

Figure 9.1: Encodeability check

The procedure *check_feasible* (modified from [116]) generates initial dichotomies from face constraints and uniqueness constraints, raises and deletes them using the next state constraints (procedures *raise_dichotomy* and *remove_invalid_dichotomies*) and finally reports the unsatisfied initial dichotomies.

Given an initial dichotomy $i = (l; r)$ and a next state constraint e, the procedure *raise_dichotomy* defines two raising rules:

1. If one child of each conjunction of e is in the left block of i, then insert the parent s of e into the left block of i (*left raising rule*).

2. If one child of all but one conjunction of e is in the left block of i and the parent s of e is in the right block of i, then insert all children of the remaining conjunction of e into the right block of i (*right raising rule*).

For a given e and i at most one of the two rules is applicable, because the conditions for applicability are contradictory. To model the semantics of an empty next-state constraint (e.g., $a = \emptyset$), it is stipulated that in *remove_invalid_dichotomies* any dichotomy d with the parent of the constraint in the right block is removed. In *raise_dichotomy* an empty next-state constraint does not force any raising.

Given an initial dichotomy $i = (l; r)$, we denote by $d(i) = (d(l); d(r))$ the maximally

```
procedure remove_invalid_dichotomies (D, constraints) {
    foreach (dichotomy d ∈ D)
        foreach (next-state constraint e)
            if (parent in right block of d &
                one child of each conjunction in left block of d)
                delete d
}
```

Figure 9.2: Detection of invalid dichotomies

```
procedure raise_dichotomy (d, constraints) {
    do {
        foreach (parent s in a next-state constraint e)
            if (one child of each conjunction in left block of d)
                insert s into left block of d
            if (one child of all but one conjunction in left block of d &
                s in right block of d)
                insert all children of remaining conjunction into right block of d
    } while (at least one insertion within loop)
}
```

Figure 9.3: Raising of dichotomies

raised dichotomy that *raise_dichotomy* generates. This definition is well-posed, because if a dichotomy i is given as an input to *raise_dichotomy*, a unique $d(i)$ is returned, according to the order of application of the next-state constraints and rules.

An **initial dichotomy** is **satisfied** if there is a valid maximally raised dichotomy that covers it. A **free initial dichotomy** is **satisfied** if at least one of the two initial dichotomies in it is satisfied [1]. We will show that any maximally raised dichotomy that covers an ID i is invalid if $d(i)$ is invalid. Therefore, a free initial dichotomy is unsatisfied if *raise_dichotomy* obtains invalid maximally raised dichotomies from both of its two initial dichotomies. One says also that a free initial dichotomy $\tilde{i} = \{i_L, i_R\}$ (or an initial dichotomy i) is **violated** by the next state constraints that are responsible for the deletion of $d(i_L)$ and $d(i_R)$ (of $d(i)$). Summarizing, next state constraints remove raised dichotomies and so they violate initial dichotomies, and therefore some face or uniqueness constraints cannot be satisfied.

We now prove that it is sufficient to check whether $d(i)$ is invalid to determine if $\exists i'$ such that i is covered by $d(i')$. This proves that *check_infeasible* detects correctly infeasibility.

Theorem 9.1.1 *Given an initial dichotomy i and the corresponding maximally raised dichotomy $d(i)$. If $d(i)$ is invalid, i cannot be covered by another maximally raised dichotomy $d(i')$, unless $d(i')$ is invalid too.*

Proof: We prove first that if $\exists i' = (l'; r'), i' \neq i$, such that $(l; r) \subseteq d(i') = (d(l'); d(r'))$, i.e. $(l; r)$ is covered by a maximally raised dichotomy $d(i')$, then $d(i) = (d(l); d(r)) \subseteq d(i') = (d(l'); d(r'))$. Suppose that some raising steps are needed to maximally raise $(l; r)$ to $(d(l); d(r))$; we prove the statement by induction on the number k of raising steps. Denote the dichotomy $(l; r)$, after the application of the first k raising steps, as $(l_k; r_k)$. The statement is true for $k = 0$, since if $(l; r) = (d(l); d(r))$, i.e. i is already maximally raised, then $(d(l); d(r)) \subseteq (d(l'); d(r'))$. Suppose that it is true for k, i.e. $(l_k; r_k) \subseteq (d(l'); d(r'))$, we want to show that it holds for $k + 1$, i.e that $(l_{k+1}; r_{k+1}) \subseteq (d(l'); d(r'))$. Either the left raising rule or the right raising rule is applied to go from $(l_k; r_k)$ to $(l_{k+1}; r_{k+1})$. If the left raising rule for next-state constraint e with parent p is applied, then $(l_{k+1}; r_{k+1}) = (l_k \cup \{p\}; r_k)$. Since $l_k \subseteq d(l')$, e is applicable also to $(d(l'); d(r'))$ and so it inserts p in the left block $d(l')$. But, since $(d(l'); d(r'))$ is maximally raised, p is already in $d(l')$, and so $(l_{k+1}; r_{k+1}) \subseteq (d(l'); d(r'))$. If the right raising rule for next state constraint e with parent p and uncommitted conjunct $b_1 \ldots b_m$ is applied, then $(l_{k+1}; r_{k+1}) = (l_k; r_k \cup \{b_1 \ldots b_m\})$. Since $l_k \subseteq d(l')$, and $r_k \subseteq d(r')$, e is also applicable to $(d(l'); d(r'))$, unless one child of $b_1 \ldots b_m$ is

[1] Note that "satisfied" is here an overloaded word.

in $d(l')$, making $(d(l');d(r'))$ invalid, because p is in $d(r')$. But, since $(d(l');d(r'))$ is maximally raised, $b_1 \ldots b_m$ is already in $d(r')$ and so $(l_{k+1};r_{k+1}) \subseteq (d(l');d(r'))$.

Finally we prove that if $(d(l);d(r))$ is invalid, then $(d(l');d(r'))$ is invalid too. Suppose that $(d(l);d(r))$ is removed by e, then the parent of e must be in $d(r)$ and one child of each conjunct must be in $d(l)$. Since we proved previously that $(d(l);d(r)) \subseteq (d(l');d(r'))$, then the parent of e must be also in $d(r')$ and one child of each conjunct must be in $d(l')$, i.e. e removes also $d(i') = (d(l');d(r'))$. ■

We will now look into the properties of the raising process, proving that not only in case of infeasibility all maximally raised dichotomies are invalid (as stated by Theorem 9.1.1), but also that in case of feasibility the same valid maximally raised encoding dichotomy is obtained, so that *raise_dichotomy* is sufficient to find all valid maximally raised dichotomies.

Theorem 9.1.2 *For any order of application of the next state constraints and of the raising rules to a given dichotomy, either the same valid maximally raised dichotomy is produced or no valid maximally raised dichotomy is produced.*

Proof: We show that if we start with i we get a maximally raised dichotomy that is unique if it is valid, i.e. the same maximally raised dichotomy is obtained independently of the order in which the next state constraints are used (the choice of the left or right rule is fixed once a next state constraint has been chosen). This shows that the procedure *raise_dichotomy* computes all valid maximally raised dichotomies. Since Theorem 9.1.1 shows that if $d(i)$ is invalid, any other maximally raised dichotomy that covers i is invalid, the theorem is proved.

Suppose that, given i, the next state constraints $e_1, e_2, \ldots, e_l$ are applicable. We will show that for any choice of e, after applying e to i, we get a raised dichotomy to which the remaining e's are still applicable with exactly the same rule, if the raised dichotomy is still valid [2]. Since the application of a next state constraint with a rule produces the same effect on the two sides of an dichotomy, if the conditions of applicability of a next-state constraint become true after applying a certain sequence of raising actions, these conditions will become true soon or later in any other sequence of raising actions. Therefore any order of raising ends up with the same valid maximally raised dichotomy.

Suppose that e_k and e_j are both applicable to $i = (l,r)$ and that e_k is applied first, producing i_{e_k}. We show that e_j is applicable to i_{e_k} with the same rule as it was to i, unless an

[2] It may happen that, after applying an applicable next state constraint, as a consequence some other applicable next state constraint does not need to be applied anymore.

invalid dichotomy is obtained. If e_j was applicable to i with the left rule, i.e. one child of each conjunction of e_j was in l, then e_j is still applicable to i_{e_k} with the left rule. If e_j was applicable to i with the right rule, i.e. one child of all but one conjunction c of e_j was in l and the parent of e_j was in r, then e_j is still applicable to i_{e_k} with the right rule, unless a previous raising has inserted in l one child of c previously unassigned; but in the last case e_j is applicable to i_{e_k} with the left rule and so it forces its parent into l, but its parent must have been already in r for e_j to be applicable with the right rule to i and so an invalid dichotomy is obtained. ■

9.1.2 Encoding of a Set of Encodeable GPI's

Once a set of GPI's is known to be encodeable, one must find codes of minimum length that satisfy the encoding constraints. If the requirement that codes are of minimum length is dropped, then it is sufficient to take the valid maximally raised dichotomies, make each of them complete by adding to the right block any state absent from the dichotomy and then choose a minimal set of complete maximally raised dichotomies that cover all free initial dichotomies [116]. Note that by adding absent states to the right block no invalid dichotomy can be produced, since no existing encoding constraints become applicable to the complete maximally raised dichotomies so obtained.

We will now discuss the case where codes of minimum length are wanted. An encoding column of a valid encoding corresponds to a complete and valid dichotomy. The next theorem proves that set of valid complete dichotomies is exactly the set of valid completions of the set of valid maximally raised dichotomies.

Theorem 9.1.3 *The set of valid complete dichotomies is the set of valid completions of valid maximally raised dichotomies.*

Proof: A free initial dichotomy generates two initial dichotomies. A dichotomy covers a free initial dichotomy iff it contains either initial dichotomy. From an initial dichotomy either one obtains a unique valid maximally raised dichotomy or no valid maximally raised dichotomy. We suppose that the given set of GPI's is encodeable, so for a given free initial dichotomy (x, y) at least one of the two initial dichotomies (l, r) yields a valid maximally raised dichotomy $(d(l), d(r))$. A valid complete dichotomy that covers a given free initial dichotomy (x, y) by block-wise containig (l, r) must contain the valid maximally raised dichotomy $(d(l), d(r))$, because adding symbols left and right to (l, r) does not invalidate any raising on (l, r) if a valid maximally raised dichotomy $(d(l), d(r))$ is obtained by maximally raising (l, r) (a raising by left rule is still applicable to supersets of l and

procedure exact_encode (*constraints*) {
    I = generate_initial_dichotomies (*constraints*)
    D = copy(I)
    foreach (dichotomy d in D)
        raise_dichotomy (d, *constraints*)
    D = remove_invalid_dichotomies (D, *constraints*)
    $\tilde{I}$ = duality_equivalence (I)
    foreach (free dichotomy $\tilde{i} = \{i_L, i_R\} \in \tilde{I}$)
        if ($\tilde{i}$ is not covered by $d(i_L) \in D$ or by $d(i_R) \in D$)
            return (*INFEASIBLE*)
    C = complete_dichotomy_generate (D)
    valid_complete = remove_invalid_dichotomies (C, *constraints*)
    mincov = minimum_cover (I, *valid_complete*)
    return (derive_codes (*mincov*))
}

Figure 9.4: Exact encoding of constraints

r; a raising by right rule is still applicable if a valid maximally raised dichotomy is obtained) and a valid complete dichotomy is a fortiori maximally raised.

By considering all possible completions of $(d(l), d(r))$ one gets all complete dichotomies that contain block-wise (l, r). By keeping only the valid completions one gets the set of valid and complete dichotomies that contain block-wise (l, r). ■

The selection of a minimum set of valid complete dichotomies that cover the original free initial dichotomies can then be cast again as a table covering problem. Attention must paid to the fact that a valid complete dichotomy covers a free initial dichotomy by covering any of its two initial dichotomies. In other words, each row of the covering table corresponds to a free initial dichotomy that has a 1 in a column corresponding to a valid complete that covers either initial dichotomy in that free initial dichotomy. Procedure *exact_encode* shows the full sequence of steps to check encodeability and, if the latter holds, to encode the set of GPI's with codes of minimum length.

9.1.3 Updating Sets and Raising Graphs

In this section we address the issue of adding more GPI's to a set of GPI's that is not encodeable. We know by Proposition 8.1.1 that there is an addition of GPI's that makes the current set encodeable, but the problem is which GPI's to add. Our strategy is to use the information gathered in checking encodeability to drive the choice of useful GPI's to add to the current cover. New notions of updating sets and raising graphs will be introduced to that purpose.

If no valid maximally raised dichotomy is produced, then, according to the order of raising, different invalid maximally raised dichotomies can be produced, as the following example shows. Consider the initial dichotomy $(bc; d)$ and the next state constraints $a = ab + ac$ and $d = da + dc$. Let L and R denote respectively the left and right raising rule.
Two different sequences of raising actions are:
$(bc; d) \overset{a=ab+ac(L)}{\longrightarrow} (abc; d)$
$(abc; d) \overset{d=da+dc(L)}{\longrightarrow} (abcd; d)$ invalid
$(abcd; d) \overset{a=ab+ac}{\longrightarrow} (abcd; d)$
$(abcd; d) \overset{d=da+dc}{\longrightarrow} (abcd; d)$

$(bc; d) \overset{d=da+dc(R)}{\longrightarrow} (bc; da)$
$(bc; da) \overset{a=ab+ac(L)}{\longrightarrow} (abc; da)$ invalid
$(abc; da) \overset{d=da+dc(L)}{\longrightarrow} (abcd; da)$ invalid
$(abcd; da) \overset{a=ab+ac}{\longrightarrow} (abcd; da)$
$(abcd; da) \overset{d=da+dc}{\longrightarrow} (abcd; da)$

At the first step both $a = ab + ac$ with L or $d = da + dc$ with R can be applied. If $a = ab + ac$ with L is applied first, then $d = da + dc$ with R cannot be applied anymore because its condition has been falsified. Instead $d = da + dc$ with L can be applied, but it must result in an invalid dichotomy because the parent of $d = da + dc$ was already in the right block and now is inserted in the left block. Applying first $d = da + dc$ with R has the advantage that both $a = ab + ac$ and $d = da + dc$ are recognized as responsible for removing $(bc; d)$, allowing more freedom to update the minterm constraints. For instance, update $a = ab + ac$ into $a = ab + ac + ad$, then $(bc; d)$ is raised to $(bc; ad)$ and it is not anymore invalid. Alternatively, update $d = da + dc$ to $d = da + dc + d$, then $(bc; d)$ is raised to $(abc; d)$ and it is not anymore invalid. If we would consider only $d = da + dc$ as responsible of deleting $(bc; d)$ we would miss that also by updating $a = ab + ac$ the cancellation does not take place anymore.

When a free initial dichotomy is violated (and therefore a face constraint cannot be satisfied), by Proposition 8.1.1 there is always a set of GPI's whose addition to the current selection makes the new set of GPI's encodeable. An optimization problem is to add the smallest number of GPI's that achieves the goal. The following result guarantees that, after adding a GPI, an existing set of free initial dichotomies is not less satisfied.

Proposition 9.1.1 *If a set of free initial dichotomies ID is satisfied by the next-state constraints of a set of GPI's G, then ID is satisfied by the next-state constraints induced by a set of GPI's $G' \supseteq G$.*

Proof: The consistency equations of minterms covered by the newly added GPI are updated. By the rule of removal, given a dichotomy and a consistency equation, if the left state of the equation is in the right block of the dichotomy and one state in each conjunct of the equation is in the left block then the dichotomy is deleted. When the equation is updated, one more conjunct is added to it and so the condition of the previous rule may fail to be true. Also it may be the case that a removal is a consequence of some previous raising. By adding a conjunct to a consistency equation it may happen that the conditions in the raising rules may not be anymore true, making impossible the raising and consequent removal. ■

Notice that the addition of a GPI may introduce more initial dichotomies (because of new face constraints), temporily making harder the overall encodeability problem.

We would like to add the smallest number of GPI's so that the resulting encoding constraints are satisfied. In the worst-case, a branch-and-bound search technique may have to explore all solutions, but a good choice of a new GPI at the branching step will bound more quickly the search. To this effect we annotate each unsatisfied free dichotomy with the next-state constraints that violated it. The following facts are important:

1. Next-state constraints of the form $a = a+\ldots$ are always trivially satisfied and once a constraint becomes such it does not need to be anymore considered (*trivial* next state constraint).

2. Next-state constraints with the *same* consistency equation, but associated to *different* minterms are *different* next state constraints. The reason is that, if they delete an initial dichotomy, all of them must be properly updated to avoid any violation of that initial dichotomy and this may require the addition of different GPI's.

3. Next-state constraints with *different* consistency equations may remove the *same* dichotomy, for *different* encoding violations. The procedure *remove_invalid_dichotomies* can be made to enumerate them all.

In order to choose an "effective" branching column, we need to *annotate* each unsatisfied initial dichotomy with the next-state constraints violating it. The annotation must capture exactly all sets of next-state constraints causing unsatisfiability. We highlight first some issues by means of the following examples.

1. A maximally raised dichotomy may be removed as a consequence of a previous raising action. Both the raising next-state constraint and deleting next-state constraint are therefore responsible of the cancellation. Selecting a GPI that updates either of them could make the cancellation go away. Consider the dichotomy $(ab; c)$ and the next state constraints $d = da + db$, $f = fa + fd$, $c = ca + cf$:
   $(ab; c) \xrightarrow{d=da+db(L)} (abd; c)$
   $(abd; c) \xrightarrow{f=fa+fd(L)} (abdf; c)$
   $(abdf; c) \xrightarrow{c=ca+cf(L)} (abcdf; c)$ invalid
   $(abcdf; c) \xrightarrow{d=da+db} (abcdf; c)$
   $(abcdf; c) \xrightarrow{f=fa+fd} (abcdf; c)$
   Updating any of $d = da + db$, $f = fa + fd$, $c = ca + cf$ can make the cancellation go away. For instance, update $d = da + db$ to $d = da + db + dc$:
   $(ab; c) \xrightarrow{d=da+db+dc(L)} (ab; c)$
   $(ab; c) \xrightarrow{f=fa+fd} (ab; c)$
   $(ab; c) \xrightarrow{c=ca+cf(R)} (ab; cf)$
   $(ab; cf) \xrightarrow{d=da+db+dc} (ab; cf)$
   $(ab; cf) \xrightarrow{f=fa+fd(R)} (ab; cdf)$
   $(ab; cdf) \xrightarrow{c=ca+cf} (ab; cdf)$
   $(ab; cdf) \xrightarrow{d=da+db+dc} (ab; cdf)$
   Update $f = fa + fd$ to $f = fa + fd + fc$:
   $(ab; c) \xrightarrow{d=da+db(L)} (abd; c)$
   $(abd; c) \xrightarrow{f=fa+fd+fc} (abd; c)$
   $(abd; c) \xrightarrow{c=ca+cf(R)} (abd; cf)$
   $(abd; cf) \xrightarrow{d=da+db} (abd; cf)$
   $(abd; cf) \xrightarrow{f=fa+fd+fc} (abd; cf)$
   $(abd; cf) \xrightarrow{c=ca+cf} (abd; cf)$
   Update $c = ca + cf$ to $c = ca + cf + c$:
   $(ab; c) \xrightarrow{d=da+db(L)} (abd; c)$

$(abd; c) \overset{f=fa+fd(L)}{\longrightarrow} (abdf; c)$
$(abdf; c) \overset{c=ca+cf+c}{\longrightarrow} (abdf; c)$
$(abdf; c) \overset{d=da+db}{\longrightarrow} (abdf; c)$
$(abdf; c) \overset{f=fa+fd}{\longrightarrow} (abdf; c)$

2. The proposed annotation is not order-independent, because invalid maximally raised dichotomies and next state constraints which remove them are order-dependent. Consider the dichotomy $(abe; c)$ and the next state constraints $d = da + db$, $c = cd + ce$, in the given order:
$(abe; c) \overset{d=da+db(L)}{\longrightarrow} (abde; c)$
$(abde; c) \overset{c=cd+ce(L)}{\longrightarrow} (abcde; c)$ invalid
$(abcde; c) \overset{d=da+db}{\longrightarrow} (abcde; c)$
Update $c = cd + ce$ to $c = cd + ce + c$:
$(abe; c) \overset{d=da+db(L)}{\longrightarrow} (abde; c)$
$(abde; c) \overset{c=cd+ce+c}{\longrightarrow} (abde; c)$
Let us now exchange the order of the two next state constraints:
$(abe; c) \overset{c=cd+ce(R)}{\longrightarrow} (abe; cd)$
$(abe; cd) \overset{d=da+db(L)}{\longrightarrow} (abde; cd)$ invalid
$(abde; cd) \overset{c=cd+ce(L)}{\longrightarrow} (abcde; cd)$ invalid
$(abcde; cd) \overset{d=da+db}{\longrightarrow} (abcde; cd)$
Update $c = cd + ce$ to $c = cd + ce + c$:
$(abe; c) \overset{c=cd+ce+c}{\longrightarrow} (abe; c)$
$(abe; c) \overset{d=da+db(L)}{\longrightarrow} (abde; c)$
$(abde; c) \overset{c=cd+ce+c}{\longrightarrow} (abde; c)$
Update $d = da + db$ to $d = da + db + dc$:
$(abe; c) \overset{c=cd+ce(R)}{\longrightarrow} (abe; cd)$
$(abe; cd) \overset{d=da+db+dc}{\longrightarrow} (abe; cd)$

In all previous examples, it was always sufficient to update a single next state constraint to make satisfiable the given initial dichotomy. It is not always so, as the following example shows.

1. Consider the dichotomy $(ab; c)$ and the next state constraints $d = da + db$, $f = fa + fd$, $c = ca + cf$, $c = ca + cd$. Update $d = da + db$ to $d = da + db + dc$:
$(ab; c) \overset{d=da+db+dc}{\longrightarrow} (ab; c)$

$(ab; c) \stackrel{f=fa+fd}{\longrightarrow} (ab; c)$
$(ab; c) \stackrel{c=ca+cf(R)}{\longrightarrow} (ab; cf)$
$(ab; cf) \stackrel{c=ca+cd(R)}{\longrightarrow} (ab; cdf)$
$(ab; cdf) \stackrel{d=da+db+dc}{\longrightarrow} (ab; cdf)$
$(ab; cdf) \stackrel{f=fa+fd}{\longrightarrow} (ab; cdf)$
$(ab; cdf) \stackrel{c=ca+cf}{\longrightarrow} (ab; cdf)$
Update $f = fa + fd$ to $f = fa + fd + fc$:
$(ab; c) \stackrel{d=da+db(L)}{\longrightarrow} (abd; c)$
$(abd; c) \stackrel{f=fa+fd+fc}{\longrightarrow} (abd; c)$
$(abd; c) \stackrel{c=ca+cf(R)}{\longrightarrow} (abd; cf)$
$(abd; cf) \stackrel{c=ca+cd(L)}{\longrightarrow} (abcd; cf)$ invalid
$(abcd; cf) \stackrel{d=da+db}{\longrightarrow} (abcd; cf)$
$(abcd; cf) \stackrel{f=fa+fd+fc}{\longrightarrow} (abcd; cf)$
$(abcd; cf) \stackrel{c=ca+cf}{\longrightarrow} (abcd; cf)$
Update $c = ca + cf$ to $c = ca + cf + c$:
$(ab; c) \stackrel{d=da+db(L)}{\longrightarrow} (abd; c)$
$(abd; c) \stackrel{f=fa+fd(L)}{\longrightarrow} (abdf; c)$
$(abdf; c) \stackrel{c=ca+cf+c}{\longrightarrow} (abdf; c)$
$(abdf; c) \stackrel{c=ca+cd(L)}{\longrightarrow} (abcdf; c)$ invalid
$(abcdf; c) \stackrel{d=da+db}{\longrightarrow} (abcdf; c)$
$(abcdf; c) \stackrel{f=fa+fd}{\longrightarrow} (abcdf; c)$
$(abcdf; c) \stackrel{c=ca+cf+c}{\longrightarrow} (abcdf; c)$

The conclusion is that to make the cancellation go away one must update either $d = da + db$ or ($f = fa + fd$ and $c = ca + cd$) or ($c = ca + cf$ and $c = ca + cd$), i.e. the minimal sets of next state constraints that must be updated have cardinality ≥ 1.

The last examples motivate the following definitions. A next state constraint is **updated** when a disjunct that has the parent among its conjuncts is added to it. When the added disjunct contains only the parent, the updated next-state constraint is **trivial**. A trivial next-state constraint can always be reduced in the form $parent = parent$. So a next-state constraint can be made trivial by adding a disjunct that is a singleton coinciding with the parent.

Given an initial dichotomy i and a set of next-state constraints C, a **set of updating next-state constraints** or **updating set** $U \subseteq C$ is a set of next-state constraints such that, if they

are replaced by trivial next-state constraints U', then i is not anymore violated by $(C - U) \cup U'$. If i is not violated by any $c \in C$, then $U = \emptyset$. If i is not satisfied by C, a trivial updating set is C itself. A **minimal updating set** is an updating set that does not contain properly a set of updating next-state constraints.

The **support** of the set of all minimal updating sets is the union of all minimal updating sets. The support can be used in the computation of a correct lower bound. As an example, suppose that the set of all minimal updating sets is

$$\{\{d = da + db\}, \{f = fa + fd, c = ca + cd\}, \{c = ca + cf, c = ca + cd\}\}, \tag{9.1}$$

then its support is

$$\{d = da + db, f = fa + fd, c = ca + cd\}. \tag{9.2}$$

We need algorithms to find:

1. an updating set;
2. a minimal updating set;
3. all minimal updating sets;
4. the support of all minimal updating sets.

We present next an elegant characterization of updating sets in terms of the raising graph, that is a graph describing all possible raisings that can be acted upon an initial dichotomy. This characterization is the basis of algorithms discussed in Section 11.3.

We state first some facts about applicability of next-state constraints to dichotomies. Given an initial dichotomy, a next-state constraint is **applicable** to it iff the conditions of either raising rule are satisfied by the dichotomy and the application of the next-state constraint adds at least one state to either block. If the former condition is true and the latter is false the constraint is **vacuously applicable**. Since the conditions for the two raising rules are mutually exclusive, at most one of them is applicable. Therefore we can say that a next-state constraint is **left applicable** or **right applicable** to a dichotomy. If a next-state constraint is applicable to a dichotomy it stays applicable to it until it is applied or until it becomes vacuously applicable (because another raising action produces the same effect), with at most a change of type of rule. Precisely, a left applicable next-state constraint stays left applicable or becomes vacuously applicable. A right applicable next-state constraint either stays right applicable or becomes vacuously applicable or becomes left

applicable, because a left raising adds to the left a state of a conjunct that before had no state to the left. In the latter case, invalidity is reached. Given a dichotomy i and a set of next-state constraints C, the latter can be partitioned into a set C_a of the ones applicable, a set C_{na} of the ones not applicable or vacuously applicable and a set C_u of the ones already applied (used). The three sets are a partition of C.

Given an initial dichotomy and a set of next-state constraints, suppose that the ones in C_a are applied in parallel to an initial dichotomy so that raised dichotomies are obtained. For each raised dichotomy, move the applied next-state constraint (which now become vacuously applicable) from C_a to C_u and check if any next-state constraint in C_{na} is now applicable, in which case it must migrate from C_{na} to C_a. At each step the sets C_a, C_{na}, C_u are a partition of C. By this process one builds a **raising graph** whose nodes are dichotomies, and whose directed edges are next-state constraints that raise the predecessor dichotomy to the successor dichotomies. After a new node (raised dichotomy) is added, one checks whether it is invalid; if so, one does not raise that node anymore. When no node can be raised, the process is terminated. The resulting graph is the collection of all possible ways to apply the next-state constraints in C to the initial dichotomy i.

Theorem 9.1.4 *The set of outgoing edges of any node (that is not a sink) of the raising graph of a violated initial dichotomy is an updating set.*

Proof: By Theorem 9.1.2, either all sinks of the graph are the same valid maximally raised dichotomy or they are invalid raised dichotomies (not necessarily the same). In the latter case, consider the outgoing edges E_n of a node n that is not a sink. If each of the next-state constraints associated to E_n is updated, say to a trivial next state constraint, the node n becomes a valid maximally raised dichotomy; it is maximally raised because C_a has been emptied, and it is valid because it has been valid up to now and no raising has been performed. But consider now the raising graph that would be obtained by starting all over the process, without using the next-state constraints in E_n. Since along a path a valid maximally raised dichotomy is reached, then all sinks must be the same valid maximally raised dichotomy, again by Theorem 9.1.2. In other words, the outgoing edges of n yield an updating set. Therefore any path in the original raising graph from the source to an invalid sink must include at least one of these edges, so that by updating the related next-state constraints an invalid raised dichotomy is not reached. ■

Corollary 9.1.1 *A minimal set of outgoing edges, i.e., not properly contained in any other set of outgoing edges, is a minimal updating set. All minimal sets of outgoing edges are all minimal*

updating sets. The union of all minimal sets of outgoing edges gives the support of all minimal updating sets.

9.1.4 Choice of a Branching Column

Given a set of selected GPI's and of unsatisfied free initial dichotomies, we add one more GPI to minimize the violations that make unsatisfied those free initial dichotomies.

An example will help in clarifying the notion. Let $\tilde{i_1} = \{i_{1_L}, i_{1_R}\}$ and $\tilde{i_2} = \{i_{2_L}, i_{2_R}\}$ be unsatisfied free initial dichotomies. Suppose that for each of them we know the minimal updating sets. For instance, suppose that the disjunction of minimal updating sets of i_{1_L} is $c_i + c_j + c_s$, of i_{1_R} is $c_i + c_p$, of i_{2_L} is $c_j c_q + c_i$ and of i_{2_R} is $c_p + c_r$, where the c's are next state constraints. From the updating next-state constraints we know the minterms that must be updated. The step from next-state constraints to minterms is clarified by the following statements:

1. the same next-state constraint may be associated to more than one minterm;
2. to update a next-state constraint a new conjunct must be or-ed to it;
3. to or a new conjunct a new GPI must be chosen that provides it by its next state tag;
4. a GPI contributes a conjunct only to the minterms that it covers, i.e. a GPI updates a next-state constraint only for the minterms that it covers;
5. if the same next state constraint comes with more than one minterm it may be necessary to update it differently for each minterm, i.e. a different GPI may be have to be selected to update that next-state constraint for each minterm to which it is associated.

For instance, suppose that c_i is associated to minterm m_i, c_j to minterms m_j and m_k, c_s to m_s and m_t, c_p to m_p, c_q to m_q and c_r to m_r (indexes of constraints and minterms vary in different sets). Then the set of all minterms to be updated of i_{1_L} is $m_i + m_j m_k + m_s m_t$, of i_{1_R} is $m_i + m_p$, of i_{2_L} is $m_j m_k m_q + m_i$ and of i_{2_R} is $m_p + m_r$. We can summarize the updating conditions of $\tilde{i_1}$ as:

$$(m_i + m_j m_k + m_s m_t) + (m_i + m_p) \tag{9.3}$$

and of $\tilde{i_2}$ as:

$$(m_j m_k m_q + m_i) + (m_p + m_r). \tag{9.4}$$

For $\tilde{i_1}$ to be satisfied it is necessary to find a GPI such that

1. its proper input and present state part covers the proper input and present state part of m_i and no state in its tag is in the left block of i_{1_L} (otherwise, one does not invalidate the if condition of *raise_dichotomy* and *remove_invalid_dichotomies*); or,

2. its proper input and present state part covers the proper input and present state part of m_j and m_k and no state in its tag is in the left block of i_{1_L}; or,

3. its proper input and present state part covers the proper input and present state part of m_s and m_t and no state in its tag is in the left block of i_{1_L}; or,

4. its proper input and present state part covers the proper input and present state part of m_i and no state in its tag is in the left block of i_{1_R}.

5. its proper input and present state part covers the proper input and present state part of m_p and no state in its tag is in the left block of i_{1_R}.

There may be no single GPI that achieves the goal, but a set of them may be needed. So we want to select the GPI that improves the overall satisfiability of unsatisfied initial dichotomies, even if it does not succeed in making satisfiable any single of them. Transform in POS the updating conditions of $\tilde{i_1}$:

$$(m_i + m_j + m_s + m_p)(m_i + m_j + m_t + m_p)(m_i + m_k + m_s + m_p)(m_i + m_k + m_t + m_p) \quad (9.5)$$

and do the same for those of $\tilde{i_2}$ [3]. In this way, the updating conditions of $\tilde{i_1}$ and $\tilde{i_2}$ can be expressed by a set of updating clauses.

In general, consider a set of clauses of the form $\tilde{i}(m_i + \ldots + m_p)$, for each unsatisfied free initial dichotomy $\tilde{i}$, and each updating clause $(m_i + \ldots + m_p)$ obtained for $\tilde{i}$. These clauses can be seen as the rows of a unate covering table, whose columns are the candidate GPI's to extend the current solution. There is an element in the table at the intersection of GPI g_k and row $\tilde{i}(m_i + \ldots + m_p)$ iff

1. the proper input and present state part of g_k covers the proper input and present state part of m_i and no state in the tag of g_k is in the left block of i_L; or,

2. . . .; or,

[3] Boolean identities allow simplification of the clauses, so that subsumed literals and clauses can be cancelled. For instance the first clause simplifies from $m_i + m_j + m_s + m_i + m_p$ to $m_i + m_j + m_s + m_p$. Apparently this alters the choice of the branching column.

3. the proper input and present state part of g_k covers the proper input and present state part of m_p and no state in the tag of g_k is in the left block of i_R.

Such a table (called the *full satisfiability table*) requires a knowledge of all the updating sets and it would be difficult to manipulate with implicit techniques, because each clause refers to a variable number of conditions. The difficulty is not with having many clauses for the same $\tilde{i}$, but with having many literals per clause. Each clause is a row of the table, but we do not know an appropriate labelling scheme for a row with a variable number of literals.

A cruder estimate is made by restriction to one minimal updating set for each initial dichotomy. In that case, each updating clause will have exactly two literals and an implicit labelling scheme for rows and columns can be devised. For instance, consider $m_j m_k$ for i_{1_L} and m_p for i_{1_R}, that give the POS

$$m_j m_k + m_p = (m_j + m_p)(m_k + m_p). \tag{9.6}$$

There is an element in the table at the intersection of GPI g_k and row $\tilde{i_1}(m_j + m_p)$ iff

1. the proper input and present state part of g_k covers the proper input and present state part of m_j and no state in the tag of g_k is in the left block of i_{1_L}; or,

2. the proper input and present state part of g_k covers the proper input and present state part of m_p and no state in the tag of g_k is in the left block of i_{1_R}.

Such a table is called *partial satisfiability table.*

This restriction affects only the quality of branching column selection, not the exactness of a final solution, that is guaranteed by the completness of the search technique. The GPI that has more entries in the table is considered to be the most desirable to choose as next branching column. This proposed algorithm requires to build a matrix and to find the column in it with maximum number of ones.

9.1.5 Computation of a Lower Bound

In ordinary unate covering, the cardinality of a maximum set of pairwise disjoint rows (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. Since finding a maximum independent set is an NP-complete problem, in practice an heuristic is used that provides a weaker lower bound. A row is found that is disjoint from a

maximum number of rows (i.e. a row of minimum length). All rows having elements in common with it are then deleted. This process is iterated until a set of pairwise disjoint rows (independent set) is found.

Consider again the example used to describe the branching column selection, where the unsatisfied free initial dichotomies $\tilde{i_1}$ and $\tilde{i_2}$ had respectively the updating conditions:

$$(m_i + m_j m_k + m_s m_t) + (m_i + m_p) \tag{9.7}$$

and:

$$(m_j m_k m_q + m_i) + (m_p + m_r). \tag{9.8}$$

Suppose that we build the full satisfiability table, as described above. The cardinality of a maximum set of pairwise disjoint rows is a lower bound on the cardinality of the solution to the constrained covering problem, because a different element must be selected for each of the independent rows in order to satisfy them. This captures the notion of a maximum set of pairwise disjoint violations of free initial dichotomies.

We already observed that building the full satisfiability table may be difficult and not prone to a simple implicit manipulation scheme. Unfortunately here we cannot use the partial satisfiability table either, because it does not yield a correct lower bound, since we would be choosing an arbitrary updating set for each initial dichotomy and we cannot claim that this is the best that can be done.

A way out of this difficulty is to build the *support satisfiability table*. Replace the *and* operators with *or* operators in the previous updating conditions to get "relaxed" updating conditions:

$$(m_i + m_j + m_k + m_s + m_t) + (m_i + m_p) \tag{9.9}$$

and:

$$(m_j + m_k + m_q + m_i) + (m_p + m_r). \tag{9.10}$$

If the original updating conditions are satisfied, the relaxed ones are too. Again, these clauses can be seen as the rows of a unate covering table. There is an element in the table at the intersection of GPI g_k and row $\tilde{i_1}$, which is associated to $(m_i + m_j + m_k + m_s + m_t) + (m_i + m_p)$, iff:

1. the proper input and present state part of g_k covers the proper input and present state part of any of $\{m_i, \ldots, m_t\}$ and no state in the tag of g_k is in the left block of i_{1_L}; or,

2. the proper input and present state part of g_k covers the proper input and present state part of any of $\{m_i, m_p\}$ and no state in the tag of g_k is in the left block of i_{1_R}.

Such a table is called *support satisfiability table*.

A maximal set of parwise disjoint rows of this table still provides a correct lower bound, albeit a lower one than the full satisfiability table does. One can manipulate this table with implicit techiques, as shown precisely in Section 11.3; the set of rows are the $\tilde{i}$'s. To compute the entries of the table one can use a relation on dichotomies and minterms, such that for instance $\tilde{i_1} m_i$ is in the relation iff m_i is a literal in the clause associated to $\tilde{i_1}$. This works because there is a unique clause associated to each $\tilde{i}$.

This table can be used also for branching column selection, but it would degrade the quality of the choice even more than the *partial satisfiability table*.

A weaker lower bound can be computed considering only the next state constraints of the type $a = \emptyset$. Since for each of them a GPI must be chosen to cover the related minterm, one can use a covering table with entries to 1 iff a GPI contains a minterm, as in ordinary unate covering.

Chapter 10

Binate Covering

10.1 Introduction

It is not feasible to generate GPI's and to set up a related unate or binate covering table by explicit techniques on non-trivial examples [19]. By means of techniques as in [79, 53, 30], GPI's can be generated using BDD-based (alias *implicit*) representations. The next step is to select an encodeable cover of GPI's using implicit representations. This motivates the development of new algorithms to solve covering problems based on the representation and manipulation of covering tables represented with BDD's. Since covering problems are ubiquitous in logic synthesis and combinatorial optimization, in this chapter we will develop a general theory of implicit solutions of binate covering problems. It is a development of large applicability, as shown by its successful application to an host of problems in state minimization [63]. In the next chapter we will see how this formulation is employed in the GPI minimization problem.

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 minimum number of encoding columns that satisfy a set of encoding constraints, 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 [46]. 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, where each column of the table is a prime compatible and each row is one of the covering or closure clauses of the problem [50]. 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.

A special case of binate covering problem is a unate covering problem, where no literal in the negative phase is present. Exact two-level minimization [87, 113] can be cast as a unate covering problem. The columns are the prime implicants, the rows are the minterms and there is a 1 entry in the matrix when a prime contains a minterm.

Various techniques have been proposed to solve binate covering problems. A class of them [14, 72] 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 [82] 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 [61]. 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 [117]. 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. A BDD [16, 10] 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 [23] and UC Berkeley [142] produced powerful techniques for implicit enumeration of subsets of states of a Finite State Machine (FSM). Later work at Bull [25, 79] 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 [11] 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 [29, 53]. 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.

This chapter describes an implicit formulation of the binate covering problem and presents an implementation. The implicit binate solver has been tested for the selection of an encodable set of

GPI's, as reported in Chapter 11, and for state minimization of ISFSM's and pseudo NDFSM's [63]. 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 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 10.2, we will compare this problem with 0-1 integer linear programming. The branch-and-bound scheme will be introduced in Section 10.3 which has be used in explicit binate covering algorithms summarized in Section 10.4. In Section 10.5, we survey the classical reduction rules used in explicit algorithms. Our implicit binate covering algorithm is then introduced in Section 10.6 and its program input, an implicit table representation, is described in Section 10.7. Section 10.8 illustrates how reduction techniques can be implicitized. Other kinds of implicit table manipulations are introduced in Section 10.9.

10.2 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.

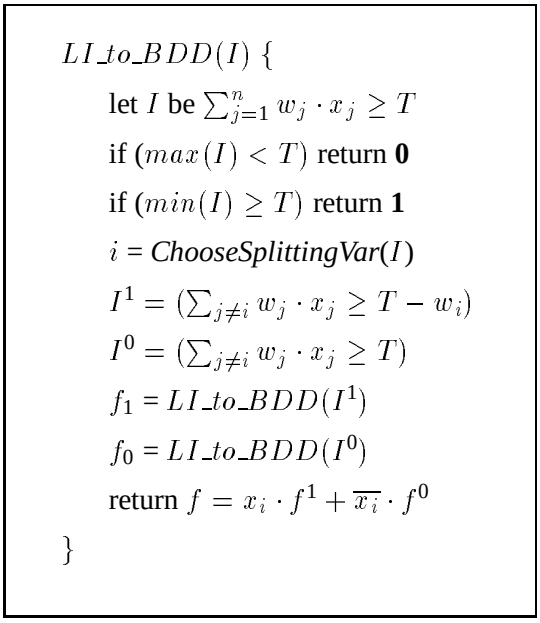


Figure 10.1: Transformation from linear inequality to Boolean expression.

As an example of reduction from ILP to BCP, a procedure (taken from [61]) that derives the Boolean expression corresponding to $\sum_{j=1}^{n} w_j.x_j \geq T$ is shown in Figure 10.1.
The idea of the recursion relies on the observation that:

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.

10.3 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 10.2 shows the classical algorithm [105]. 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.

10.4 A Branch-and-Bound Algorithm for Minimum Cost Binate Covering

We will survey in this section a branch-and-bound solution of minimum cost binate covering. This technique has been described in [51, 50, 13, 14], and implemented in successful computer programs [112, 108, 130]. 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.

```
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 10.2: Structure of branch-and-bound.

10.4.1 The Binary Recursion Procedure

The inputs to the algorithm are:

- 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 10.5. Another more complex reduction criterion (Gimpel's rule) can also be applied (see Subsection 10.5.12). 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 10.4.3). 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 10.4.2) and each of them is solved recursively. When everything fails, binary recursion is performed by choosing a branch column (see Subsection 10.4.4). 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:

- 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

```
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, select1, 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 10.3: Detailed branch-and-bound algorithm.

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.

10.4.2 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 10.4.

Theorem 10.4.1 *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 [108, 130] 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

```
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 10.4: N-way partitioning.

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*.

Theorem 10.4.2 *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. ■

10.4.3 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 [112, 108, 130], 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.

10.4.4 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 [112, 108, 130], 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).

10.5 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 10.5.7). A dual statement holds for unacceptable columns (Section 10.5.8). Also related is the case of unnecessary columns (Section 10.5.9).

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 10.5.5).

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 10.5.1).

Even though more complex criteria of dominance have been investigated (for instance, Section 10.5.12), the previous ones are basic in any table covering solver. Reduction rules have previously been stated for the binate covering case [50, 51, 14, 13], and also for the unate covering case [87, 113, 13]. 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 [50, 51, 14, 13], 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 10.5 shows how reductions are applied in [112, 108, 130][1].

10.5.1 Row Dominance

Definition 10.5.1 *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* $M_{j,k} = 0$,

[1] The reductions *β-dominance* and *row_consensus* are only in [108] and the reduction by implication is only in [130].

```
sm_reduce(A, solution, 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 solution
            if (cost of solution ≥ 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 solution
}
```

Figure 10.5: Flow of reduction rules.

- $M_{i,k} = 2$.

Theorem 10.5.1 *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

- similar to column dominance (Rule 3) in [50], 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 [51], 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 [14],
- identical to row dominance (Definition 2.11) in [13].

Row Dominance for a Unate Table

Definition 10.5.2 *A row R_i dominates another row R_j if for all columns C_k, $M_{i,k} = 1 \Rightarrow M_{j,k} = 1$.*

10.5.2 Row Consensus

Theorem 10.5.2 *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 [108]. This criterion generalizes the one given in [59].

10.5.3 Column α-Dominance

Definition 10.5.3 *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 10.5.3 *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 [50], 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 [51]: (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 [14],
- identical to column dominance (Definition 2.12, Theorem 2.4.1) in [13].

Column Dominance for a Unate Table

Definition 10.5.4 *A column C_i dominates another column C_j if for all rows R_k, $M_{k,j} = 1 \Rightarrow M_{k,i} = 1$.*

10.5.4 Column β-Dominance

Definition 10.5.5 *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 10.5.4 *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 10.5.3,
- more general than row β-dominance (Rule 5) in [50], because the latter assumes that the covering table contains only rows with no or one 0,
- equivalent to second half of Rule 4 in [51]: (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 [14] and [13].

10.5.5 Column Dominance

Definition 10.5.6 *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 10.5.5 *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.*

10.5.6 Column Mutual Dominance

Definition 10.5.7 *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 10.5.6 *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 [128],
- not mentioned in other papers.

10.5.7 Essential Column

Definition 10.5.8 *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 10.5.7 *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 [50],
- identical to Rule 1 in [51],
- included in Definition 9 in [14]: the row R_i in the above definition corresponds to a singleton-1 essential row in [14],
- included in Definition 2.10 in [13]: the row R_i in the above definition corresponds to a singleton-1 essential row in [13].

Essential Column for a Unate Table

Definition 10.5.9 *A column is an essential column if it contains the 1 of a singleton row.*

10.5.8 Unacceptable Column

Definition 10.5.10 *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 10.5.8 *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 [50],
- identical to Rule 2 in [51],
- included in Definition 9 in [14]: the row R_i in the above definition corresponds to a singleton-0 essential row in [14],
- included in Definition 2.10 in [13]: the row R_i in the above definition corresponds to a singleton-0 essential row in [13].

10.5.9 Unnecessary Column

Definition 10.5.11 *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 10.5.9 *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 [50],
- identical to Rule 5 in [51],
- not mentioned in [14] and [13].

10.5.10 Trial Rule

Theorem 10.5.10 *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.*

This reduction rule is

- identical to Rule 6 in [50],
- not mentioned in other papers.

10.5.11 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 10.5.11 *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 [50] and [51],
- briefly mentioned in [14],
- identical to the unfeasible problem in [13].

10.5.12 Gimpel's Reduction Step

Another heuristic for solving the minimum cover problem has been suggested by Gimpel [48]. 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.

[2] It is possible that a row is left with only 2's by a sequence of reduction steps.

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 10.5.12 *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 [113], 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 [60] 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 [48] and then implemented in [112]. In [108, 130] Gimpel's rule has been extended to handle the binate case. This extension has been described in [131].

10.6 Implicit Binate Covering

```
mincov(R, C, U) {
    (R, C) = Reduce(R, C, U)
    if (Terminal_Case(R, C))
        if (cost(R, C) ≥ U)  return no solution
        else  U = cost(R, C); return solution
    L = Lower_Bound(R, C)
    if (L ≥ U)  return no 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 10.6: Implicit branch-and-bound algorithm.

The classical branch-and-bound algorithm [50, 51] 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 solvers. For example, we would not expect these binate solvers to handle examples requiring over 10^6 columns (up to 2^{1500} columns), reported in state minimization of FSM's [63]. 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 $(r, c, R, C, \mathtt{0}, \mathtt{1})$, 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 10.3. In our implicit formulation, we keep the branch-and-bound scheme summarized in Figure 10.6, 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.

10.7 Implicit Table Generation

Here we define three ways of specifying the binate covering table in decreasing order of generality. A table is defined implicitly by generating BDD-based representations of the rows and columns and by giving relations specifying the 1 and 0 entries, given the rows and columns. By imposing restrictions on the way in which rows and columns are labeled and entries are defined, one gets representations with varying degrees of generality. Historically the third (less general) way was implemented first to solve exact state minimization of ISFSM's [65]. It is applicable to other problems whose covering table can be represented in the same way, e.g., the exact formulation of technology mapping for area minimization [113]. The difference between the first and second formulation is only in some computation simplification in the latter one, for tables that have at most one 0 per row. There is a trade-off between generality of the representation and efficiency of the computations: "hard-wiring" the rules that define a table may speed up table manipulations, to the price of more limited applicability.

In Chapter 11 we will see how the covering tables occurring in GPI minimization are generated. In [63] it is shown how covering tables occurring in state minimization of FSM's are constructed. In the next section, we will describe how a binate covering table can be manipulated implicitly so as to solve the minimum cost binate covering problem.

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: $\mathtt{0}(r, c)$
   - the 1-entries relation at the intersection of row r and column c: $\mathtt{1}(r, c)$
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: $\mathtt{0}(r, c)$
- the 1-entries relation at the intersection of row r and column c: $\mathtt{1}(r, c)$

3. Specialized binate covering table for exact state minimization and similar problems:

- 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: $\mathtt{0}((c, d), p) = (p = c)$
- the 1-entries relation at the intersection of row (c, d) and column p: $\mathtt{1}((c, d), p) = (p \supseteq d)$

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.

10.8 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 10.7.

```
Reduce(R, C, U) {
    repeat {
        Collapse_Columns(C)
        Column_Dominance(R, C)
        Sol = Sol ∪ Essential_Columns(R, C)
        if (|Sol| ≥ U)  return no solution
        Unacceptable_Columns(R, C)
        Unnecessary_Columns(R, C)
        if (C does not cover R)  return no solution
        Collapse_Rows(R)
        Row_Dominance(R, C)
    } until (both R and C unchanged)
    return (R, C)
}
```

Figure 10.7: Implicit reduction loop.

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.

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.

10.8.1 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 10.8.1 *Two columns are duplicates, if on every row, their corresponding table entries are identical.*

Theorem 10.8.1 Duplicated columns *can be computed as:*

$$dup\_col(c',c) \stackrel{1}{=} \forall r\,\{R(r) \Rightarrow [(\mathtt{0}(r,c') \Leftrightarrow \mathtt{0}(r,c)) \cdot (\mathtt{1}(r,c') \Leftrightarrow \mathtt{1}(r,c))]\}$$

$$dup\_col(c',c) \stackrel{2}{=} \forall r\,\{R(r) \Rightarrow [\neg\mathtt{0}(r,c') \cdot \neg\mathtt{0}(r,c) \cdot (\mathtt{1}(r,c') \Leftrightarrow \mathtt{1}(r,c))]\}$$

$$dup\_col(p',p) \stackrel{3}{=} \nexists d\ R(p',d) \cdot \nexists d\ R(p,d) \cdot \forall d\,\{[\exists c\ R(c,d)] \Rightarrow [(p' \supseteq d) \Leftrightarrow (p \supseteq d)]\}$$

Proof: As discussed at the end of Section 10.7, 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., $\mathtt{0}(r,c') \Leftrightarrow \mathtt{0}(r,c)$), and (2) the entry (r,c) is a 1 if and only if the entry (r,c') is a 1, (i.e., $\mathtt{1}(r,c') \Leftrightarrow \mathtt{1}(r,c)$). 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., $\neg\mathtt{0}(r,c') \cdot \neg\mathtt{0}(r,c)$). ■

Theorem 10.8.2 *Duplicated columns can be collapsed by:*

$$C(c) \stackrel{1,2}{=} C(c) \cdot \nexists c'\,[C(c') \cdot (c' \prec c) \cdot dup\_col(c',c)]$$

$$C(p) \stackrel{3}{=} C(p) \cdot \nexists p'\,[C(p') \cdot (p' \prec p) \cdot dup\_col(p',p)]$$

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] ■

10.8.2 Duplicated Rows

Definition 10.8.2 *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 10.8.3 Duplicated rows *can be computed as:*

$$\begin{aligned} dup\_row(r', r) \quad &^{1,2} = \quad \forall c \, \{C(c) \Rightarrow [(\mathtt{0}(r', c) \Leftrightarrow \mathtt{0}(r, c)) \cdot (\mathtt{1}(r', c) \Leftrightarrow \mathtt{1}(r, c))]\} \\ dup\_row(c', d', c, d) \quad &^{3} = \quad (c' = c) \cdot \nexists p \, [C(p) \cdot ((p \supseteq d') \not\Leftrightarrow (p \supseteq d))] \end{aligned}$$

Proof: Similar to the proof for Theorem 10.8.1. 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., $\mathtt{0}(r', c) \Leftrightarrow \mathtt{0}(r, c)$), and (2) the entry (r, c) is a 1 if and only if the entry (r', c) is a 1, (i.e., $\mathtt{1}(r', c) \Leftrightarrow \mathtt{1}(r, c)$). ■

Theorem 10.8.4 *Duplicated rows can be collapsed by:*

$$\begin{aligned} R(r) \quad &^{1,2} = \quad R(r) \cdot \nexists r' \, [R(r') \cdot (r' \prec r) \cdot dup\_row(r', r)] \\ R(c, d) \quad &^{3} = \quad R(c, d) \cdot \nexists c', d' \, [R(c', d') \cdot (d' \prec d) \cdot dup\_row(c', d', c, d)] \end{aligned}$$

Proof: The proof is similar to that for Theorem 10.8.2, 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 [80] to pick a representative column out of each set of duplicated columns.

10.8.3 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 10.8.3 *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 10.8.5 *The α-dominance relation can be computed as:*

$$\begin{aligned}
\alpha_dom(c',c) \quad ^{1} &= \quad \nexists r\ \{R(r)\cdot[\mathtt{1}(r,c)\cdot\neg\mathtt{1}(r,c')]+[\mathtt{0}(r,c')\cdot\neg\mathtt{0}(r,c)]\}\\
\alpha_dom(c',c) \quad ^{2} &= \quad \nexists r\ \{R(r)\cdot[\mathtt{1}(r,c)\cdot\neg\mathtt{1}(r,c')+\mathtt{0}(r,c')]\}\\
\alpha_dom(p',p) \quad ^{3} &= \quad \nexists c,d\ [R(c,d)\cdot(p\supseteq d)\cdot(p'\not\supseteq d)]\cdot\ \nexists d\ R(p',d)
\end{aligned}$$

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 10.8.4 *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 10.8.6 *The β-dominance relation can be computed by:*

$$\begin{aligned}
\beta_dom(c',c) \quad ^{1,2} = \quad & \nexists r'\ \{R(r')\cdot[\mathtt{1}(r',c)\cdot\neg\mathtt{1}(r',c')\\
& +\mathtt{0}(r',c')\cdot\ \nexists r\ [R(r)\cdot\mathtt{0}(r,c)\cdot\ \nexists c''\ [C(c'')\cdot(c''\neq c')\cdot\mathtt{1}(r,c'')\cdot\neg\mathtt{1}(r',c'')]\,]]\}\\
\beta_dom(p',p) \quad ^{3} = \quad & \nexists d'\ \{\exists c'\ (R(c',d'))\cdot(p\supseteq d')\cdot(p'\not\supseteq d')\}\\
& \cdot\ \nexists d'\ \{R(p',d')\cdot\ \nexists d\ [R(p,d)\cdot\ \nexists q\ [C(q)\cdot(q\neq p')\cdot(q\supseteq d)\cdot(q\not\supseteq d')]\,]\}\}
\end{aligned}$$

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., $\mathtt{1}(r',c)\cdot\neg\mathtt{1}(r',c')$), *or* (2) c' has a 0 in row r' (i.e., $\mathtt{0}(r',c')$) 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 $\nexists c''\,[C(c'')\cdot(c''\neq c')\cdot\mathtt{1}(r,c'')\cdot\neg\mathtt{1}(r',c'')]$ 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 10.8.7 *The set of dominated columns in a table (R, C) can be computed as:*

$$\begin{aligned} D(c) \quad &^{1,2}= \quad C(c) \cdot \exists c' \, [C(c') \cdot (c' \neq c) \cdot col\_dom(c', c)] \\ D(p) \quad &^{3}= \quad C(p) \cdot \exists p' \, [C(p') \cdot (p' \neq p) \cdot col\_dom(p', p)] \end{aligned}$$

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 10.8.8 *The following computations delete a set of columns $D(c)$ from a table (R, C) and all rows intersecting these columns in a 0.*

$$\begin{aligned} C(c) \quad &^{1,2}= \quad C(c) \cdot \neg D(c) \\ R(r) \quad &^{1,2}= \quad R(r) \cdot \nexists c \, [D(c) \cdot \mathtt{0}(r, c)] \end{aligned}$$

$$\begin{aligned} C(p) \quad &^{3}= \quad C(p) \cdot \neg D(p) \\ R(c, d) \quad &^{3}= \quad R(c, d) \cdot \neg D(c) \end{aligned}$$

Proof: The first computation removes columns in $D(c)$ from the set of columns $C(c)$. The expression $\exists c \, [D(c) \cdot \mathtt{0}(r, c)]$ defines all rows r intersecting the columns in D in a 0. They are deleted from the set of rows R. ■

10.8.4 Row Dominance

Definition 10.8.5 *A row r'* **dominates** *another row r if r has all the 1's and 0's of r'.*

Theorem 10.8.9 *The row dominance relation can be computed by:*

$$\begin{aligned} row\_dom(r', r) \quad &^{1,2}= \quad \nexists c \, \{C(c) \cdot [\mathtt{1}(r', c) \cdot \neg \mathtt{1}(r, c) + \mathtt{0}(r', c) \cdot \neg \mathtt{0}(r, c)]\} \\ row\_dom(c', d', c, d) \quad &^{3}= \quad \nexists p \, [C(p) \cdot (p \supseteq d') \cdot (p \not\supseteq d)] \cdot [unate\_row(c') + (c' = c)] \end{aligned}$$

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 10.8.10 *Given a table $(R(r), C(c))$, the set of unate row labels r can be computed as*

$$unate\_row(r)\ ^{1,2} = \nexists c\ [C(c) \cdot \mathtt{0}(r,c)].$$

Given a table $(R(c,d), C(p))$, the set of unate row labels c can be computed as

$$unate\_row(c)\ ^{3} = \nexists p\ [C(p) \cdot (p = c)] = \nexists c\ C(c).$$

Theorem 10.8.11 *The set of rows not dominated by other rows can be computed as:*

$$\begin{aligned} R(r) \quad ^{1,2} = \quad & R(r) \cdot \nexists r'\ [R(r') \cdot (r' \neq r) \cdot row\_dom(r', r)] \\ R(c,d) \quad ^{3} = \quad & R(c,d) \cdot \nexists c', d'\ \{R(c',d') \cdot [(c',d') \neq (c,d)] \cdot row\_dom(c', d', c, d)]\} \end{aligned}$$

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. ■

10.8.5 Essential Columns

Definition 10.8.6 *A column c is an* **essential column** *if there is a row having a 1 in column c and 2 everywhere else.*

Theorem 10.8.12 *The set of essential columns can be computed by:*

$$\begin{aligned} ess\_col(c) \quad ^{1} = \quad & C(c) \cdot \exists r\ \{R(r) \cdot \mathtt{1}(r,c) \cdot \nexists c'[C(c') \cdot (c' \neq c) \cdot (\mathtt{0}(r,c') + \mathtt{1}(r,c'))]\} \\ ess\_col(c) \quad ^{2} = \quad & C(c) \cdot \exists r\ \{R(r) \cdot \mathtt{1}(r,c) \cdot unate\_row(r) \cdot \nexists c'\ [C(c') \cdot (c' \neq c) \cdot \mathtt{1}(r,c')]\} \\ ess\_col(p) \quad ^{3} = \quad & C(p) \cdot \exists c, d\ \{R(c,d) \cdot (p \supseteq d) \cdot unate\_row(c) \cdot \nexists p'\ [C(p') \cdot (p' \neq p) \cdot (p' \supseteq d)]\} \end{aligned}$$

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., $\mathtt{1}(r,c)$), and (2) there is not another *different* column intersecting the row in a 1 or 0 (i.e., $\nexists c'\ [C(c') \cdot (c' \neq c) \cdot (\mathtt{0}(r,c') + \mathtt{1}(r,c'))]$).

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., $\mathtt{1}(r,c)$), 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., $\nexists c'\ [C(c') \cdot (c' \neq c) \cdot \mathtt{1}(r,c')]$). ■

Theorem 10.8.13 *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:

$$\begin{aligned} solution(c) \quad &^{1,2}= \quad solution(c) + ess\_col(c) \\ C(c) \quad &^{1,2}= \quad C(c) \cdot \neg ess\_col(c) \\ R(r) \quad &^{1,2}= \quad R(r) \cdot \nexists c\,[ess\_col(c) \cdot \mathtt{1}(r,c)] \end{aligned}$$

$$\begin{aligned} solution(p) \quad &^{3}= \quad solution(p) + ess\_col(p) \\ C(p) \quad &^{3}= \quad C(p) \cdot \neg ess\_col(p) \\ R(c,d) \quad &^{3}= \quad R(c,d) \cdot \neg ess\_col(c) \end{aligned}$$

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. ■

10.8.6 Unacceptable Columns

Definition 10.8.7 *A column c is an* **unacceptable column** *if there is a row having a 0 in column c and 2 everywhere else.*

Theorem 10.8.14 *The set of unacceptable columns can be computed by:*

$$\begin{aligned} unacceptable\_col(c) \quad &^{1}= \quad C(c) \cdot \exists r\,\{R(r) \cdot \mathtt{0}(r,c) \cdot \nexists c'\,[C(c') \cdot (c' \neq c) \cdot \mathtt{0}(r,c')]\} \\ &\qquad\qquad \cdot \nexists c'\,[C(c') \cdot \mathtt{1}(r,c')]\} \\ unacceptable\_col(c) \quad &^{2}= \quad C(c) \cdot \exists r\,\{R(r) \cdot \mathtt{0}(r,c) \cdot \nexists c'\,[C(c') \cdot \mathtt{1}(r,c')]\} \\ unacceptable\_col(p) \quad &^{3}= \quad C(p) \cdot \exists d\,\{R(p,d) \cdot \nexists p'\,[C(p') \cdot (p' \supseteq d)]\} \end{aligned}$$

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., $\nexists c'\,[C(c') \cdot (c' \neq c) \cdot \mathtt{0}(r,c')]$), and (3) no column c' intersects that row r in a 1 (i.e., $\nexists c'\,[C(c') \cdot \mathtt{1}(r,c')]$). Condition 2 is not needed if we assume that each row contains at most one 0. ■

10.8.7 Unnecessary Columns

Definition 10.8.8 *A column is an* **unnecessary column** *if it does not have any 1 in it.*

Theorem 10.8.15 *The set of unnecessary columns can be computed as:*

$$\begin{aligned} unnecessary\_col(c) &\overset{1,2}{=} C(c) \cdot \nexists r\ [R(r) \cdot \mathtt{1}(r,c)] \\ unnecessary\_col(p) &\overset{3}{=} C(p) \cdot \nexists c,d\ [R(c,d) \cdot (p \supseteq d)] \end{aligned}$$

Proof: A column $c \in C$ is unnecessary if no row $r \in R$ intersects it in a 1. ■

Theorem 10.8.16 *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 10.8.8 by setting

$$\begin{aligned} D(c) &\overset{1,2}{=} unacceptable\_col(c) + unnecessary\_col(c) \\ D(p) &\overset{3}{=} unacceptable\_col(p) + unnecessary\_col(p) \end{aligned}$$

Proof: Obvious. ■

10.9 Other Implicit Covering Table Manipulations

To have a fully implicit binate covering algorithm as described in Section 10.6, 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.

10.9.1 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 10.8.

$Lmax$ 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 10.8: Pseudo-code for $Lmax$.

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.

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 2 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 10.9. 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.

10.9.2 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.

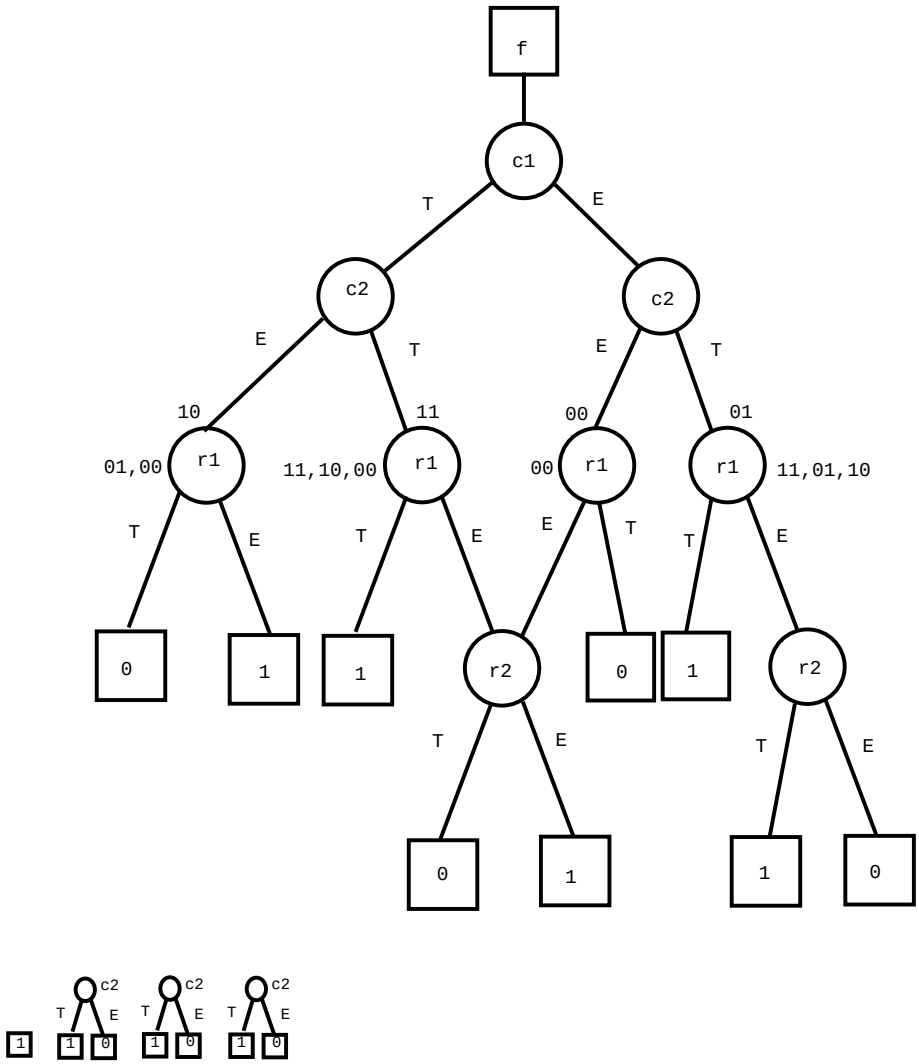


Figure 10.9: BDD of $F(r, c)$ to illustrate the routine *Lmax*

more quickly a good solution. We adopt a simplified selection criterion: select a column with a maximum number of 1's. By defining $F'(r,c) = R(r) \cdot C(c) \cdot \mathtt{1}(r,c)$ 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.

10.9.3 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 $F''(c,r) = R(r) \cdot unate\_row(r) \cdot C(c) \cdot \mathtt{1}(r,c)$. 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:

$$F''(c,r) = F''(c,r).\ \nexists c'\ \{\exists r'\ [shortest(r') \cdot F''(c',r')] \cdot F''(c',r)\}$$

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.

10.9.4 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 $\mathtt{01}(r,c) = \mathtt{0}(r,c) + \mathtt{1}(r,c)$. The implicit algorithm in Figure 10.10 takes a table description in terms of its set of rows $R(r)$, its set of columns $C(c)$ and the nonempty-entry relation $\mathtt{01}(r,c)$, 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

$n\_way\_partition(R(r), C(c), \mathtt{01}(r,c))$ {
    $n = 0$
    while (R not empty) {
        $k = 0$
        $R_0(r) = Lmax(R(r) \cdot C(c) \cdot \mathtt{01}(r,c))$
        repeat {
            $k = k + 1$
            $C_k(c) = C(c) \cdot \exists r \, \{R_{k-1}(r) \cdot \mathtt{01}(r,c)\}$
            $R_k(r) = R(r) \cdot \exists c \, \{C_k(c) \cdot \mathtt{01}(r,c)\}$
        } 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 $\{(R^i, C^i) : 0 \leq i \leq n-1\}$
}

Figure 10.10: Implicit n-way partitioning of a covering table.

sub-block in (R, C).

$$\begin{aligned} R_0(r) &= Lmax(R(r) \cdot C(c) \cdot \mathtt{01}(r,c) \ , \ c) \\ C_k(c) &= C(c) \cdot \exists r \, \{R_{k-1}(r) \cdot \mathtt{01}(r,c)\} \\ R_k(r) &= R(r) \cdot \exists c \, \{C_k(c) \cdot \mathtt{01}(r,c)\} \end{aligned}$$

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. [65] provides a more detailed explanation.

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 $C(c) \cdot \exists r\ [R_0(r) \cdot \mathtt{01}(r,c)]$ (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 $R(r) \cdot \exists c\ [C_1(c) \cdot \mathtt{01}(r,c)]$. 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 10.10 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 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.

10.10 Implicit Two-level Logic Minimization

The implicit computations presented to manipulate a binate table are valid *a fortiori* when the table is unate. In the latter case, however, more specialized algorithms can be designed to exploit fully the features of the simpler problem. Historically speaking, an implicitization of covering problems has been carried on first for the case of unate tables generated in the minimization of two-level logic functions.

Given a boolean function f, consider the problem of finding a minimum two-level cover. A classical exact algorithm by Quine and McCluskey reduces it to a unate covering problem where the rows of the table are minterms and the columns of the table are primes of the function. There is a 1 at the intersection of a row and column, if the prime associated to the column contains the minterm associated to the row. An efficient implementation of unate covering is provided in the program ESPRESSO. In that implementation an improvement has been introduced, because there is only one row for each set of minterms that are covered by the same set of primes. In other words, the table is constructed in such a way that there are no equal rows in it.

The set of all primes and minterms may be exponential in the number of input variables. Manipulating a table with an exponential number of rows and columns may add another exponential blow-up. To overcome these problems, researchers at Bull [29, 30] and UCB [53] have represented the set of primes and the unate table with logic functions implemented with ROBDD's. The key steps have been:

1. Define a boolean space where all those sets could be represented.

2. Transform the computation of the primes, unate table and the table reduction operations into operations on boolean functions defined on the boolean space of the problem.

An whole suite of papers has been produced by the French group [25, 79, 27, 26, 24, 28, 29, 30, 22]. Here we will outline only the key steps of this approach.

We remind that a literal is a propositional variable x_k or its negation $\overline{x_k}$. P_n is the set of products that can be built from the set of variables $\{x_1, \ldots, x_n\}$. The subset relation $\subseteq$ is a partial order on the set P_n. P is maximal iff there do not exist two products p and p' of P such that $p \subset p'$. A product p is an implicant of a boolean function f iff $p \subseteq \{x \in \{0,1\}^n \mid f(x) \neq 0\}$. A product p is a prime implicant of f iff it is a maximal element of the set of implicants of f with respect to $\subseteq$. Any subset P of P_n can be partitioned in the following way:

$$P = P_{1_k} \cup (\{\overline{x_k}\} \times P_{\overline{x_k}}) \cup (\{x_k\} \times P_{x_k})$$

where P_{1_k} is the set of products of P where neither the variable x_k nor $\overline{x_k}$ occurs; $P_{\overline{x_k}}$ (respectively P_{x_k}) is the set of products of P where $\overline{x_k}$ (x_k) occurs, after dropping $\overline{x_k}$ (x_k).

A boolean space to represent all products can be obtained by a number of variables double with respect to the number of input variables of f. It is the *metaproduct* representation in the literature by researchers at Bull and the *extended space* in the literature by researchers at UCB. The basic idea is to encode the presence of x_k or $\overline{x_k}$ or both (i.e. neither literal appears explicitly in the product) with two bits.

The computation of primes reduces to finding the maximal products over all implicants of f. The following recursive computation finds all prime implicants:

$$\begin{aligned} Prime(f) \;=\; & Prime(f_{\overline{x_k}} \wedge f_{\overline{x_k}}) \\ & \cup \{\overline{x_k}\} \times (Prime(f_{\overline{x_k}}) \setminus Prime(f_{\overline{x_k}} \wedge f_{\overline{x_k}})) \\ & \cup \{x_k\} \times (Prime(f_{x_k}) \setminus Prime(f_{\overline{x_k}} \wedge f_{\overline{x_k}})) \end{aligned}$$

It is easy to transpose this computation to the case of the extended space or metaproducts representation.

The table covering problem can now be described by the triple $< Q, P, \subseteq >$, where Q is the set of minterms of f, P is the set of primes of f and $\subseteq$ describes the table building relation. Notice that this is already a progress with respect to the traditional approach because we do not represent

directly the table, but we have instead an operator ($\subseteq$) to infer the table entries. This is a special case of the encoding scheme of binate tables for exact state minimization, previously reported. Strictly speaking, this reformulation is not tied to the fact of using an implicit representation. It could be used also with an explicit representation. When coupled with a BDD-based representation it lends itself to very efficient algorithms, because the final size of the representation is not linearly proportional to the number of primes computed.

A unate table is reduced by applying row and column dominance and detecting essential primes. Row dominance is stated as follows.

Definition 10.10.1 *A row R_j dominates another row R_i if for all columns C_k, $M_{j,k} = 1 \Rightarrow M_{i,k} = 1$.*

In the case of $< Q, P, \subseteq >$, this translates into:

$$q \preceq_Q q' \Leftrightarrow (\forall p \in P(q' \subseteq p) \Rightarrow (q \subseteq p))$$

Moreover, if there are rows that intersect exactly the same set of columns, i.e. they are equivalent, one should compute this equivalence relation and then replace each equivalence class with one representative (called sometimes *projection* operation [78]). Row dominance should then be applied to these representatives only.

Instead of using such a projection and then applying the definition of dominance relation, one can define a row transposing function that maps the rows on objects whose manipulation can be done more efficiently. The maximal elements of the transposed objects are the dominating rows.

The basic idea is that each row of a covering table corresponds to a cube, called *signature cube*, that is the intersection of the primes covering the minterm associated to the row. This was noticed first in [99]. A rigorous theory and an efficient algorithm were developed at UCB [89]. The steps of the algorithm follow. Compute the signature cube of the each cube of an arbitrary initial cover and make irredundant the resulting cover. Using the fact that for each cube of an arbitrary irredundant cover of signature cubes, there is some essential signature cube contained by it, obtain the irredundant cover of essential signature cubes (called minimum canonical cover). For each cube of the minimum canonical cover, generate the set of primes containing it (the essential signature set). Solve the resulting unate covering problem as usual. The resulting unate covering problem is exactly what one could get by applying row domination to the minterms/primes table.

One can define a row transposing function $\tau_Q(Q)$ based on the idea of signature cubes.

Definition 10.10.2 *$\tau_Q : Q \longrightarrow P_n$ is defined as:*

$$\tau_Q(q) = \bigcap_{\{p \in P \mid q \subseteq p\}} p$$

In other words, each element of $\tau_Q(Q)$ is obtained by an element q of Q, by intersecting all elements of P that cover q.

The following theorem relates row dominance to the row transposing function.

Theorem 10.10.1 *The function τ_Q is such that*

$$q \preceq_Q q' \Leftrightarrow \tau_Q(q) \subseteq \tau_Q(q').$$

Given a set covering problem $(Q, P, \subseteq)$, the function $max_{\subseteq}\tau_Q(Q)$ computes the maximal elements of the set $\tau_Q(Q)$, i.e., the dominating rows.

Since the range τ_Q is P_n, the computation of τ_Q can be easily transposed to the case of the extended space or metaproducts representation. The most obvious implementation would use quantified boolean formulas, but in practice they tend to produce huge intermediate ROBDD's. A quantifier free recursive computation of $max_{\subseteq}\tau_Q(Q)$ has given better experimental results.

We present now a pseudo-code description of $MaxTauQ(Q, P, k)$, the recursive procedure used to compute $max_{\subseteq}\tau_Q(Q)$. We define first two auxiliary functions $Supset$ and $Subset$:

$$Supset(P, Q) = \{p \in P \mid \exists q \in Qp \supseteq q\}$$

$$Subset(P, Q) = \{p \in P \mid \exists q \in Qp \subseteq q\}$$

Theorem 10.10.2 *$MaxTauQ(Q, P, 1)$ computes $max_{\subseteq}\tau_Q(Q)$.*

Proof: The terminal cases are easy. Consider a variable x_k. One can divide the set P in three subsets: P_{x_k}, the products of P in which x_k occurs, $P_{\overline{x_k}}$, the products of P in which $\overline{x_k}$ occurs and P_{1_k}, the products of P in which neither x_k nor $\overline{x_k}$ occurs. Similarly, one can divide the set Q in three subsets: Q_{x_k}, the products of Q in which x_k occurs, $Q_{\overline{x_k}}$, the products of Q in which $\overline{x_k}$ occurs and Q_{1_k}, the products of Q in which neither x_k nor $\overline{x_k}$ occurs.

The products of $Q_{\overline{x_k}}$ can be contained by products of $P_{\overline{x_k}}$ or by products of P_{1_k}. The products of Q_{x_k} can be contained by products of P_{x_k} or by products of of P_{1_k}. The products of Q_{1k} can be contained only by products of P_{1_k}. $K0$ has the products of $Q_{\overline{x_k}}$ contained by products

$MaxTauQ(Q, P, k)$ {
  if $Q = \emptyset$ or $Q = \emptyset$ {
  if $P = \{1\}$ return $\{1\}$
  $K0 = Subset(Q_{\overline{x_k}}, P_{\overline{x_k}})$
  $K1 = Subset(Q_{x_k}, P_{x_k})$
  $K0 = Q_{1k} \cup (Q_{\overline{x_k}} \setminus K0) \cup (Q_{x_k} \setminus K1)$
  $R = MaxTauQ(K, P_{1_k}, k+1)$
  $R0 = MaxTauQ(K0, P_{1_k} \cup P_{\overline{x_k}}), k+1)$
  $R1 = MaxTauQ(K1, P_{1_k} \cup P_{x_k}), k+1)$
  return $R\cup$
    $\{\overline{x_k}\} \times Subset(R0, R))\cup$
    $\{x_k\} \times Subset(R1, R))\cup$
}

Figure 10.11: Recursive computation of $max_{\subseteq} \tau_Q(Q)$

of $P_{\overline{x_k}}$. $K1$ has the products of Q_{x_k} contained by products of P_{x_k}. K has the products of Q_{1_k}, the products of $Q_{\overline{x_k}}$ that are not contained by products of $P_{\overline{x_k}}$ and the products of Q_{x_k} that are not contained by products of P_{x_k}.

Also the set $MaxTauQ(Q, P, 1)$ can be divided in three subsets: the set of products in which x_k occurs, the set of products in which $\overline{x_k}$ occurs and the set of products of P in which neither x_k nor $\overline{x_k}$ occurs. The last set is given by R, that is $MaxTauQ(K, P_{1_k}, k+1)$. Indeed in R the second argument is P_{1_k}, the set of products of P where neither x_k nor $\overline{x_k}$ occurs. The first argument is K that includes the products of Q where x_k nor $\overline{x_k}$ occurs and so can be contained only by products of P_{1_k}, and the products of Q where either x_k or $\overline{x_k}$ occurs but they are not covered by P_{x_k} or $P_{\overline{x_k}}$ and so they can be covered only by P_{1_k}. The second set is obtained from $R0$, that is $MaxTauQ(K0, P_{1_k} \cup P_{\overline{x_k}}, k+1)$, by the following modification. In the first argument of $R0$ there are the products of Q where $\overline{x_k}$ occurs, which are contained by the products of P in the second argument. A product in $R0$ must be multiplied by $\{\overline{x_k}\}$ because for sure each $q \in K0$ is contained by a product of $P_{\overline{x_k}}$, and by definition of $\tau_Q(q)$ one must intersect all the products that contain q. But before multiplying by $\{\overline{x_k}\}$ we must subtract from $R0$ the products contained in R ($Subset(R0, R)$), because if a product $r0$ of $R0$ is contained by a product r of R (or is equal to) it

means that there are $q \in K$ and $q0 \in K0$ such that $\tau_Q(q) \supseteq \tau_Q(q0)$ (because r contains $r0$ and $r0$ is multiplied by $\{\overline{x_k}\}$) and we want to keep only $\tau_Q(q)$ because we are computing $max_{\subseteq}\tau_Q$. Instead if a product of R is contained by a product of $R0$, the fact that the product of $R0$ must be multiplied by $\{\overline{x_k}\}$ makes the two products not comparable. Therefore $\{x_k\} \times (R0 \setminus Subset(R0, R))$ is the set of products of $MaxTauQ(Q, P, 1)$ in which $\overline{x_k}$ occurs. Replacing verbatim $\{\overline{x_k}\}$ with x_k, the same reasoning applies for the addition coming from $R1$, from which the first set is obtained. ■

After the set $Q' = max_{\subseteq}\tau_Q(Q)$ has been computed, the problem $< Q, P, \subseteq >$ transforms to $< Q', P, R' >$, where $q'R'p$ iff $q' = \tau_Q(q)$ and $q \subset p$. $R' \equiv \subseteq$, since $q \subseteq p$ iff $\tau_Q(q) \subseteq p$. Therefore the new covering problem is $< Q', P, \subseteq >$.

A similar development holds for column dominance.

Definition 10.10.3 *A column C_i dominates another column C_j if for all rows R_k, $M_{k,j} = 1 \Rightarrow M_{k,i} = 1$.*

In the case of $< Q, P, \subseteq >$, this translates into:

$$p \preceq_P p' \Leftrightarrow (\forall q \in Q(q \subseteq p) \Rightarrow (q \subseteq p'))$$

Moreover, if there are columns that intersect exactly the same set of rows, i.e. they are equivalent, one should compute this equivalence relation and then replace each equivalence class with one representative (projection operation). Column dominance should then be applied to these representatives only.

Instead of using such a projection and then applying the definition of dominance relation, one can define a column transposing function that maps the columns on objects whose manipulation can be done more efficiently. The maximal elements of the transposed objects are the dominating columns.

Consider the following column transposing function $\tau_P(p)$:

Definition 10.10.4

$$\tau_P(p) = C(\bigcup_{\{q \in Q | q \subseteq p\}} q),$$

where $C(E) = min_{\subseteq}\{p \in P_n \mid p \supseteq E\}$.

$C(E)$ is the unique smallest product that contains the set E. Here min is an intersection operator, so

$$\tau_P(p) = min_{\subseteq}\{p \in P_n \mid p \supseteq E\},$$

$MaxTauP(Q, P, k)$ {
  if $Q = \emptyset$ or $Q = \emptyset$ {
  if $Q = p_n$ return P
  $K = Supset(P_{1_k}, Q_{1_k}) \cup$
    $Supset(P_{1_k}, Q_{\overline{x_k}}) \cap Supset(P_{1_k}, Q_{x_k})$
  $K0 = Supset(P_{1_k} \cup P_{\overline{x_k}}, Q_{\overline{x_k}}) \setminus K \qquad K1 = Subset(P_{1_k} \cup P_{\overline{x_k}}, Q_{x_k}) \setminus K$
  $R = MaxTauP(Q_{1_k} \cup Q_{\overline{x_k}} \cup Q_{x_k}, K, k+1)$
  $R0 = MaxTauP(Q_{\overline{x_k}}), K0, k+1)$
  $R1 = MaxTauQ(Q_{x_k}), K1, k+1)$
  return $R \cup$
    $\{\overline{x_k}\} \times Subset(R0, R)) \cup$
    $\{x_k\} \times Subset(R1, R)) \cup$
}

Figure 10.12: Recursive computation of $max_{\subseteq} \tau_P(P)$

or,

$$\tau_P(p) = \bigcap \{p \in P_n \mid p \supseteq \bigcup_{\{q \in Q \mid q \subseteq p\}} q\}.$$

The following theorem relates column dominance to the column transposing function.

Theorem 10.10.3 *The function τ_P is such that*

$$p \preceq_P p' \Leftrightarrow \tau_P(p) \subseteq \tau_P(p').$$

Given a set covering problem $(Q, P, \subseteq)$, the function $max_{\subseteq} \tau_P(P)$ computes the maximal elements of the set $\tau_P(P)$, i.e. the dominating columns.

Since the range τ_P is P_n, the computation of τ_P can be easily transposed to the case of the extended space or metaproducts representation. The most obvious implementation would use quantified boolean formulas, but in practice they tend to produce huge intermediate ROBDD's. A quantifier free recursive computation of $max_{\subseteq} \tau_P(P)$ has given better experimental results.

We present now a pseudo-code description of $MaxTauP(Q, P, k)$, the recursive procedure used to compute $max_{\subseteq} \tau_Q(Q)$.

Theorem 10.10.4 $MaxTauP(Q, P, 1)$ *computes* $max_{\subseteq} \tau_P(P)$.

Proof: The terminal cases are easy. Consider a variable x_k. The set K is the set of products of P_{1_k} that contain a product of Q_{1_k}, or that contain a product of $Q_{\overline{x_k}}$ and a product of Q_{x_k}. So K is the set of products p of P such that $\tau_P(p)$ does not contain the literal x_k nor $\overline{x_k}$. Therefore the set R is the set of products of $max_{\subseteq}\tau_P(P)$ that do not contain the literal x_k nor $\overline{x_k}$. The set $K0$ (respectively $K1$) is the set of of products p of P that only contain products of Q where the literal $\overline{x_k}$ (respectively x_k) occurs. Since in the definition of $\tau_P(p)$ one takes an intersection of products (primes that contain the products contained by p), the set $R0$ is the set of products of $\tau_P(P)$ that contain the literal $\overline{x_k}$, and that are maximal with respect to $\tau_P(P)_{\overline{x_k}}$. Since we want only the maximal products with respect to $\tau_P(P)$, from $R0$ one subtracts the products that are contained by a product of R. ■

After the set $P' = max_{\subseteq}\tau_P(P)$ has been computed, the problem $< Q, P, \subseteq >$ transforms to $< Q, P', R' >$, where $qR'p'$ iff $p' = \tau_P(p)$ and $q \subset p$. $R' \equiv \subseteq$, since $q \subseteq p$ iff $q \subseteq \tau_P(p)$. Therefore the new covering problem is $< Q, P', \subseteq >$.

One more table reduction operation is the detection of essential columns.

Definition 10.10.5 *A column is an essential column if it contains the* 1 *of a singleton row.*

Theorem 10.10.5 *The set of essential products is* $E = P \cap max_{\subseteq}\tau_Q(Q)$.

After the set $E = P \cap max_{\subseteq}\tau_Q(Q)$ has been computed, the problem $< Q, P, \subseteq >$ transforms to $< Q \setminus E, P \setminus E, \subseteq >$.

Successive application of row dominance, essential detection and column dominance computes the cyclic core of the unate covering problem. A branch-and-bound procedure, where table reduction is invoked on subtables splitted along a branching column, leads to a final solution, that is a minimum number of primes needed to cover all the minterms. Notice that in the papers by the researchers at Bull *no implicitization* is reported of the choice of a branching column and of a lower bound computation. Implicit formulations of such operations were instead reported first in [66].

In [30] it is stated that the usage of Zero-Suppressed BDD's by Minato [95] instead of ROBDD's [16] resulted in more efficient implicit representations of the computations of the problem.

Chapter 11

Implicit Minimization of GPI's

11.1 Implicit Representations and Manipulations

Algorithms for sequential synthesis have been developed primarily for State Transition Graphs (STG's). STG's have been usually represented in two-level form where state transitions are stored explicitly, one by one. Alternatively, STG's can be represented implicitly with Binary Decision Diagrams (BDD's) [16, 10]. BDD's represent Boolean functions (e.g. characteristic functions of sets and relations) and have been amply reported in the literature [16, 10], to which we refer.

11.1.1 Implicit FSM Representation

A Finite State Machine (FSM) can be represented by a 5-tuple $(I, O, S, \mathcal{T}, \mathcal{O})$. I and O are the sets of input patterns and output patterns. S is the set of states. $\mathcal{T} \subseteq I \times S \times S$ is the transition relation that relates a next state to an input and a present state. $\mathcal{O} \subseteq I \times S \times O$ is the output relation that relates an output to an input and a present state. An FSM, where each (input, state) pair is related to exactly one next state and one output, is a **completely specified FSM**. An **incompletely specified FSM** is one where either the next state or the output is not specified for at least one (input, state) pair.

If a next state is unspecified, no transitions on the (input, state) pair need to be considered for the purpose of state minimization, so they are omitted from $\mathcal{T}$. On the other hand, we represent all unspecified output patterns in $\mathcal{O}$ corresponding to an (input, state) pair. The **transition and**

output relations are given by:

$\mathcal{T}(i,p,n) = 1$ iff n is the specified next state of state p on input i

$\mathcal{O}(i,p,o) = 1$ iff o is a (possibly unspecified) output of state p on i

where i and o are Boolean vectors of signals while p and n are represented by positional-sets defined below.

11.1.2 Positional-set Representation

To perform sequential optimization, one needs to represent and manipulate efficiently sets of states, or state sets, (such as compatibles) and sets of sets of states (such as sets of compatibles). Our goal is to represent any set of sets of states implicitly as a single BDD, 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 BDD.

Suppose a FSM has n states, there are 2^n possible distinct subsets of states. In order to represent collections of them, each subset of states is represented in **positional-set** form, using a set of n Boolean variables, $x = x_1x_2 \ldots x_n$. The presence of a state 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 state s_k is not a member of the set. One Boolean variable is needed for each state because the state can either be present or absent in the set. For example, if $n = 6$, the set with a single state s_4 is represented by 000100 while the set of states $s_2s_3s_5$ is represented by 011010.

A set of sets of states is represented as a set S of positional-sets by a characteristic function $\chi_S : B^n \rightarrow B$ as: $\chi_S(x) = 1$ iff the set of states represented by the positional-set x is in the set S. A BDD representing $\chi_S(x)$ will contain minterms, each corresponding to a state set in S.

11.1.3 Operations on Positional-sets

With our definitions of relations and positional-set notation for representing set of states, useful operators on sets and sets of sets can be derived. We have proposed in [65] a unified notational framework for set manipulation, extending the work by Lin *et al.* in [79]. Here we define some basic operators.

Proposition 11.1.1 *Set* **equality, mirroring, containment**, *and* **strict-containment** *between two positional-sets x and y can be computed by:* $(x = y) \equiv \prod_{k=1}^{n}(x_k \Leftrightarrow y_k)$; $compl(x,y) \equiv \prod_{k=1}^{n}(x_k \Leftrightarrow \neg y_k)$; $(x \supseteq y) \equiv \prod_{k=1}^{n}(y_k \Rightarrow x_k)$; $(x \supset y) \equiv (x \supseteq y) \cdot (x \neq y)$.

Proposition 11.1.2 *Given two sets of positional-sets,* **complementation**, **union**, **intersection**, *and* **sharp** *can be performed on them as logical operations* $(\neg, +, \cdot, \cdot\neg)$ *on their characteristic functions.*

Proposition 11.1.3 *The* **Maximal** *of a set* F *of sets is the set containing sets in* F *not strictly contained by any other set in* F*, and is given by:*

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

The term $\exists y\, [\chi_F(y) \cdot (y \supset x)]$ is true iff there is a positional-set y in χ_F such that $y \supset x$. In such a case, x cannot be in the maximal set by definition, and are taken away from $\chi_F(x)$. One defines symmetrically the **Minimal** of a set.

Proposition 11.1.4 *The operation* **Set_Minimal**$_b(F(a,b))$ *keeps in the relation* $F(a,b)$ *only the pairs* (a,b) *such that there is no* a' *related to exactly a proper subset of the* b*'s with which* a *is in relation and it is computed by:*

$$Set\_Minimal_b(F(a,b)) = F(a,b) \cdot \nexists c\{\exists d\, F(c,d) \cdot \forall d\, [F(c,d) \Rightarrow F(a,d)] \cdot \exists d\, [\neg F(c,d) \cdot F(a,d)]\}.$$

Each a is connected to a set of b's. By varying a, we have all sets of b's and we keep the minimal ones of them. We keep in the minimality relation only the pairs (a,b) where a is connected to a minimal set of b's. The fact that the minimality is computed over the b's is indicated by the subscript b of $Set\_Minimal$. It is necessary to add the term $\exists d\ F(c,d)$ in order to constrain the c's in the following implication.

Example 11.1.1 *Given the relation* $F(a,b)$ *with elements* $(001, 011, 100, 101, 211, 201, 210)$*, the relation Set_Minimal_*$b(F(a,b))$ *has elements* $(001, 011, 101, 111)$*, the relation Minimal_*$b(F(a,b))$ *has elements* $(001, 100, 201, 210)$.

An often used family of operators is **Tuple** that computes for a given k the k-out-of-n positional-sets. For instance $Tuple_{|x|}(x)$ gives the universe set on the support x, $Tuple_0(x)$ gives the empty set on the support x.

Finally we need the operators of the family **Lmin** and **Lmax**, first proposed in [66], to which we refer for detailed explanations. Besides those already described in [66], we introduce a new operator **Multi_Lmin**, that is a variant of $Lmin$. Given a binary relation $F(r,c)$ as a BDD, $Lmin(F(r,c),r)$ computes $F_{Lm}(c)$, the set of c's which relate to the minimum 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 minimum count. Instead the algorithm $Lmin$ is implemented as a primitive BDD operator that traverses each node of F exactly once. Variables in c are required to be ordered above (before) variables in r.

As a variant of the $Lmin$ operator, the $Multi\_Lmin(R(l,r,i,p,x,y),(i,p))$ operator computes a relation $R_{MLm}(l,r,x,y)$ such that, for each (x,y), (l,r) relates to the minimum number of (i,p) in $R(l,r,i,p,x,y)$, i.e., for a given (x,y), it finds $Lmin(R|_{x,y}(l,r,i,p,x,y),(i,p))$. Again the computation is performed with a BDD primitive that traverses once each node of R. Variables (x,y) are required to be ordered above (before) (l,r) which in turn must be above (i,p).

11.1.4 Relations for Implicit Encodeability of GPI's

In the next sections we will present in detail a set of implicit computations that generate the GPI's and select a minimal subset of encodeable GPI's that cover the original FSM. Here we introduce the basic relations used in the implicit algorithms. Others will be presented in the coming sections.

- $i =$ input vector
- $p =$ positional set of present states
- $n =$ positional set of next states (tag)
- $m =$ positional set of next states (tag)
- $o =$ output vector (tag)
- $cover\_f(i,p,n,m,o) =$ onset of the original FSM
  where the combination (i,p) denotes a cube in the input/present-state part of the STT, n represents the next state tag of the cube, and o is the output vector.
  $cover\_fd(i,p,n,m,o) =$ union of onset and dcset of the original FSM
  $cover\_r(i,p,n,m,o) =$ offset of the original FSM
- $M(i,p,n,o) =$ minterms of a STT
  where the combination (i,p) denotes a minterm in the input/present-state part of the STT, n represents the next state tag of the minterm, and o is the output vector.
  $M(i,p,n) = \exists o\, M(i,p,n,o)$
  $M(i,p) = \exists n, o\, M(i,p,n,o)$

- $M_n(i,p,n,o) =$ next-state minterms of a STT
  where the field o is null

- $M_o(i,p,n,o) =$ output minterms of a STT
  where the field n is null

- $GMI(i,p,n,m,o) =$ minterms of a STT
  where the combination (i,p) denotes a minterm in the input/present-state part of the STT, n represents the complemented next state tag of the minterm, m represents the complemented present state part, and o is the output vector.

- $GPI(i',p',n',m',o') =$ the set of the GPI's
  where the combination (i',p') denotes a cube (GPI) in the input/present-state part of the STT, n' represents the complemented next state tag of the GPI, m' represents the complemented present state part, and o' is the output tag.

- $P(i',p',n',o') =$ the set of the GPI's
  where the combination (i',p') denotes a cube (GPI) in the input/present-state part of the STT, n' represents the next state tag of the GPI, and o' is the output tag.

- $G(i',p',n',o') =$ a selection of GPI's
  where the combination (i',p') denotes a cube (GPI) in the input/present-state part of the STT, n' represents the next state tag of the GPI, and o' is the output tag.
  $G(i',p',n') = \exists o'\ G(i',p',n',o')$
  $G'(i',p',n') =$ the set of GPI's which have not been selected yet

- $D(l,r) =$ a set of encoding dichotomies
  Each dichotomy $(l_1,l_2,\ldots,l_i;r_1,r_2,\ldots,r_j)$ is represented by a pair of positional sets (l,r).

11.2 Implicit Generation of GPI's and Minterms

11.2.1 Implicit Generation of GPI's

The step of computing the set of GPI's can be reduced to computing the prime implicants of a boolean function associated to the given FSM [39]. A very fruitful recent research effort [53, 30] succeeded in finding efficiently by implicit computations the prime implicants of a boolean function.

We refer to [53, 30] for a complete treatment of the topic and we report here a few facts required in our application.

If a multiple-valued function is represented in positional notation [114], cubes of the onset (or dcset or offset) of the function can be mapped into vertices of a suitable Boolean space (**extended Boolean space**), which has as many variables as the length of a positional vector. So sets of minterms, implicants and primes of a multiple-valued boolean function are subsets of vertices in the extended Boolean space. Key properties of this extended space representation are that primality of an implicant corresponds to maximality when the order is given by set inclusion and that set operations can be performed as boolean operations on the characteristic functions of the sets. Here we review only some core facts and show the sequence of computations to compute the prime implicants of a multi-valued function.

Consider a cube $S = X_1^{S_1} \times X_2 S_2 \ldots \times X_n^{S_n}$. Each S_i is a subset of $0, 1, \ldots, P_i$, where P_i is the set of possible values of the i-th variable X_i.

Definition 11.2.1 *A cube $S = X_1^{S_1} \times X_2 S_2 \ldots \times X_n^{S_n}$ is represented by the vertex $\prod_{i,j} x_{ij}$, where $x_{ij} = 0$ if $j \notin S_i$ and $x_{ij} = 1$ if $j \in S_i$, in the boolean space $B^{\sum |P_i|}$. This representation is called* **extended space representation** .

Example 11.2.1 *The cube $X^{\{0,2\}} \times X^{\{0,1\}}$, where $P_1 = \{0, 1, 2\}$ and $P_2 = \{0, 1\}$ is represented by the vertex $x_{11}\overline{x}_{12}x_{21}x_{22}x_{23}$, i.e., $(1, 0, 1, 1, 1)$ in B^5.*

In this way, each cube is mapped into a unique vertex of the extended Boolean space, except for the empty cube (i.e., the cube which has at least a part completely empty). The empty cube is mapped into a set of points in the extended Boolean space, the so-called null points.

Definition 11.2.2 *The* **null set** *or set of* **null points** *is the representation in the extended space of the null cube of the original function space.*

Proposition 11.2.1 *The set of null points is given by $null(x) = \sum_i \prod_j \overline{x}_{ij}$*

Definition 11.2.3 *The* **vertex set** *is the representation in the extended space of all the vertices of the original function space.*

The vertex set can be computed by the following proposition.

Proposition 11.2.2 *The vertex set is given by $vertex(x) = \prod_i \sum_j x_{ij} \prod_{k \neq j} \overline{x}_{ik}$.*

Figure 11.1 shows how to compute implicitly the prime implicants of a multi-valued function.

procedure implicit_pi_generation($cover\_fd$) {

/* minterms of (onset + dcset) */

$vertex\_fd(i'p'm'n'o') = \exists ipmno[cover\_fd(ipmno)vertex(i'p'm'n'o')(ipmno \supseteq i'p'm'n'o')]$

/* minterms of offset */

$vertex\_r(i'p'm'n'o') = vertex(i'p'm'n'o') - vertex\_fd(i'p'm'n'o')$

/* implicants of (onset + dcset) and null cubes */

$impl\_null(i'p'm'n'o') = U(i'p'm'n'o') - \exists ipmno\ vertex\_r(ipmno)(i'p'm'n'o' \supseteq ipmno)$

/* prime implicants */

$cprime(i'p'm'n'o') = maximal(impl\_null(i'p'm'n'o'))$

/* remove remaining null cubes (e.g., 00 11 111 11111111) */

$prime(i'p'm'n'o') = cprime(i'p'm'n'o') - null\_cube(i'p'm'n'o')$

}

Figure 11.1: Implicit computation of prime implicants

11.2.2 Reduced Representation of GPI's and Minterms

GPI's are found in a (extended) representation $GPI(i', p', n', m', o')$, that can be easily converted to a (reduced) representation $P(i', p', n', o')$. The meaning of the different fields of GPI and P has been given in Section 11.1.4. The extended representation has the advantage that column dominance, which requires the same present state literal, can be done simply by checking containment of the representations. A GPI (column) covers a minterm (row) iff the GPI contains the minterm. The reduced representation has the advantage that a smaller number of variables is required. This advantage is not trivial when many sets of variables are required.

To get the reduced representation one must transform back from the (i, p, n, m, o) space into the original (i, p, n, o) space, while enforcing that the transformation conventions are satisfied. The reduced representation of the primes is given by:

$$P(i', p', n', o') = (\tilde{m} \rightarrow n')\exists n'(\exists m' GPI(i', p', n', m', o') \cdot compl(n', \tilde{m}))$$

The equation drops the m' field of GPI and converts the n' field from complemented 1-hot encoding to 1-hot encoding.

The reduced representation of the minterms is given by:

$$red\_cover\_f(i', p', n', o') = (\tilde{m} \rightarrow n')\exists n'(\exists m' cover\_f(i', p', n', m', o') \cdot compl(n', \tilde{m}))$$

$$M(i,p,n,o) = \exists m(vertex(i,p,n,m,o)Tuple_0(m))$$
$$\exists i'p'n'o'[red\_cover\_f(i',p',n',o')(ipno \subseteq i'p'n'o')]$$

red_cover_f is the reduced representation of the onset of the original FSM. It is obtained by dropping the m field of $cover\_f$ and converting the n field from complemented 1-hot encoding to 1-hot encoding. The equation for M selects the minterms of $vertex$ with an empty m field and then keeps only those that are in the reduced representation of the onset of the original FSM. **Output minterm** is a minterm where $n = Tuple_0(n)$ and **next-state minterm** is a minterm where $o = Tuple_0(o)$.

11.2.3 Pruning of Primes

Some primes can be removed because they do not correspond to GPI's. One removes primes of one of the two following types:

1. Primes that are covered by another prime, with full present state part and with the same next state and output tags.

2. Primes with full next state tag and null output tag.

The first operation is implemented by:

$$P(i',p',n',o') = P(i',p',n',o') - \exists i,p(P(i,p,n',o')(i \supseteq i')Tuple_{|p|}(p)(ip \neq i'p'))$$

Notice that the clause $ip \neq i'p'$ avoids the self-cancellation of primes with a full present state part. The second operation is implemented by:

$$P(i',p',n',o') = P(i',p',n',o') - Tuple_{|n'|}(n')Tuple_0(o')$$

11.3 Implicit Selection of GPI's

Once the GPI's, or a subset of them, have been computed one must select a subset of them that is encodeable and covers the original FSM.

11.3.1 Implicit Selection of a Cover of GPI's

Once GPI's and minterms are obtained, one sets up a covering problem. The rows of the table are the minterms and the columns are the GPI's. If the next state tag of a GPI is a superset of

the next state tag of a minterm and the GPI asserts all the outputs that the minterms asserts then there is a 1 at the intersection of the given GPI and minterm. The table is unate, i.e., either an entry is 1 or it is empty. We will use an implicit table solver to select a subset of GPI's that cover the minterms. Implicit algorithms to solve binate covering problems were presented in [66]. We implemented two implicit binate solvers: a specialized one with a fixed table definition rule and a general one, where one specifies by means of functions how entries are evaluated. Notice that for this application only a unate solver is required, but we do not have a specialized unate solver, which could capitalize on the restricted type of input. Here we could use either binate solver program and the specialized one might be faster. But in Section 11.3.3 it will be necessary to use the general implicit binate solver. So the latter will be used in both cases. In our application there is a 1 at the intersection of a given minterm and GPI iff the next state tag of the GPI is a superset of the next state tag of the minterm and the GPI asserts all the outputs that the minterm does. The implicit general binate solver requires the sets of columns and rows and a rule to compute a table entry. In this case they are:

1. Columns are $C(q) = P(q)$.

2. Rows are $R(d) = R_u(d) = M(d)$.

3. The table entry at the intersection of the column labelled by $q \in C$ and of the row labelled by $d \in R$ is 1 iff $q \supseteq d$.

4. The table entry at the intersection of the column labelled by $q \in C$ and of the row labelled by $d \in R$ is never 0.

If the minterms and GPI's are in the reduced representation it is sufficient to set $d = i, p, n, o$ and $q = i', p', n', o'$ to guarantee that there is a 1 at row c, d and column q iff $q \supseteq d$, since there is a 1 iff $i', p', n', o' \supseteq i, p, n, o$ [1].

[1] If, instead, the minterms and GPI's are in the extended representation, setting $d = i, p, n, m, o$ and $q = i', p', n', m', o'$ there is a 1 at row c, d and column q iff $i' \supseteq i, p' \supseteq p, n' \subseteq n, m' \subseteq m, o' \supseteq o$. The latter rule is different from the rule $i' \supseteq i, p' \supseteq p, n' \supseteq n, m' \supseteq m, o' \supseteq o$ hardwired in the specialized binate solver. Therefore with an extended representation one cannot use the specialized binate solver. It is also the case that the larger number of variables of the extended representation will slow down the binate solver. An advantage of an extended representation is that if one would implement column dominance as a maximal operation on columns, restricted column dominance (or better, a strenghtened version of it) would correspond to a maximal operation on columns in extended representation. But our binate solvers implement a more general definition of column dominance, that does not reduce to a maximal operation.

11.3.2 Implicit Computations for Encodeability

Given a set of minterms M corresponding to a FSM (STT) and a selection of GPI's G, one must check if the uniqueness constraints, the face embedding constraints and the encoding constraints induced by the GPI's are satisfiable. If not, one selects one more GPI from the set of unselected GPI's G', with the objective to minimize the number of unsatisfied face constraints.

Figure 11.2 shows computations to check for constraint satisfaction, to select one more GPI to improve satisfiability and to compute a lower bound on the number of GPI's to be added to make the problem feasible. They differ significantly from those proposed in [116] because the fact of using a BDD-base representation has motivated a different formulation of the encodeability check. The encodeability problem is such that the number of encoding constraints is proportional to the number of minterms. The characteristic functions of sets of dichotomies and of encoding constraints are represented implicitly using BDD's. Furthermore, implicit operations can be applied to multiple objects simultaneously. As a result, enumerative processes such as the raising of dichotomies can performed efficiently with the proposed representation.

Each of the following major steps is described in a separate subsection:

- Computation of encoding constraints.
- Computation of free initial dichotomies from face embedding constraints.
- Computation of free initial dichotomies from uniqueness constraints.
- Duplication of free initial dichotomies into pairs of fixed initial dichotomies.
- Iterative raising of initial dichotomies, until they become maximally raised or invalid.

If a problem is infeasible, one disregards the free initial dichotomies and raised dichotomies that have been satisfied and carries on, instead, the following steps on the unsatisfied dichotomies:

- Computation of the set of minimal updating sets of encoding constraints.
- Selection of a branching column (i.e., a GPI in G').
- Computation of a lower bound.

The routine $implicit\_encodeability$ returns $(unsat\_FID, GPI\_selected, lower\_bound)$ to the calling routine. If the given constraints are satisfiable, $implicit\_encodeability$ will return

$unsat\_FID = GPI\_selected = \emptyset$. Otherwise, the calling routine receives a non-empty set of unsatisfied free initial dichotomies $unsat\_FID$; moreover, it can set the lower bound to $lower\_bound$, and then perform branching with the column in $GPI\_selected$.

Encoding Constraints

Each encoding constraint, represented by a set of quadruples (i, p, n, n'), is associated to a minterm denoted by (i, p).[2] The left hand-side of the encoding constraint is a single state n (called the parent) and the right hand-side is a disjunction of conjuncts, so that the right hand-side can be represented by a set of positional sets n' (each element of n' is called a child of the conjunct). In other words, if an n' is related to i, p, n in such a quadruple, n' represents one of the conjuncts on the right hand-side of the encoding constraint.

Given a minterm in the input part, (i, p), the parent n is uniquely determined by $M(i, p, n)$. By definition, each conjunct n' corresponds to a next state tag of a GPI containing that input minterm. Thus the set of encoding constraints can be computed as:

$$encoding\_constraints(i, p, n, n') = M(i, p, n) \cdot \exists i', p' \, [G(i', p', n') \cdot (i \subseteq i') \cdot (p \subseteq p')].$$

These constraints can be further simplified as illustrated by the following example: $a = a + abc$. First, we know that the set $\{a\}$ is contained in the set $\{abc\}$ and thus the latter conjunct is redundant in the right hand-side. Such redundancies can be removed by the $Minimal_{n'}$ operator [66]. The constraint is then simplified to $a = a$ which is trivially satisfiable. Then the trivial constraints can be taken away by the term $(n \neq n')$:

$$constraints(i, p, n, n') = Minimal_{n'}(encoding\_constraints(i, p, n, n')) \cdot (n \neq n').$$

Free Initial Dichotomies from Face Embedding Constraints

Face embedding constraints are state sets of present state literals in the selected GPI's, and can be derived by the following expression: $\exists i', n' \, [G(i', p', n')]$. To generate the *free* initial dichotomy originated from a face embedding constraint, we choose (arbitrarily) the left block, x, of the free initial dichotomy to represent the present state literal of a GPI, and the right block, y, to represent a single state (i.e., $Tuple_1(y)$ is true) not present in the literal (i.e., $(y \not\subseteq x)$). Thus the set of free initial dichotomies originated from face embedding constraints can be computed by:

$$FID_{face}(x, y) \quad = \quad \exists p' \, \{\exists i', n' \, [G(i', p', n')] \cdot (x = p')\} \cdot Tuple_1(y) \cdot (y \not\subseteq x)$$

[2] The relation between encoding constraints and input minterms is, in general, a one-to-many function.

procedure implicit_encodeability(G, G', M) {

$encoding\_constraints(i,p,n,n') = M(i,p,n) \cdot \exists i', p' \, [G(i',p',n') \cdot (i \subseteq i') \cdot (p \subseteq p')]$

$constraints(i,p,n,n') = Minimal_{n'}(encoding\_constraints(i,p,n,n')) \cdot (n \neq n')$

$FID_{face}(x,y) = \exists i', n' \, [G(i',x,n')] \cdot Tuple_1(y) \cdot (y \not\subseteq x) - Tuple_1(x) \cdot Tuple_1(y)$

$FID_{unique}(x,y) = Tuple_1(x) \cdot Tuple_1(y) \cdot (x \succ y)$

$\cdot \; \nexists x', y' \, \{FID_{face}(x',y') \cdot [(x \in x') \cdot (y \in y') + (x \in y') \cdot (y \in x')]\}$

$FID(x,y) = FID_{face}(x,y) + FID_{unique}(x,y)$

$ID(l,r,x,y) = FID(x,y) \cdot [(l = x) \cdot (r = y) + (l = y) \cdot (r = x)]$

$left\_rule(l',r',l,r,i,p) = (r = r') \cdot \exists n \, \{\exists n' \; constraints(i,p,n,n')(n \cup l' = l) \cdot (n \cap l' = \emptyset)$

$\cdot \forall n' \, [constraints(i,p,n,n') \Rightarrow (n' \cap l' \neq \emptyset)]\}$

$right\_rule(l',r',l,r,i,p) = (l = l') \cdot \exists n' \, \{(n' \cup r' = r) \cdot (n' \cap r' \neq n') \cdot \exists n \, [(n \subseteq r') \cdot constraints(i,p,n,n')$

$\cdot (n' \cap l = \emptyset) \cdot \; \forall n'' \, [((n'' \neq n') \cdot constraints(i,p,n,n'')) \Rightarrow (n'' \cap l \neq \emptyset)]]\}$

$rules(l',r',l,r,i,p) = left\_rule(l',r',l,r,i,p) + right\_rule(l',r',l,r,i,p)$

$invalid(l,r) = (l \cap r \neq \emptyset)$

$maximally\_raised(l',r') = \nexists l,r,i,p \; rules(l',r',i,p,l,r)$

/* traverse raising graphs */

$D_{valid}(l,r,x,y) = raising\_graphs(ID(l,r,x,y), rules(l',r',l,r,i,p), invalid(l,r,x,y))$

/* prune satisfied raising graphs */

$unsat\_FID(x,y) = FID(x,y) \cdot \; \nexists l,r \, [D_{valid}(l,r,x,y) \cdot maximally\_raised(l,r)]$

$D_{valid}(l,r,x,y) = D_{valid}(l,r,x,y) \cdot unsat\_FID(x,y)$

/* compute set of min updating sets */

$updating\_sets(l,r,i,p,x,y) = \exists l', r' \, [D_{valid}(l,r,x,y) \cdot rules(l,r,l',r',i,p)]$

$min\_updating\_sets(l,r,i,p,x,y) = Set\_Minimal_{i,p}(updating\_sets(l,r,i,p,x,y))$

/* select branch column */

$min\_outdeg\_node(l,r,x,y) = Multi\_Lmin(min\_updating\_sets(l,r,i,p,x,y),(i,p),(x,y))$

$min\_outdeg\_edges(l,r,i,p,x,y) = min\_outdeg\_node(l,r,x,y) \cdot min\_updating\_sets(l,r,i,p,x,y)$

$T_1(i,p,x,y;i',p',n') = \exists l,r \, [min\_outdeg\_edges(l,r,i,p,x,y) \cdot (n' \cap l = \emptyset)] \cdot G'(i',p',n') \cdot (i' \supseteq i) \cdot (p' \supseteq p)$

$GPI\_selected(i',p',n') = Lmax(T_1,(i,p,x,y))$

/* compute lower bound */

$T_2(x,y;i',p',n') = \exists i,p \, \{\exists l,r \, [min\_updating\_sets(l,r,i,p,x,y) \cdot (n' \cap l = \emptyset)] \cdot G'(i',p',n') \cdot (i' \supseteq i) \cdot (p' \supseteq p)\}$

$lower\_bound = Max\_Indep\_Set(T_2,(x,y),(i',p',n'))$

return $(unsat\_FID, GPI\_selected, lower\_bound)$

}

Figure 11.2: Implicit encodeability computations

$$= \exists i', n' \, [G(i', x, n')] \cdot Tuple_1(y) \cdot (y \not\subseteq x).$$

Free Initial Dichotomies from Uniqueness Constraints

Uniqueness constraints generate initial dichotomies with a singleton state in the x and y blocks (i.e., $Tuple_1(x) \cdot Tuple_1(y)$ is true). We need to generate an initial dichotomy (x, y) if states x and y are not already distinguished by any free initial dichotomy resulting from face embedding constraints. This condition is expressed by: $\nexists x', y' \{FID_{face}(x', y') \cdot [(x \in x') \cdot (y \in y') + (x \in y') \cdot (y \in x')]\}$.

The previous relation generates the set of fixed initial dichotomies related to uniqueness constraints. However for subsequent computations, we need also the set of free initial dichotomies. So we must pick one dichotomy out of each complementary pair of fixed initial dichotomies, and this can be done systematically by the clause $(x \succ y)$. Here we exploit the fact that any positional-set can be represented as a binary number, and we only pick an initial dichotomy (x, y) to be a free initial dichotomy if the binary representation of x is greater than that of y. In summary, the set of free initial dichotomies originated from uniqueness constraints can be computed by:

$$\begin{aligned} FID_{unique}(x, y) &= Tuple_1(x) \cdot Tuple_1(y) \cdot (x \succ y) \\ &\quad \cdot \nexists x', y' \{FID_{face}(x', y') \cdot [(x \in x') \cdot (y \in y') + (x \in y') \cdot (y \in x')]\}. \end{aligned}$$

Now we combine these two sets to form the set of free initial dichotomies as follows:

$$FID(x, y) = FID_{face}(x, y) + FID_{unique}(x, y).$$

Initial Dichotomies

Each free initial dichotomy (x, y) in FID corresponds to two *fixed* dichotomies (x, y), $(y, x) \in ID$. They can be computed as follows:

$$\begin{aligned} ID(l, r) &= \exists x, y \{FID(x, y) \cdot [(l = x) \cdot (r = y) + (l = y) \cdot (r = x)]\} \\ &= FID(l, r) + FID(r, l). \end{aligned}$$

In the algorithm shown in Figure 11.2, each dichotomy (l, r) is actually annotated by the free initial dichotomy (x, y) from which it is originally derived or raised. A *raising graph* is a rooted connected graph. The (x, y) label is useful to distinguish dichotomies in different raising graphs. In other words, the same dichotomy (i.e., same left and right blocks) can be reached starting from

different free initial dichotomies, but the reached dichotomies are treated as different. As a result, raising graphs will not overlap. To obtain the annotated $ID(l, r, x, y)$, the existential quantification over x and y is omitted from the ID computation in Figure 11.2.

Raising Graphs and Implicit Tools for their Traversal

The problem of branch column selection and lower bound computation requires the exploration of different raising actions. The process of raising can be modeled by a forest of raising graphs. Each raising graph has a free initial dichotomy as its root. Its intermediate nodes are non-maximally raised valid dichotomies, while its leaves are either all invalid dichotomies or all maximally raised valid dichotomies. The properties of the leaves have been proved in Section 9.1.1 and will be exploited by our algorithm. The outgoing edges from a dichotomy are labeled by encoding constraints which are applicable to that dichotomy. The edges point to their corresponding raised dichotomies.

The advantage of casting the problem to one of graph traversal is that efficient implicit graph traversal techniques can be employed. As a result, we can perform all the following computations in a single implicit iterative step:

1. manipulate all separate raising graphs simultaneously,
2. for each raising graph, operate on all leaf-dichotomies in it simultaneously,
3. for each raising graph and each leaf-dichotomy in it, test applicability of all encoding constraints and obtain all raised dichotomies simultaneously.

As mentioned before, each node of a raising graph is labeled by a dichotomy (possibly an invalid or a maximally raised valid one). Each edge is labeled by an applicable encoding constraint. Thus each edge can be expressed by a 6-tuple (l', r', i, p, l, r) which is labeled by the input minterm (i, p), originates from the dichotomy (l', r') and is raised (or pointed) to the dichotomy (l, r). Pictorially, we have $(l', r') \xrightarrow{(i,p)} (l, r)$.[3] The set of possible raising edges is represented by the set $rules(l', r', i, p, l, r)$, which represents the rules that raise dichotomies. The set $rules$ consists of the sets $left\_rule$ and $right\_rule$:

$$rules(l', r', l, r, i, p) = left\_rule(l', r', l, r, i, p) + right\_rule(l', r', l, r, i, p).$$

[3]Note that a single encoding constraint can be associated to more than one input minterm. Such a case is correctly modelled by multiple edges between nodes (l', r') and (l, r).

A $left\_rule$ does not modify the right block, but adds a state n *originally absent* from the left block l', $(n \cap l' = \emptyset)$, to form a new left block l, $(n \cup l' = l)$. Thus the raising rules here cannot be applied vacuously (because each rule must add at least one state to one block). In addition, the raising conditions as described in Section 9.1.1 require that at least one child of each conjunct is in the left block:

$$\forall n' \, [constraints(i, p, n, n') \Rightarrow (n' \cap l' \neq \emptyset)].$$

The $left\_rule$ is computed by:

$$\begin{aligned} left\_rule(l', r', l, r, i, p) \;\; = \;\; & (r = r') \cdot \exists n \, \{(n \cup l' = l) \cdot (n \cap l' = \emptyset) \\ & \cdot \forall n' \, [constraints(i, p, n, n') \Rightarrow (n' \cap l' \neq \emptyset)]. \end{aligned}$$

The $right\_rule$ is similarly computed by:

$$\begin{aligned} right\_rule(l', r', l, r, i, p) \;\; = \;\; & (l = l') \cdot \exists n' \, \{(n' \cup r' = r) \cdot (n' \cap r' \neq n') \\ & \cdot \exists n \, [(n \subseteq r') \cdot constraints(i, p, n, n') \cdot (n' \cap l = \emptyset) \\ & \cdot \forall n'' \, [((n'' \neq n') \cdot constraints(i, p, n, n'')) \Rightarrow (n'' \cap l \neq \emptyset)]]\}. \end{aligned}$$

The above computations are not specific to a particular set of dichotomies and thus they can be computed once and for all before the iterative loop.

To test for termination, one checks if a dichotomy is invalid or not. As compared with the explicit algorithm in [116], raising is stopped once an invalid dichotomy is detected by a simpler way of testing invalidity. A dichotomy (l, r) defined to be *invalid* if an element is common to both its left and right blocks $(l \cap r \neq \emptyset)$:

$$invalid(l, r) = (l \cap r \neq \emptyset).$$

A valid dichotomy has been maximally raised if no encoding constraint in $rules$ can be applied to it. The maximality of a dichotomy (l', r') is tested as follows:

$$maximally\_raised(l', r') = \nexists l, r, i, p \; rules(l', r', i, p, l, r).$$

Raising by Implicit Graph Traversal

The raising graphs are traversed in an iterative manner. The goal is to collect the reached dichotomies into two sets: D_{valid} representing the set of valid (partially or maximally) raised dichotomies and $D_{invalid}$ denoting the set of invalid dichotomies. A free initial dichotomy is

unsatisfied if the raising subgraphs[4] rooted at both of its fixed initial dichotomies have all their leaves in $D_{invalid}$. In this case, one wants the GPI that, once added, improves more satisfiability. On the other hand, if any leaf of a raising subgraph is valid *and* maximally raised, one concludes by Theorem 9.1.2 that the free initial dichotomy is satisfiable. In this case, the whole raising graph should be ignored during the computation of a branching column and lower bound.

We start with the set of fixed initial dichotomies ($D_0 = ID$). At the k-th iteration, a current set of dichotomies $D_k(l', r')$ is raised with respect to all applicable $rules$ to give a new set of raised dichotomies $D_{k+1}(l, r)$. The current set of dichotomies is transformed as follows:

$$D_{k+1}(l, r, x, y) = \exists l', r', i, p\ [D_k(l', r', x, y) \cdot rules(l', r', l, r, i, p)].$$

Invalid dichotomies obtained above are then detected and added to the set $D_{invalid}$, and they are removed from the set D_{k+1}. This remaining set D_{k+1} is added to the set of valid dichotomies D_{valid}. These updatings are performed by the following computations:

$$\begin{aligned} D_{invalid}(l, r, x, y) &= D_{invalid}(l, r, x, y) + D_{k+1}(l, r, x, y) \cdot invalid(l, r) \\ D_{k+1}(l, r, x, y) &= D_{k+1}(l, r, x, y) \cdot \neg invalid(l, r) \\ D_{valid}(l, r, x, y) &= D_{valid}(l, r, x, y) + D_{k+1}(l, r, x, y). \end{aligned}$$

The value of k is incremented, and the next iteration is applied again if $D_k \neq \emptyset$. Note that if all dichotomies in $D_k(l', r')$ have been maximally raised, no rules will be applicable to any (l', r') in it, and therefore $D_{k+1}(l, r)$ becomes empty after the k-th iteration. Also if all dichotomies $D_{k+1}(l, r)$ become invalid, the above computations will leave D_{k+1} empty. The iteration will terminate in both cases. A procedure to compute the raising graphs is shown in Fig. 11.3.

Pruning Satisfied Free Initial Dichotomies and their Raising Graphs

As discussed in Section 9.1.1, a free initial dichotomy is satisfied iff it can be maximally raised to a valid dichotomy. In other words, a free initial dichotomy (x, y) is unsatisfied if it cannot be raised to a dichotomy (l, r) that is both valid (i.e., $D_{valid}(l, r, x, y)$) and maximally-raised (i.e., $maximally\_raised(l, r)$). The set of unsatisfied free initial dichotomies can be computed by:

$$unsat\_FID(x, y) = \nexists l, r\ [D_{valid}(l, r, x, y) \cdot maximally\_raised(l, r)].$$

[4]The root of a *raising graph* is a free initial dichotomy in $FID(x, y)$, and has two children which are fixed initial dichotomies in $ID(l, r, x, y)$. In the sequel, the term *raising subgraphs* will be used to refer to the subgraphs rooted at those fixed initial dichotomies.

procedure raising_graphs($ID, rules, invalid$) {

$k = 0$; $D_k(l,r,x,y) = D_{valid}(l,r,x,y) = ID(l,r,x,y)$; $D_{invalid}(l,r,x,y) = \emptyset$

do {

$D_{k+1}(l,r,x,y) = \exists l', r', i, p\,[D_k(l',r',x,y) \cdot rules(l',r',l,r,i,p)]$

$D_{invalid}(l,r,x,y) = D_{invalid}(l,r,x,y) + D_{k+1}(l,r,x,y) \cdot invalid(l,r)$

$D_{k+1}(l,r,x,y) = D_{k+1}(l,r,x,y) \cdot \neg invalid(l,r)$

$D_{valid}(l,r,x,y) = D_{valid}(l,r,x,y) + D_{k+1}(l,r,x,y)$

$k = k + 1$

} until $(D_k(l,r,x,y) = \emptyset)$

return (D_valid)

}

Figure 11.3: Implicit encodeability computations

Once a free initial dichotomy is satisfied, it will remain satisfied even if we add more GPI's to our selection. As a result, there is no reason to traverse the raising graph rooted at each satisfied free initial dichotomy again. To ignore these satisfied raising graphs when computing updating sets in the next section, the dichotomies annotated with $(x,y) \notin unsat\_FID$ are taken away from the set D_{valid}:

$$D_{valid}(l,r,x,y) = D_{valid}(l,r,x,y) \cdot unsat\_FID(x,y).$$

Computing the Set of Minimal Updating Sets

If a free initial dichotomy is removed, we find and update a set of encoding constraints responsible of removing the dichotomy. Such a set of encoding constraints is called an updating set, and it is associated with a particular free initial dichotomy (x,y) (and the raising graph rooted there). As mentioned in Section 9.1.3, each updating set corresponds to a dichotomy node (l,r) in the raising graph, and the updating encoding constraints correspond to the labels of the outgoing edges of that node. We represent the set of updating sets by the relation $updating\_sets(l,r,i,p,x,y)$: an encoding constraint denoted by input minterm (i,p) is in an updating set associated with dichotomy (l,r) within the raising graph rooted at (x,y) iff the 6-tuple (l,r,i,p,x,y) is in the $updating\_sets$ relation. The (l,r) label is kept because it will be used later. The set of all updating sets can be obtained implicitly as shown below, by considering all annotated valid dichotomies and identifying

all applicable encoding constraints (via $rules$) from each of these valid dichotomies:

$$updating\_sets(l, r, i, p, x, y) = \exists l', r' \, [D_{valid}(l, r, x, y) \cdot rules(l, r, l', r', i, p)].$$

In the subsequent computations, only a subset of minimal $updating\_sets$, called $min\_updating\_sets$, matters. An updating set is in $min\_updating\_sets$ if no encoding constraint can be removed from it, while the set still remains an updating set. The set of all minimal updating sets can be computed by identifying nodes (l, r) whose sets of outgoing edge labels (i, p) are not subsets of other updating sets:

$$min\_updating\_sets(l, r, i, p, x, y) = Set\_Minimal_{i,p}(updating\_sets(l, r, i, p, x, y)).$$

Branching Column Selection

As the existing selection of GPI's does not satisfy all free initial dichotomies (if $unsat\_FID \neq \emptyset$), at least one more GPI must be selected. The objective of GPI (branching column) selection is to maximally improve the overall satisfiability of the unsatisfied free initial dichotomies. The addition of a GPI will update a number of encoding constraints, and therefore will improve (or at least not worsen) the satisfiability of $unsat\_FID$. To select such a GPI optimally, we must use the set of all updating sets of encoding constraints ($updating\_set$) to construct a *full satisfiability table*. Here heuristically, we build a *simplified partial satisfiability table* [5] instead.

For each unsatisfied free initial dichotomy (x, y), we find an updating set with the minimum number of encoding constraints, i.e., a *minimum cardinality updating set*. Because any GPI selection that updates these constraints may satisfy the given free initial dichotomy, one hopes that by updating constraints in a minimum cardinality updating set, a small number of GPI's will suffice to find an encodeable cover. A minimum cardinality updating set corresponds to the minimum out-degree node in the raising graph.

The minimum out-degree node (l, r, x, y) in the raising graph rooted at (x, y) can be extracted by the $Multi\_Lmin$ operator on the set of minimal updating sets:

$$min\_outdeg\_node(l, r, x, y) = Multi\_Lmin(min\_updating\_sets(l, r, i, p, x, y), (i, p), (x, y)).$$

The edges (i, p) associated with each minimum cardinality updating set are obtained by:

$$\begin{aligned} min\_outdeg\_edges(l, r, i, p, x, y) \quad &= \quad min\_outdeg\_node(l, r, x, y) \\ &\quad \cdot min\_updating\_sets(l, r, i, p, x, y). \end{aligned}$$

[5] With respect to the *partial satisfiability table* presented in Section 9.1.4, this table is simplified, because each updating clause has exactly one literal, and not two.

The columns of the simplified partial satisfiability table, T_1, are labeled by the unselected GPI's $G'(i', p', n')$. The rows of table T_1 are divided into sections corresponding to different unsatisfied free initial dichotomies. Thus a part of the row label is (x, y) to distinguish the sections. Within a section, a row is also labeled by (i, p) corresponding to an encoding constraint in the minimum cardinality updating set (i.e., $(i, p) \in min\_outdeg\_edges$). A table entry $(i, p, x, y; i', p', n')$ is a 1-entry iff the input part of the GPI covers the input minterm of the encoding constraint (i.e., $(i' \supseteq i) \cdot (p' \supseteq p)$) and no child of the conjunct n' is in the left block, $(n' \cap l = \emptyset)$. The implicit table is obtained by the following computation:

$$\begin{aligned} T_1(i, p, x, y; i', p', n') \quad = \quad & \exists l, r \, [min\_outdeg\_edges(l, r, i, p, x, y) \cdot (n' \cap l = \emptyset)] \\ & \cdot G'(i', p', n') \cdot (i' \supseteq i) \cdot (p' \supseteq p). \end{aligned}$$

To select a GPI to improve the overall satisfiability of $unsat\_FID$, we select a column in table T_1 that contains the maximum number of 1's. The $Lmax$ operator is used to pick such a column as follows:

$$GPI\_selected(i', p', n') = Lmax(T_1, (i, p, x, y)).$$

Lower Bound Computation

For reasons described in Section 9.1.5, we cannot use the simplified partial satisfiability table T_1 for lower bound computation. Instead, we construct the *support satisfiability table,* T_2. We still start with the set of minimal updating sets. The rows are now labeled only by $(x, y) \in unsat\_FID$. Each row represents an *or* clause of the encoding constraints in all $min\_updating\_sets$ associated with (x, y). The 1-entries in table T_2 are obtained as those in T_1, except that here all edges in the support are used instead of only those in min_outdeg_edges, and the whole right-hand expression is existentially quantified by i, p because each clause represents an *or* of all encoding constraints in the support. Table T_2 is computed as follows:

$$\begin{aligned} T_2(x, y; i', p', n') \quad = \quad & \exists i, p \, \{\exists l, r \, [min\_updating\_sets(l, r, i, p, x, y) \cdot (n' \cap l = \emptyset)] \\ & \cdot G'(i', p', n') \cdot (i' \supseteq i) \cdot (p' \supseteq p)\}. \end{aligned}$$

A lower bound on the number of additional GPI's to make the problem satisfiable can be found by computing the maximal independent set of rows in table T_2, by means of the Max_Indep_Set operator [66] as follows:

$$lower\_bound = Max\_Indep\_Set(T_2, (x, y), (i', p', n')).$$

procedure codes_implicit_gpi_selection($D_{valid}, maximally\_raised, FID$) {

/* find valid maximally raised dichotomies */

$D_{start}(l,r) = \exists x,y D_{valid}(l,r,x,y) \cdot maximally\_raised(l,r)$

/* complete valid maximally raised dichotomies */

$D_{complete}(l,r) = \exists l',r'\{D_{start}(l',r')(l \supseteq l')(r \supseteq r')(l \cdot r = \emptyset)\ \nexists x[Tuple_1(x)(l \not\supseteq x)(r \not\supseteq x)]\}$

/* remove invalid dichotomies */

$D_{valid}(l,r) = D_{complete}(l,r) - \exists l'r'ip[rules(l,r,l',r',i,p) \cdot invalid(l',r')]$

/* select a minimum set of valid complete dichotomies that cover the FID's */

$D_{columns}(l,r) = unate\_encoding(D_{valid}, FID)$

}

Figure 11.4: Computation of codes satisfying a selection of GPI's

11.3.3 Implicit Encoding of an Encodeable Set of GPI's

In this section we describe the generation of codes that satisfy an encodeable set of GPI's. The cost function is the number of encoding bits. The problem is to generate valid complete dichotomies and then set up and solve a unate covering problem.

Figure 11.4 shows an exact implicit algorithm to find codes of minimum length that satisfy a given set of encoding constraints (in this case already known to be encodeable), based on the notion of completion of a dichotomy. The algorithm computes the completion $D_{complete}(l,r)$ of the set $D_{start}(l,r)$ of valid maximally raised dichotomies. Then it removes from $D_{complete}(l,r)$ the invalid dichotomies, i.e., the dichotomies that could be raised again. Since the dichotomies in $D_{complete}(l,r)$ are complete, if raising is still possible, it must introduce some invalidity. By Theorem 9.1.3 this procedure finds a minimum set of encoding columns.

The last step solves a table covering problem. The rows of the table are the free initial dichotomies and the columns are the valid complete dichotomies. If a valid complete dichotomy covers one of the two initial encoding dichotomies associated to a free initial dichotomy (itself and the one with the two blocks exchanged) [6], then there is a 1 at the intersection of the valid complete dichotomy and the free initial dichotomy. The table is unate, i.e. either an entry is 1 or it is empty. The implicit general binate solver previously mentioned is used here.

The general binate solver requires the sets of columns and rows and a rule to compute a

[6] In other words, the left and right blocks of the free initial dichotomy are subsets respectively either of the left and right blocks, or of the right and left blocks of the valid complete dichotomy.

table entry. In this case they are:

1. Columns are $C(q) = D_{valid}(l, r)$, where $q = l, r$.
2. Rows are $R(d) = R_u(d) = FID(x, y)$, where $d = x, y$.
3. The table entry at the intersection of the column labelled by $(l, r) \in C$ and of the row labelled by $(x, y) \in R$ is 1 iff $l \supseteq x, r \supseteq y$ or $l \supseteq y, r \supseteq x$.
4. The table entry at the intersection of the column labelled by $(l, r) \in C$ and of the row labelled by $(x, y) \in R$ is never 0.

As a result a set of valid complete dichotomies $D_{columns}(l, r)$ is selected. The columns in $D_{columns}$ are a minimum cover of all the rows.

11.3.4 Approximate Implicit Selection of an Encodeable Cover of GPI's

Fig. 11.5 shows a detailed description of an approximate implicit algorithm to find a selection of GPI's that is a cover of the original FSM and that is encodeable. A simplified view of the algorithm was already shown in Fig. 8.8 and related issues commented. The computations introduced in Section 11.3.2 are used to check encodeability and select a branching column. One minor efficiency improvement is the addition of a set $acc\_sat\_FID(x, y)$ to accumulate the free initial dichotomies (x, y) already shown to be satisfied, because by Theorem 9.1.1, they will stay satisfied when adding more GPI's to the solution. Notice also that a FID (x, y) already verified could be generated again by a newly selected GPI. So when we recompute the FID's generated by the augmented set of GPI's, we check that none of them has been found satisfiable already. To update the set $acc\_sat\_FID(x, y)$, at each iteration one adds to it $sat\_FID(x, y)$, the set of the FID's (x, y) found satisfied in the current iteration. There are various efficiency issues regarding partially duplicated computations in the $while$ loop. We consider them an implementative detail, not to be discussed here. An implementation of this implicit approximate algorithm will be reported next.

11.4 A Worked Example

We show the main steps of the algorithm presented in 11.3.4 on the FSM *leoncino*. The first call to the implicit binate solver returns the following cover of GPI's [7]:

[7] The numbers within () identify them in the lists of GPI's and covering tables given in Section 8.1.

procedure approx_implicit_gpi_selection(P, M) {

$G(i', p', n', o') = unate\_encoding(P, M)$;

$G'(i', p', n', o') = P(i', p', n', o') - G(i', p', n', o')$; $unsat\_FID(x, y) = 1$; $acc\_sat\_FID(x, y) = \emptyset$

while $(unsat\_FID(x, y) \neq \emptyset)$ {

$FID_{face}(x, y) = \exists i', n' \, [G(i', x, n')] \cdot Tuple_1(y) \cdot (y \not\subseteq x) - Tuple_1(x) \cdot Tuple_1(y)$

$FID_{unique}(x, y) = Tuple_1(x) \cdot Tuple_1(y) \cdot (x \succ y)$

$\cdot \; \nexists x', y' \, \{FID_{face}(x', y') \cdot [(x \subseteq x') \cdot (y \subseteq y') + (x \subseteq y') \cdot (y \subseteq x')]\}$

$FID(x, y) = FID_{face}(x, y) + FID_{unique}(x, y)$; $FID(x, y) = FID(x, y) - acc\_sat\_FID(x, y)$

$ID(l, r, x, y) = FID(x, y) \cdot [(l = x) \cdot (r = y) + (l = y) \cdot (r = x)]$

$encoding\_constraints(i, p, n, n') = M_n(i, p, n) \cdot \exists i', p' \, [G(i', p', n') \cdot (i \subseteq i') \cdot (p \subseteq p')]$

$constraints(i, p, n, n') = Minimal_{n'}(encoding\_constraints(i, p, n, n')) \cdot (n \neq n')$

$left\_rule(l', r', l, r, i, p) = (r = r') \cdot \exists n \, \{\exists n' \, constraints(i, p, n, n')(n \cup l' = l) \cdot (n \cap l' = \emptyset)$

$\cdot \; \forall n' \, [constraints(i, p, n, n') \Rightarrow (n' \cap l' \neq \emptyset)]\}$

$right\_rule(l', r', l, r, i, p) = (l = l') \cdot \exists n' \, \{(n' \cup r' = r) \cdot (n' \cap r' \neq n') \cdot \exists n \, [(n \subseteq r') \cdot constraints(i, p, n, n')$

$\cdot (n' \cap l = \emptyset) \cdot \; \forall n'' \, [((n'' \neq n') \cdot constraints(i, p, n, n'')) \Rightarrow (n'' \cap l \neq \emptyset)]]\}$

$rules(l', r', l, r, i, p) = left\_rule(l', r', l, r, i, p) + right\_rule(l', r', l, r, i, p)$

$invalid(l, r) = (l \cap r \neq \emptyset)$; $maximally\_raised(l', r') = \nexists l, r, i, p \, rules(l', r', i, p, l, r)$

/* traverse raising graphs */

$D_{valid}(l, r, x, y) = raising\_graphs(ID(l, r, x, y), rules(l', r', l, r, i, p), invalid(l, r, x, y))$

/* prune satisfied raising graphs */

$unsat\_FID(x, y) = FID(x, y) \cdot \; \nexists l, r \, [D_{valid}(l, r, x, y) \cdot maximally\_raised(l, r)]$

$sat\_FID(x, y) = FID(x, y) - unsat\_FID(x, y)$; $acc\_sat\_FID(x, y) = acc\_sat\_FID(x, y) + sat\_FID(x, y)$

$D_{unsat\_valid}(l, r, x, y) = D_{valid}(l, r, x, y) \cdot unsat\_FID(x, y)$

/* compute set of min updating sets and select branching column */

$updating\_sets(l, r, i, p, x, y) = \exists l', r' \, [D_{unsat\_valid}(l, r, x, y) \cdot rules(l, r, l', r', i, p)]$

$min\_updating\_sets(l, r, i, p, x, y) = Set\_Minimal_{i,p}(updating\_sets(l, r, i, p, x, y))$

$min\_outdeg\_node(l, r, x, y) = Multi\_Lmin(min\_updating\_sets(l, r, i, p, x, y), (i, p), (x, y))$

$min\_outdeg\_edges(l, r, i, p, x, y) = min\_outdeg\_node(l, r, x, y) \cdot min\_updating\_sets(l, r, i, p, x, y)$

$T_1(i, p, x, y; i', p', n') = \exists l, r \, [min\_outdeg\_edges(l, r, i, p, x, y) \cdot (n' \cap l = \emptyset)] \cdot G'(i', p', n') \cdot (i' \supseteq i) \cdot (p' \supseteq p)$

$GPI\_selected(i', p', n') = Lmax(T_1, (i, p, x, y))$

$G(i', p', n') = G(i', p', n') + GPI\_selected(i', p', n')$; $G'(i', p', n') = G'(i', p', n') - GPI\_selected(i', p', n')$

}

return$(P(i' p' n', o') \cdot G(i', p', n'))$

}

Figure 11.5: Approximate implicit selection of GPI's - Detailed view

$1 - 00100111\ (3), 0111101011\ (5), -111011000\ (24), -010010001\ (6), -001101110\ (17).$

The next-state constraints are:

$m_2\ \ st0 = st0$

$m_4\ \ st0 = st0$

$m_5\ \ st0 = st0.st1$

$m_6\ \ st1 = st1 + st0.st1$

$m_8\ \ st1 = st1.st2$

$m_{10}\ st1 = st1 + st0.st1$

$m_{11}\ st0 = st0.st1$

$m_{13}\ st2 = st1.st2$

$m_{16}\ st2 = st2 + st1.st2$

$m_{19}\ st2 = st2$

$m_{21}\ st1 = st1.st2$

Trivial next-state constraints are $m_5, m_8, m_{11}, m_{13}, m_{21}$.

The non-trivial face constraints are $(st0, st1)$ and $(st1, st2)$. The free initial dichotomies are $(st0, st1; st2)$ and $(st1, st2; st0)$. The initial dichotomies are $(st0, st1; st2)$, $(st2; st0, st1)$, $(st1, st2; st0)$ and $(st0; st1, st2)$.

There are two raising graphs, one rooted at $(st0, st1; st2)$ and the other rooted at $(st1, st2; st0)$. The edges of the raising graph rooted at $(st0, st1; st2)$ are:

$(st0, st1; st2) \longrightarrow (st0, st1; st2)$,

$(st0, st1; st2) \longrightarrow (st2; st0, st1)$,

$(st0, st1; st2) \xrightarrow{m_{13}} (st2, st0, st1; st2)$,

$(st2; st0, st1) \xrightarrow{m_8} (st1, st2; st0, st1)$,

$(st2; st0, st1) \xrightarrow{m_{21}} (st1, st2; st0, st1)$.

All maximally raised dichotomies (sinks of the graph), i.e., the nodes $(st2, st0, st1; st2)$ and $(st1, st2; st0, st1)$, are invalid, so the FID $(st0, st1; st2)$ is violated.

The edges of the raising graph rooted at $(st1, st2; st0)$ are:

$(st1, st2; st0) \longrightarrow (st1, st2; st0)$,

$(st1, st2; st0) \longrightarrow (st0; st1, st2)$,

$(st1, st2; st0) \xrightarrow{m_5} (st0, st1, st2; st0)$,

$(st1, st2; st0) \xrightarrow{m_{11}} (st0, st1, st2; st0)$.

While sink $(st0, st1, st2; st0)$ is invalid, sink $(st0; st1, st2)$ is valid, so the FID $(st1, st2; st0)$ is not violated. In this example, all raising actions happen to be due to the left rule.

Since the FID $(st0, st1; st2)$ is violated, the given selection of GPI's is not encodeable. A new GPI is added to it, returned by $Lmax$: 1001100110 (16). The non-trivial next-state constraints are the same as before, except the one corresponding to m_{13} that is updated to m_{13} $st2 = st1.st2 + st2$ becoming a trivial next-state constraint. If we repeat the process of building the raising graphs, we obtain the same graphs as before except that the edge $(st0, st1; st2) \xrightarrow{m_{13}} (st2, st0, st1; st2)$ will be missing, because m_{13} cannot force anymore raisings. Therefore also the FID $(st0, st1; st2)$ is not anymore violated, because it has a valid sink, i.e., $(st0, st1; st2)$. So an encodeable cover of 6 GPI's has been obtained.

11.5 Verification of Correctness

After obtaining an encodeable cover of GPI's and codes that satisfy the constraints, one replaces the codes in the GPI cover and minimizes it to get a minimized encoded GPI cover, F_{min_gpi}. It is useful also to replace the codes in the original FSM cover and then to minimize it, getting F_{min_fsm}. Since the don't care set can be used differently, the two minimized covers may differ and the smallest one is picked.

It is also important to verify that the minimized encoded GPI cover, F_{min_gpi}, is still a cover of the onset of the original FSM. This can be achieved by checking that F_{min_gpi} is contained in the union of the onset and dcset of the encoded (not minimized!) FSM cover and that the onset of the encoded (not minimized!) FSM cover is contained in the union of F_{min_gpi} and the dcset of the encoded FSM cover. If this check is routinely successful one is confident that the algorithm has been implemented correctly. This check is always performed at the end of our program.

Figure 11.6 shows the operations to encode and verify the correctness. F, D and R denote respectively onset, dcset and offset.

We demonstrate the procedure on the example previously utilized to explain the algorithm. The set of selected GPI's, G, is:

```
1- 001 001 11
10 011 001 10
01 111 010 11
-1 110 110 00
-0 100 100 01
-0 011 011 10
```

The codes are: $enc(st0) = 00$, $enc(st1) = 10$, $enc(st2) = 11$. By encoding the GPI cover, one obtains the covers F_{gpi} and R_{gpi}:

procedure code_and_verify($G, D_{columns}, FSM$) {

/* encode the GPI cover */

$F_{gpi} = encode\_gpi(G, D_{columns}, FSM)$

/* minimize encoded GPI cover */

$D_{gpi} = \emptyset(cover)$

$R_{gpi} = complement(F_{gpi})$

$F_{min\_gpi} = espresso(F_{gpi}, D_{gpi}, R_{gpi})$

/* encode the FSM cover */

$(F_{fsm}, R_{fsm}) = encode\_fsm(D_{columns}, FSM)$

$D_{fsm} = complement(F_{fsm} \cup R_{fsm})$

/* verify correctness */

if ($F_{min\_gpi} \subseteq F_{fsm} \cup D_{fsm}$ and $F_{fsm} \subseteq F_{min\_gpi} \cup D_{fsm}$) {

/* minimize encoded FSM cover */

$F_{min\_fsm} = espresso(F_{fsm}, D_{fsm}, R_{fsm})$

return(F_{min_gpi}, F_{min_fsm})

} else return("error")

}

Figure 11.6: Computation of minimized encoded covers and correctness check

```
1-11 1111       -00- 1110
101- 1110       0--- 0100
01-- 1011       11-0 1111
-1-0 0000       1-0- 1100
-000 0001       -001 1111
-01- 1010       001- 0001
                -010 0001
                110- 1111
```

The minimized encoded GPI cover, F_{min_gpi}, is:

```
-000 0001
101- 0100
01-- 1011
-01- 1010
1-11 1111
```

By encoding the FSM cover, one obtains the covers F_{fsm}, R_{fsm} and D_{fsm}:

```
-000 0001       -000 1110       --01 1111
0100 1000       1100 1111       01-1 1111
0-10 1010       0100 0100       010- 0011
1010 1110       0-10 0100       0-10 0001
1-11 1111       1110 1110       -110 0001
0011 1010       1010 0001
                0011 0101
```

The minimized encoded FSM cover, F_{min_fsm}, is:

```
101- 0100
-00- 0001
01-- 1010
-01- 1010
1--1 1111
```

11.6 Implementation Issues

11.6.1 Order of BDD Variables

The ordering of the BDD variables is one of the most excruciating problems encountered while implementing BDD-based computations. Four arrays of variables are needed: A_0, A_1, A_2, A_3, where in turn each array is composed of five subarrays of variables: I, P, N, M, O. I is an array of input variables, P, N, and M are each an array of state variables and O is an array of output variables. Consider an example with 1 input, 1 output and 3 states; A_0 will consist of:

```
i  p1 p2 p3  n1 n2 n3  m1 m2 m3  o
I      P         N         M     O
```

In the computation of prime compatibles only arrays A_0 and A_1 are used. In the solution of the first covering table all four of them are used. It is imperative that the variables in A_0, A_1, A_2, A_3 be interleaved, in order to have linear-sized BDD representations of various key intermediate computations both when computing the primes and solving the first covering table.

We show a compatible order for two arrays of variables A_0 and A_1. Unprimed variables are those in A_0 and primed ($'$) are those in A_1:

```
0 1  2  3   4  5   6  7   8  9   10 11  12 13  14 15  16 17  18 19
i i′ p1 p1′ p2 p2′ p3 p3′ n1 n1′ n2 n2′ n3 n3′ m1 m1′ m2 m2′ m3 m3′
20 21
 o o′
```

Notice that within each array of the type A there is freedom of ordering the variables in I, P, N, M, O. We refer to this ordering as **single interleaving**. When primes are computed, we keep enabled the dynamic reordering routine available in the CMU BDD package.

But it is also necessary that the variables in the arrays P, N and M are interleaved, in order to have linear-sized BDD representations of various key intermediate computations in the encodeability step and when solving the second covering table. We show an order compatible with both requirements for two arrays of variables A_0 and A_1. Unprimed variables are those in A_0 and primed ($'$) are those in A_1:

```
0 1  2  3   4  5   6  7   8  9   10 11  12 13  14 15  16 17  18 19
i i′ p1 p1′ n1 n1′ m1 m1′ p2 p2′ n2 n2′ m2 m2′ p3 p3′ n3 n3′ m3 m3′
20 21
 o o′
```

This order insures both:

1. interleaving between the variables in A_0 and A_1; and
2. interleaving between the variables in P, N, M within array A_0 and within array A_1.

Notice that within each array of the type P or N or M there is freedom of ordering the variables. There is also freedom in ordering I and O with respect to P, N, M. We refer to this ordering as **double interleaving**. Double interleaving is required only for the encodeability computations and the second covering table. We have implemented two variants of double interleaving. In both cases one starts with single interleaving, then in the first variant one switches to double interleaving before invoking the table solver (on the first covering table), while in the second variant one switches to

double interleaving after invoking the table solver (on the first covering table). Again dynamic reordering is allowed during the computation of primes. The second variant is to be preferred because it constrains less the ordering when solving the first covering table, and the experiments confirm it. The ordering with only single interleaving, instead, is not recommended because it is often unable to pass successfully through the second covering table solver.

Dynamic reordering has not been applied yet to the computations in the encodeability step. It will be interesting to find out whether some hard computations in this part can be sped-up by reordering. One must pay attention to the fact that the computations that use the *line_count* primitive BDD operator must be carried on with dynamic reordering disabled.

11.6.2 Computation of Set_Minimal

In the encodeability step it is necessary to compute the following relation:

$$Set\_Minimal_b(F(a,b)) = F(a,b) \cdot \nexists c\{\exists d\, F(c,d) \cdot \forall d\, [F(c,d) \Rightarrow F(a,d)] \cdot \exists d\, [\neg F(c,d) \cdot F(a,d)]\}.$$

It turns out that this is a difficult operation with BDD's even when implemented with the BDD and-smooth operator by rewriting it as:

$$Set\_Minimal_b(F(a,b)) = F(a,b) \cdot \nexists c\{\exists d\, F(c,d) \cdot \nexists d\, [F(c,d) \cdot \neg F(a,d)] \cdot \exists d\, [\neg F(c,d) \cdot F(a,d)]\}.$$

A solution is to approximate the computation using the following logical validities:

$$[\exists d\, F(c,d) \Rightarrow \forall d\, F(a,d)] \Rightarrow \forall d\, [F(c,d) \Rightarrow F(a,d)],$$

and

$$[\exists d\, \neg F(c,d) \cdot \forall d\, F(a,d)] \Rightarrow \exists d\, [\neg F(c,d) \cdot F(a,d)].$$

If we replace in the computation of $Set\_Minimal_b(F(a,b))$ the right-hand sides with the left-hand sides of the previous logic validities we obtain a superset of $Set\_Minimal_b(F(a,b))$, which is a conservative approximation.

11.6.3 The Filtering Heuristic

After a cover of GPI's is returned from the first table covering step, more GPI's are added one at a time to make it encodeable. An alternative is to add to the cover a set of GPI's guaranteed to make it encodeable, find codes that satisfy all of them and then let the final minimization step choose a minimal cover of encoded GPI's. The set of GPI's that we add contains, out of all unselected GPI's,

those with full or singleton present state literal or with a present state literal already occurring in a GPI of the cover. Also the generalized implicants of the original cover are added, to guarantee that at least one encodeable cover can be found. A motivation of this choice is to avoid the introduction of GPI's that add new initial dichotomies, making encodeability temporarily harder to satisfy.

This heuristic is a preliminary attempt in an interesting direction to improve on the present strategy of adding greedily one more GPI at a time. When this heuristic is active we stop at the first solution of the second covering table. The reason is that since the encodeability problem is less constrained one gets more primes dichotomies and therefore the second covering table is not relatively simple as it is often otherwise. In particular it is an experimental fact that these tables generate a lot of branching activities not adequately controlled by the bounding mechanism, so that suboptimal regions of the solution space are explored in depth before being recognized as suboptimal.

11.7 Experiments

We have implemented a program ISA, an acronym for implicit state assignment, that computes the set of GPI's or a subset of them and then implements the procedure for approximate implicit selection of an encodeable cover of GPI's described in Section 11.3.4. The program capitalizes on different existing software technologies. It is built on top of ESPRESSO, to exploit the logic optimization capabilities of the latter in the two-level domain. Two-level logic optimization capabilities are needed at the beginning to do pre-processing (reading a symbolic FSM cover, building its onset, don't care set and offset, computing a cover of the companion function), and at the end to do post-processing (replacing the codes in the encodeable set of GPI's and in the original FSM cover and minimizing them - with an appropriate don't care set - to measure the quality of the final result). The program ISA computes the primes of the companion function (from which GPI's are obtained after a reduction process) using routines kindly provided by G.M.Swamy from her two-level logic minimizer [53]. Then ISA selects a cover of GPI's calling the implicit table solver described in [66]. As a next step, we have implemented the computations shown in Figure 11.5 to obtain a minimal encodeable cover of GPI's.

The core computations are based on the representation of the characteristic functions of relations by means of BDD's. The program can use both the UCB and the CMU BDD packages through the BDD interface developed at UCB. All reported experiments have been done linking the CMU package.

11.7.1 Analysis of the Experiments

We report here a set of experiments to demonstrate the status of the current implementation, which is still in a development phase. GPI's can model an host of encoding problems targetting two-level implementations. Here we have used FSM's as a test case, because they exhibit the most general formulation of encodeability and so they test fully the theory. Other applications can be handled by simple modifications. All run times are reported in CPU seconds on a DEC DS5900/260 with 440 Mb of memory, unless otherwise stated.

The objective of the current implementation has not been to compete with existing state assignment programs like NOVA [147] that have been heavily optimized, but to show that implicit techniques are mature enough to generate and select encodeable sets of GPI's. While up to now it has not been practical to manipulate sets of GPI's because they are very large even for small symbolic covers, our contribution shows that large sets of GPI's for non-trivial examples can be manipulated with implicit techniques. Improvements to the implicit algorithms can extend the frontier of the problems that can be handled.

An open issue left for future investigations is how to use effectively this capability in order to do state assignment or other types of encoding. An exact algorithm that explores all possible subsets of GPI's to find a minimum encodeable one is hardly practical, so one must introduce heuristic restrictions in the search of the solution space. We have used the simplest possible strategy of adding one more GPI at a time (chosen to maximize a cost function measuring the lack of encodeability of the current cover), and then of stopping at this first solution. In order to produce an high-quality result (measured by the size of the minimized encoded cover) this greedy strategy must be replaced by one with a limited amount of backtracking to explore increasingly smaller sets of encodeable covers of GPI's. Here it would help the implicit lower bound criterion presented in Section 11.3.2, currently not used in ISA.

Tables 11.1 and 11.2 report the results of generating GPI's for FSM's of the MCNC benchmark and other examples. We have included FSM's with up to around 30 states, that is the size that can be currently handled. We report the number of primes of the companion function and the number of GPI's. Comparisons of run times to generate the primes of the companion function *only* are made with ESPRESSO [11]. Both programs were timed out at 7200 seconds of CPU time. Notice that we report also the number of variables of the companion function (given by $2 * i + 3 * p + o$, where i, p, o are respectively the number of inputs, states and outputs of the FSM), because it is a more indicative measure (than the number of states) of the the complexity of

the computation to generate the GPI's.

Tables 11.3 and 11.4 report the results of running ISA to select a minimal encodeable cover of GPI's. For these experiments ISA has been run with option $-m$, that computes a subset of the GPI's, by applying the minimal transformation, instead of the maximal transformation that gives all GPI's (see Section 7.4.4 for a definition of minimal and maximal transformations). The reason is that smaller tables are generated, to the price of a solution of lesser quality. The tables provide the following information:

- Under the column "table size" we provide the dimensions of the original table and of its cyclic core, i.e., the dimensions of the table obtained when the first cycle of reductions converges.
- "mincov calls" is the number of recursive calls of the implicit table solver.
- The column "table sol." is the cardinality of the cover of GPI's returned by the table solver.
- The column "final sol." is the cardinality of the final encodeable cover of GPI's.
- CPU time gives the time for the first call to the table solver under the column "table red.". Under the column "total" there is the total time of ISA on the example, inclusive of the time to compute the primes, get a cover of GPI's by calling the implicit table solver, find an encodeable cover of GPI's and get the codes by another call to the implicit table solver. Since the latter call is usually on a small table, it is lumped with the rest.

Out of the examples in Table 11.3, ISA fails to complete the first table reduction of *slave* because of timeout at 18000 seconds, during collapse columns. Ouf of the examples in Table 11.4, ISA fails to complete some of them, again due to timeout or no more memory in the collapse column step of the first table reduction. The runs of *ex2, maincont, saucier* did not complete because of timeouts during the selection of new GPI's: the time-consuming operations there are *i_set_minimum* (which can be successfully approximated as seen in Section 11.6.2) and changes of BDD variables support necessary for the *multi_lmin* computation. Causes of failure are described more precisely in the tables. The results reported for *cse, dk512, keyb* were obtained with option *-q* (heuristic of Section 11.6.3), and those for *ex2, maincont, pkheader* with option *-k* (approximation to $Set\_Minimal$ in Section 11.6.2). FSM's *cse, dk512, keyb, ex2, maincont, pkheader, mark1* were run on a DEC 7000 Model 610 AXP with 1Gb of memory.

Tables 11.5 and 11.6 report the cover size of the encoded and minimized covers produced by ISA and compare them with the best results of NOVA. The tables provide the following information:

FSM	states	# vars. compan.fn.	primes	GPI's	CPU time (sec) ISA	ESPRESSO
bbara	10	40	14760	13518	9	532
bbtas	6	24	252	230	0	0
beecount	7	31	1834	959	4	1
chanstb	4	44	619	571	8	0
cpab	5	49	3509	2841	44	17
dk14	7	32	2850	1228	3	2
dk15	4	23	231	143	0	0
dk17	8	31	2021	1575	2	2
dk27	7	25	377	296	0	0
dol2	5	20	194	170	0	0
es	4	18	101	80	0	0
ex3	10	36	8686	8125	7	181
ex5	9	33	4232	3741	3	20
ex6	8	42	5720	3495	12	26
ex7	10	36	8538	7931	6	147
fstate	8	45	5949	5231	14	23
leoncino	3	15	39	26	0	0
lion	4	17	79	51	0	0
lion9	9	32	2122	1136	3	7
mc	4	23	94	77	0	0
ofsync	4	28	185	155	1	0
opus	10	46	16735	15934	23	329
s8	5	24	326	316	0	0
scud	8	44	43602	30259	74	2026
shiftreg	8	27	764	527	0	0
slave	10	91	273027	228463	147	7135
tav	4	24	81	81	0	0
test	2	10	5	5	0	0
virmach	4	44	257	216	11	0

Table 11.1: GPI's of small examples from the MCNC benchmark and others.

FSM	states	transf.	primes	GPI's	CPU time (sec) ISA	ESPRESSO
bbsse	16	53	1493485	1399079	136	1286
cf	13	69	2206595	2134887	178	-
cse	16	69	2335927	1832229	109	-
dk512	15	50	98238	91947	11	-
ex1	20	97	149755546	146394042	336	-
ex2	19	63	4640888	4597063	151	-
ex4	14	63	120835	120721	29	-
keyb	19	73	28592198	27327259	212	-
kirkman	16	78	2106843	2106783	252	-
maincont	16	74	1484786	1418800	37	-
mark1	15	71	733697	728799	89	-
master	15	122	269304493	264757774	5630	-
modulo12	12	39	12282	11961	4	5246
pkheader	16	85	229946	229726	823	-
ricks	13	82	120576	119488	80	-
s1	20	82	-	-	-$^{(a)}$	-
s1a	20	82	693626434	616527717	3902	-
saucier	20	87	111895231	111852040	126	-
tma	20	80	12324742	12118857	3711	-
train11	11	38	6444	4856	11	207
donfile	24	77	150994935	64959680	2348	-
dk16	27	88	1207950375	1179949953	3775	-
pma	24	96	1267371428	1248519820	2671	-
rpss	22	115	1229747382	1226813350	536	-
tr4	22	105	2770731006	2769352444	138	-

$^{(a)}$ out of memory

all runs timed out 7200 seconds

Table 11.2: GPI's of medium examples from the MCNC benchmark and others.

- The column "FSM cover" gives the cardinality of the original FSM cover.
- The column "1-hot cover" gives the cardinality of the FSM cover, after 1-hot encoding and minimization.
- Under "results of ISA", the column "min.gpi" gives the cardinality of the encoded and minimized cover of GPI's, while the column "min.FSM" gives the cardinality of the encoded and minimized initial cover. In both cases the codes used are those returned by the second call to the table solver, which satisfy the encoding constraints. The column "bits" returns the length of the codes, that is the cardinality of the solution of the second call to the table solver. When two numbers in the same column are given the second one is the result with the filtering heuristic, option *-q*.
- Under "results of NOVA", the column "best." gives the cardinality of the smallest cover found by NOVA, using the options *-e ig -r, -e ih -r, -e ioh -r*. The length of the codes is in the column "bits".

It is a fact that NOVA does consistently better both as cardinality of the cover and length of the codes. It must be pointed out that the results of NOVA are the best out of many runs with different encoding options (the option *-r* effectively tries all possible complementations of the codes). In terms of cover cardinality ISA gets often close to the best of NOVA. The encoding length required by ISA is instead hard to justify. It is a weakness that should be investigated, if one wants to do high-quality state assignment using GPI's. We reiterate that the current version of ISA is addressing the problem of manipulating GPI's with implicit techniques. The next step is to search efficiently encodeable cover of GPI's for specific applications.

11.7.2 Evaluation of the Experiments

We have presented a complete algorithm to compute implicitly minimal covers of GPI's. After the seminal contribution in [39], this is the first in-depth algorithmic study that probes the feasibility of generating and selecting sets of GPI's. Since even small symbolic covers generate large sets of GPI's, implicit techniques have been used to generate GPI's, solve table covering problems and verify encodeability. The implicit procedure to check encodeability is a novelty of this work, together with the technique to select GPI's based on minimal updating sets.

A fair conclusion is that GPI's push to the limit even the most efficient BDD-based computations. For instance the generation of prime implicants induced by GPI's is usually harder

than the generation of primes for the logic functions of the ESPRESSO benchmark. Also the covering problems faced to select covers of GPI's and of prime encoding dichotomies, even though they are unate, are often tougher than those encountered in the ESPRESSO benchmark and in the state minimization of FSM's [66], a reason being the larger variable support of the BDD representations of columns and rows. To be able to solve the examples of the previous tables, the package described in [66] had to be further optimized and inadequacies still remain to be addressed. The implicit check of feasibility has not been a bottleneck in experiments tried so far. Instead the selection of new GPI's based on minimal updating sets failed sometimes due to explosive intermediate BDD operations; they have been partly solved by replacing the computation of $Set\_Minimal$ with a conservative approximation, but for others there is not yet a satisfactory solution. It is an open problem how to drive the selection of GPI's with a more global view, in order to obtain encodeable covers of cardinality less or equal to the best encoded covers obtained by various tools [147]. This was not an objective of this work, even though the experience gained here will be very useful to attack the issue.

The demonstrated techniques exhibit a window of small-medium examples where it is possible to compute minimal symbolic covers using GPI's. Further computational optimizations and improvements to the quality of the search will make it competitive with the best existing tools.

11.8 Conclusions

We have presented a complete procedure to generate and select GPI's [39] based on implicit computations. This approach combines techniques for implicit enumeration of primes and implicit solution of covering tables together with a new formulation of the problem of selecting an encodeable cover of GPI's. The proposed algorithms have been implemented using state assignment of FSM's as a test case. The experiments exhibit a set of medium FSM's where large GPI problems could be solved for the first time, showing that these techniques open a new direction in the minimization of symbolic logic. Since the problem of symbolic minimization is harder than two-valued logic minimization, more practical work is required to improve the efficiency of the implementation and to tie it with good heuristics to explore the solution space of encoding problems. The present contribution shows how to extract a minimal encodeable cover from a large set of GPI's, allowing - in line of principle - the exploration of all minimal encodeable covers. This advances the state-of-art of symbolic minimization, which up to now has been done with various heuristic tools [92, 147, 42, 77], often very well-tuned for their domain of application, but lacking a rigorous

connection beween an exact theory and the approximations made. For instance it is noticeable that these tools (with the exception of ESP_SA) cannot predict the cardinality of the covers that they produce, while the size of a minimized encoded cover of GPI's matches the size of the cover obtained after encoding (with the same codes) and minimizing the original cover.

FSM	table size (row x col) before red.	after red.	mincov calls	table sol.	final sol.	CPU time (seconds) table red.	total
bbara	187 x 4124	98 x 35	9	8	33	329	872
bbtas	28 x 107	9 x 6	3	4	17	3	32
beecount	153 x 176	0 x 0	1	6	12	44	82
chanstb	169216 x 525	0 x 0	1	11	36	1218	1407
cpab	208896 x 1892	683 x 73	4	8	48	7774	11279
dk14	157 x 199	0 x 0	1	17	31	129	311
dk15	88 x 68	0 x 0	1	14	17	9	13
dk17	64 x 164	0 x 0	1	9	19	46	435
dk27	20 x 71	0 x 0	1	4	9	5	23
dol2	20 x 113	19 x 25	2	2	15	8	47
es	23 x 45	0 x 0	1	5	11	1	2
ex3	42 x 495	0 x 0	1	5	21	563	4026
ex5	50 x 301	0 x 0	1	3	19	139	508
ex6	908 x 423	0 x 0	1	22	24	645	672
ex7	48 x 583	0 x 0	1	4	20	106	1101
fstate	5360 x 1605	11 x 11	2	8	21	12770	13402
leoncino	21 x 22	0 x 0	1	5	6	0	1
lion	25 x 29	0 x 0	1	4	10	0	4
lion9	42 x 175	0 x 0	1	2	11	10	55
mc	96 x 71	0 x 0	1	7	11	5	10
ofsync	300 x 97	48 x 24	18	12	33	69	107
opus	914 x 2830	0 x 0	1	14	23	704	958
s8	40 x 206	0 x 0	1	1	13	8	27
scud	2966 x 2533	0 x 0	1	57	95	15633	16885
shiftreg	24 x 89	8 x 6	5	3	8	6	21
slave	2207744 x 16845	-[a]	-	-	-	timeout	-
tav	100 x 81	4 x 4	5	10	11	10	11
test	8 x 5	0 x 0	1	3	3	0	0
virmach	4992 x 144	0 x 0	1	16	17	778	793

[a] timeout 18000 in collapse columns

Table 11.3: Selection of a minimal encodeable GPI cover

FSM	table size (row x col) before red.	after red.	mincov calls	table sol.	final sol.	CPU time (seconds) table red.	total
bbsse	3480 x 34727	-[a]	-	-	-	timeout	-
cf	30208 x 102781	-[b]	-	-	-	-	-
cse	2588 x 21798	0 x 0	1	23	232	6534	14484
dk512	43 x 1777	0 x 0	1	6	52	4150	6108
ex2	86 x 38410	0 x 0	1	3	-[c3]	830	timeout
ex4	1072 x 26759	0 x 0	1	10	20	803	1762
keyb	2666 x 361240	0 x 0	1	8	373	1706	3398
kirkman	100252 x 1081088	-[a]	-	-	-	timeout	-
maincont	67586 x 245784	0 x 0	1	4	-[c4]	115	timeout
mark1	1936 x 50258	5 x 5	3	7	20	1313	5194
modulo12	24 x 9039	24 x 36	17	2	24	50	416
pkheader	140288 x 29099	0 x 0	1	19	36	5850	10299
ricks	31232 x 16561	14 x 14	18	27	39	3301	5378
s1	15336 x 586240	-[b]	-	-	-	-	-
s1a	5120 x 586240	-[b]	-	-	-	-	-
saucier	18496 x 7106239	0 x 0	1	15	[d]	6802	timeout
tma	2028 x 287558	-[b]	-	-	-	-	-
train11	43 x 583	0 x 0	1	2	13	177	711

[a] timeout 18000 in collapse columns

[b] out-of-memory in collapse columns

[c3] timed out 18000 in adding 3rd GPI

[c4] timed out 18000 in adding 1st GPI

[d] timed out 18000 in i_set_minimum

Table 11.4: Selection of a minimal encodeable GPI cover

	FSM	1-hot	results of ISA			results of NOVA	
FSM	cover	cover	min.gpi	min.FSM	bits	best	bits
bbara	60	34	29	27/29	7/5	24	4
bbtas	24	16	13	11/10	6/3	8	3
beecount	28	12	12	10/15	5/4	10	3
chanstb	52	49	26	26/26	2/2	26	2
cpab	76	49	43	43	5	32	3
dk14	56	25	28	25/26	7/5	24	5
dk15	32	17	16	16/18	4/4	16	4
dk17	32	20	18	18/20	8/6	17	3
dk27	14	10	9	9/4	6/9	7	3
dol2	20	13	13	13/10	5/3	9	3
es	12	10	11	8/9	4/3	6	2
ex3	36	21	21	19/22	8/6	17	4
ex5	32	19	18	16/23	9/6	14	4
ex6	34	23	24	24/24	8/6	23	5
ex7	36	20	19	16/24	9/6	15	4
fstate	30	22	21	21	6	16	3
leoncino	8	6	5	5/6	2/2	5	2
lion	11	8	8	8/8	3/3	6	2
lion9	25	10	10	8/10	8/4	8	4
mc	6	10	11	10/10	4/2	8	2
ofsync	41	31	31	32/25	4/4	22	2
opus	22	19	22	16/19	8/7	15	4
s8	22	14	12	12/10	4/3	9	3
scud	127	86	90	78	11	60	5
shiftreg	16	9	6	6/6	5/4	4	3
slave	75	46	-	-	-	35	4
tav	49	12	11	11/11	3/3	11	2
test	4	3	2	2/2	1/2	2	1
virmach	18	16	16	16/16	3/3	14	2

Table 11.5: Final solutions and comparison with NOVA

	FSM	1-hot	results of ISA			results of NOVA	
FSM	cover	cover	min.gpi	min.FSM	bits	best	bits
cse	91	55	-/78	-/55	-/9	45	4
dk512	30	21	-/28	-/23	-/5	18	4
ex4	21	21	18/26	18/21	12/5	14	4
keyb	170	77	-/86	-/72	-/10	47	6
mark1	22	19	19/27	16/20	13/8	17	4
modulo12	24	24	20/17	20/15	11/5	11	4
pkheader	1804	26	32/33	24/26	13/7	24	5
ricks	51	33	39/46	32/32	10/6	30	4
train11	25	11	13/22	12/15	10/5	9	4

Table 11.6: Final solutions and comparison with NOVA

Chapter 12

Conclusions

This thesis investigated algorithms to encode symbolic input and output variables of sequential behaviors represented by STG's or STT's, when the cost function is minimum two-level area. Various techniques developed here were applied or are applicable also to encoding problems with different cost functions and objectives.

Technical contributions have been presented in the area of heuristic symbolic minimization (Chapters 5), satisfaction of encoding constraints (Chapter 6), minimization of GPI's (Chapters 7, 8, 9 and 11) and implicit binate covering (Chapter 10).

In Chapter 5 we have presented a symbolic minimization procedure capable of exploring minimal symbolic covers by using face, dominance and disjunctive constraints. The treatment of disjunctive constraints is a novelty of this work. Conditions on the completness of sets of encoding constraints and a bridge to disjunctive-conjunctive constraints (presented in [39]) have been given.

An invariant of the algorithm is that the minimal symbolic cover under construction is always guaranteed to be encodeable. Encodeability is checked efficiently using the procedures described in Chapter 6. Therefore, this synthesis procedure has predictive power that precedent tools lacked, i.e. the cardinality of the cover obtained by symbolic minimization and of the cover obtained by replacing the codes in the initial cover and then minimizing with ESPRESSO are very close. Experiments show that the encoded covers produced by our procedure are usually smaller or equal than those of the best option of state-of-art tools like NOVA [147].

An improvement to the procedure would be to introduce some iterated expansion and reduction scheme, as in ESPRESSO [11], to escape from local minima. Currently the algorithm builds a minimal symbolic cover, exploring a neighborhood of the original FSM cover, with variations of one single expansion and reduction for each slice of the FSM. A weak point of the current algorithm

is that the final code-length is often too long. Currently the algorithm is unable to trade-off final code-length vs. cardinality of the encoded cover.

In Chapter 6 we have presented a comprehensive solution to the problem of satisfying encoding constraints. We have shown that the problem of determining a minimum length encoding to satisfy face constraints is NP-complete. Based on an earlier method for satisfying face constraints [154], we have provided an efficient algorithm that determines the minimum length encoding that satisfies both input (face) and output (dominance, disjunctive and disjunctive-conjunctive) constraints. It is shown how this algorithm can be used to determine the feasibility of a set of input and output constraints in polynomial time in the size of the input.

A heuristic procedure for solving input encoding constraints with bounded code-length in both two-level and multi-level implementations is also demonstrated. In the multi-level case, only a very time-consuming algorithm based on simulated annealing was known before. This framework has also been used for solving a variety of encoding constraints generated by other applications.

In Chapter 11 we have presented a complete procedure to generate and select GPI's [39] based on implicit computations. This approach combines techniques for implicit enumeration of primes and implicit solution of covering tables together with a new formulation of the problem of selecting an encodeable cover of GPI's. A novel theory of encodeability of GPI's has been developed in Chapter 9.

The proposed algorithms have been implemented using state assignment of FSM's as a test case. The experiments exhibit a set of medium FSM's where hard GPI minimization problems could be solved for the first time, showing that these techniques open a new direction in the minimization of symbolic logic. Since symbolic minimization has an eumerative complexity higher than two-valued logic minimization, more practical work is required to improve the efficiency of the implementation and to tie it with good heuristics to explore the solution space of encoding problems.

The present contribution shows how to extract a minimal encodeable cover from a large set of GPI's, allowing - in line of principle - the exploration of all minimal encodeable covers. This advances the state-of-art of symbolic minimization, otherwise restricted to the use of heuristic tools. that do not guarantee a complete exploration of the solution space. It is true, though, that competing algorithms [92, 147, 146] are often well-tuned for their domain of application, while our prototype of GPI minimization is not yet mature for field applications.

In Chapter 10 we have presented an implicit procedure to solve binate covering problems. It is based on the idea of representing the columns and the rows of a table by labelling functions such that the existence of a 1 or 0 entry at the intersection of a given row and column can be computed

by applying a simple computation on the labels (both the labels and the table entry computation depend from the problem). All sets are represented and manipulated based on BDD's. New BDD operations to manipulate sets and sets of sets were designed, including a primitive operation that, given a binary relation $R(a, b)$, finds the a's (b's) that occur the most or the least with b's (a's). This operation was needed to find implicitly a branching column and compute a maximum independent set to lower bound the computation.

This procedure has been applied both to finding a cover of GPI's and to selecting a minimum-state behavior of a nondeterministic FSM. It has potential applications to many problems of logic synthesis and combinatorial optimization. Very large covering tables that could not be generated or solved with traditional techniques were handled by this implicit algorithm, as experiments in Chapter 11 show.

Bibliography

[1] R. Amann and U. Baitinger. Optimal state chains and state codes in finite state machines. *IEEE Transactions on Computer-Aided Design*, February 1989.

[2] D. Armstrong. On the efficient assignment of internal codes to sequential machines. *IRE Transactions on Electronic Computers*, pages 611–622, October 1962.

[3] D. Armstrong. A programmed algorithm for assigning internal codes to sequential machines. *IRE Transactions on Electronic Computers*, pages 466–472, August 1962.

[4] P. Ashar. *Synthesis of sequential circuits for VLSI design*. PhD thesis, University of California, Berkeley, 1992.

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

[6] P. Ashar, S. Devadas, and A. R. Newton. Optimum and heuristic algorithms for an approach to finite state machine decomposition. *IEEE Transactions on Computer-Aided Design*, pages 296–310, March 1991.

[7] M. Beardslee and A. Sangiovanni-Vincentelli. An algorithm for improving partitions of pin-limited multi-chip systems. In *The Proceedings of the International Conference on Computer-Aided Design*, November 1993.

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

[9] D. Bovet and P. Crescenzi. *Introduction to the theory of complexity*. Prentice Hall, 1994.

[10] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. In *The Proceedings of the Design Automation Conference*, pages 40–45, June 1990.

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

[12] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, November 1987.

[13] R. Brayton, A. Sangiovanni-Vincentelli, G. Hachtel, and R. Rudell. *Multi-level logic synthesis*. Unpublished book, 1992.

[14] R. Brayton and F. Somenzi. An exact minimizer for Boolean relations. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 316–319, November 1989.

[15] F. M. Brown. *Boolean Reasoning*. Kluwer Academic Publishers, 1990.

[16] R. Bryant. Graph based algorithm for Boolean function manipulation. In *IEEE Transactions on Computers*, pages C–35(8):667–691, 1986.

[17] N. Calazans. Boolean constrained encoding: a new formulation and a case study. In *The Proceedings of the International Conference on Computer-Aided Design*, November 1994.

[18] K-T. Cheng and V.D. Agrawal. State assignment for initializable synthesis. In *The Proceedings of the International Conference on Computer-Aided Design*, November 1989.

[19] S.M. Chiu. Exact state assignment via binate covering. *EE290ls Project*, May 1990.

[20] M. Ciesielski, J-J. Shen, and M. Davio. A unified approach to input-output encoding for FSM state assignment. *The Proceedings of the Design Automation Conference*, pages 176–181, June 1991.

[21] M. Ciesielski and S. Yang. PLADE: a two stage PLA decomposition. *IEEE Transactions on Computer-Aided Design*, pages 943–954, August 1992.

[22] O. Coudert. Two-level logic minimization: an overview. *Integration*, 17-2:97–140, October 1994.

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

[24] O. Coudert, H.Fraisse, and J.C. Madre. Towards a symbolic logic minimization algorithm. In *The Proceedings of the VLSI Design 1993 Conference*, pages 329–334, January 1993.

[25] O. Coudert and J.C. Madre. Implicit and incremental computation of prime and essential prime implicants of Boolean functions. In *The Proceedings of the Design Automation Conference*, pages 36–39, June 1992.

[26] O. Coudert and J.C. Madre. A new implicit graph based prime and essential prime computation technique. In *Proceedings of the International Symposium on Information Sciences*, pages 124–131, July 1992.

[27] O. Coudert and J.C. Madre. A new method to compute prime and essential prime implicants of boolean functions. In *Advanced Research in VLSI and Parallel Systems*, pages 113–128. The MIT Press, T. Knight and J. Savage Editors, March 1992.

[28] O. Coudert and J.C. Madre. A new viewpoint on two-level logic minimization. *Bull Research Report N. 92026*, November 1992.

[29] O. Coudert, J.C. Madre, and H.Fraisse. A new viewpoint on two-level logic minimization. In *The Proceedings of the Design Automation Conference*, pages 625–630, June 1993.

[30] O. Coudert, J.C. Madre, H.Fraisse, and H. Touati. Implicit prime cover computation: an overview. In *The Proceedings of the SASIMI Conference*, pages 413–422, 1993.

[31] G. Cybenko, D. Krumme, and K. Venkataraman. Fixed hypercube embedding. *Information Processing Letters*, April 1987.

[32] M. Davio, J.-P. Deschamps, and A. Thayse. *Discrete and Switching Functions*. Georgi Publishing Co. and McGraw-Hill International Book Company, 1978.

[33] W. Davis. An approach to the assignment of input codes. *IEEE Transactions on Electronic Computers*, August 1967.

[34] S. Devadas. General decomposition of sequential machines: Relationships to state assignment. In *The Proceedings of the Design Automation Conference*, pages 314–320, June 1989.

[35] S. Devadas, H-T. Ma, R. Newton, and A. Sangiovanni-Vincentelli. Synthesis and optimization procedures for fully and easily testable sequential machines. In *The Proceedings of the International Conference on Computer-Aided Design*, November 1987.

[36] S. Devadas, H-T. Ma, R. Newton, and A. Sangiovanni-Vincentelli. Mustang: state assignment of finite state machines targeting multi-level logic implementations. *IEEE Transactions on Computer-Aided Design*, December 1988.

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

[38] S. Devadas and R. Newton. Corrections to "Exact algorithms for output encoding, state assignment and four-level Boolean minimization". *IEEE Transactions on Computer-Aided Design*, 10(11):1469–1469, November 1991.

[39] S. Devadas and R. Newton. Exact algorithms for output encoding, state assignment and four-level Boolean minimization. *IEEE Transactions on Computer-Aided Design*, pages 13–27, January 1991.

[40] S. Devadas, A. Wang, R. Newton, and A. Sangiovanni-Vincentelli. Boolean decomposition in multilevel logic optimization. *IEEE Journal of solid-state circuits*, April 1989.

[41] T. Dolotta and E. McCluskey. The coding of internal states of sequential machines. *IEEE Transactions on Electronic Computers*, October 1964.

[42] X. Du, G.D.Hachtel, B. Lin, and A.R.Newton. MUSE:a MUltilevel Symbolic Encoding algorithm for state assignment. *IEEE Transactions on Computer-Aided Design*, pages CAD–10(1):28–38, January 1991.

[43] C. Duff. Codage d'automates et theorie des cubes intersectants. *Thèse, Institut National Polytechnique de Grenoble*, March 1991.

[44] E.I.Goldberg. Matrix formulation of constrained encoding problems in optimal PLA synthesis. *Preprint No. 19, Institute of Engineering Cybernetics, Academy of Sciences of Belarus*, 1993.

[45] E.I.Goldberg. Face embedding by componentwise construction of intersecting cubes. *Preprint No. 1, Institute of Engineering Cybernetics, Academy of Sciences of Belarus*, 1995.

[46] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, 1979.

[47] J. Gimpel. A method of producing a boolean function having an arbitrarily prescribed prime implicant table. *IRE Transactions on Electronic Computers*, EC-14:485–488, June 1965.

[48] J. Gimpel. A reduction technique for prime implicant tables. *IRE Transactions on Electronic Computers*, EC-14:535–541, August 1965.

[49] G.N.Raney. Sequential functions. *Journal of the Association of Computing Machinery*, pages 177–180, 1958.

[50] A. Grasselli and F. Luccio. A method for minimizing the number of internal states in incompletely specified sequential networks. *IRE Transactions on Electronic Computers*, EC-14(3):350–359, June 1965.

[51] A. Grasselli and F. Luccio. Some covering problems in switching theory. In *Networks and Switching Theory*, pages 536–557. Academic Press, New York, 1968.

[52] D. Gregory, K. Bartlett, A. DeGeus, and G. Hachtel. SOCRATES: A system for automatically synthesizing and optimizing combinational logic. In *The Proceedings of the Design Automation Conference*, 1986.

[53] G.Swamy, R.Brayton, and P.McGeer. A fully implicit Quine-McCluskey procedure using BDD's. *Tech. Report No. UCB/ERL M92/127*, 1992.

[54] J. Hartmanis. On the state assignment problem for sequential machines - 1. *IRE Transactions on Electronic Computers*, June 1961.

[55] J. Hartmanis and R. E. Stearns. Some dangers in the state reduction of sequential machines. In *Information and Control*, volume 5, pages 252–260, September 1962.

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

[57] B. Holmer. What are the ingredients for a good state assignment program ? *Tech. Report No. CSE-95-002, EECS Department, Northwestern University*, April 1995.

[58] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.

[59] R. W. House and D.W. Stevens. A new rule for reducing cc tables. *IEEE Transactions on Computers*, C-19:1108–1111, November 1970.

[60] S. Robinson III and R. House. Gimpel's reduction technique extended to the covering problem with costs. *IRE Transactions on Electronic Computers*, EC-16:509–514, August 1967.

[61] 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.

[62] S.-W. Jeong and F. Somenzi. A new algorithm for the binate covering problem and its application to the minimization of boolean relations. In *The Proceedings of the International Conference on Computer-Aided Design*, November 1992.

[63] T. Kam. State minimization of finite state machines using implicit techniques. *Ph.D. Thesis, University of California, Berkeley*, 1995.

[64] T. Kam and R.K. Brayton. Multi-valued decision diagrams. *Tech. Report No. UCB/ERL M90/125*, December 1990.

[65] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. A fully implicit algorithm for exact state minimization. *Tech. Report No. UCB/ERL M93/79*, November 1993.

[66] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. A fully implicit algorithm for exact state minimization. In *The Proceedings of the Design Automation Conference*, pages 684–690, June 1994.

[67] R. Karp. Some techniques for state assignment for synchronous sequential machines. *IEEE Transactions on Electronic Computers*, October 1964.

[68] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, February 1970.

[69] K. Keutzer and D. Richards. Computational complexity of logic optimization. *Unpublished manuscript*, March 1994.

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

[71] D. Krumme, K. Venkataraman, and G. Cybenko. Hypercube embedding is NP-complete. In *Proceedings of SIAM Hypercube Conference*, September 1985.

[72] L. Lavagno. Heuristic and exact methods for binate covering. *EE290ls Report*, May 1989.

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

[74] L. Lavagno, C. W. Moon, R. K. Brayton, and A. Sangiovanni-Vincentelli. Solving the state assignment problem for signal transition graphs. *The Proceedings of the Design Automation Conference*, pages 568–572, June 1992.

[75] D. Lewin. *Computer-Aided Design of Digital Systems*. Russak-Arnold, 1977.

[76] B. Lin. Experiments with jedi. Private communication, October 1989.

[77] B. Lin. Synthesis of multiple level logic from symbolic high-level description languages. *Proceedings of the IFIP International Conference on VLSI*, pages 187–196, August 1989.

[78] B. Lin. Synthesis of VLSI designs with symbolic techniques. *Tech. Report No. UCB/ERL M91/105*, November 1991.

[79] B. Lin, O. Coudert, and J.C. Madre. Symbolic prime generation for multiple-valued functions. In *The Proceedings of the Design Automation Conference*, pages 40–44, June 1992.

[80] B. Lin and A.R. Newton. Implicit manipulation of equivalence classes using binary decision diagrams. In *The Proceedings of the International Conference on Computer Design*, pages 81–85, September 1991.

[81] B. Lin and R. Newton. A generalized approach to the constrained cubical embedding problem. In *The Proceedings of the International Conference on Computer Design*, 1989.

[82] B. Lin and F. Somenzi. Minimization of symbolic relations. In *The Proceedings of the International Conference on Computer-Aided Design*, November 1990.

[83] C. Y. Liu. A system for for synthesis of area-efficient testable FSM's. *Ph.D. Thesis, University of Wisconsin*, 1994.

[84] S. Malik. Combinational logic optimization techniques in sequential logic synthesis. *Tech. Report No. UCB/ERL M90/115*, November 1990.

[85] S. Malik, L. Lavagno, R. Brayton, and A. Sangiovanni-Vincentelli. Symbolic minimization of multilevel logic and the input encoding problem. In *IEEE Transactions on Computer-Aided Design*, volume vol.11, (no.7), pages 825–43, July 1992.

[86] M. Marcus. Derivation of maximal compatibles using Boolean algebra. *IBM Journal of Research and Development*, November 1964.

[87] E. McCluskey. Minimization of Boolean functions. *Bell Laboratories Technical Journal*, November 1956.

[88] E.J. McCluskey and S.H. Unger. A note on the number of internal variable assignments for sequential switching circuits. *IRE Transactions on Electronic Computers*, pages 439–440, December 1959.

[89] P. McGeer, J. Sanghavi, R. Brayton, and A. Sangiovanni-Vincenetelli. Espresso-signature: a new exact minimizer for logic functions. *IEEE Transactions on VLSI Systems*, pages 432–440, December 1993.

[90] C. Mead and L. Conway. *Introduction to VLSI Systems*, chapter 3, pages 85–86. Addison Wesley, 1980.

[91] G. De Micheli. Symbolic design of combinational and sequential logic circuits implemented by two-level logic macros. *IEEE Transactions on Computer-Aided Design*, October 1986.

[92] G. De Micheli, R. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Transactions on Computer-Aided Design*, July 1985.

[93] G. De Micheli, T. Villa, and A. Sangiovanni-Vincentelli. Computer-aided synthesis of PLA-based finite state machines. In *The Proceedings of the International Conference on Computer-Aided Design*, September 1983.

[94] R. E. Miller. *Switching theory. Volume I: combinational circuits*. J. Wiley and & Co., N.Y., 1965.

[95] S. Minato. Zero-suppressed BDD's for set manipulation in combinatorial problems. In *The Proceedings of the Design Automation Conference*, pages 272–277, June 1993.

[96] E. Moore. Gedanken-experiments on sequential machines. In C. Shannon and J. McCarthy, editors, *Automata Studies*. Princeton University Press, 1956.

[97] R. Murgai, R. Brayton, and A. Sangiovanni-Vincentelli. Using encoding in functional decomposition. *Submitted for publication*, 1993.

[98] R. Narasimhan. Minimizing incompletely specified sequential switching functions. *IRE Transactions on Electronic Computers*, EC-10:531–532, September 1961.

[99] L. Nguyen, M. Perkowski, and N. Goldstein. Palmini - fast boolean minimizer for personal computers. In *The Proceedings of the Design Automation Conference*, pages 615–621, July 1987.

[100] A. Nichols and A. Bernstein. State assignments in combinational networks. *IEEE Transactions on Electronic Computers*, June 1965.

[101] P.S. Noe. Remarks on the SHR-optimal state assignment procedure. *IEEE Transactions on Computers*, pages 873–875, September 1973.

[102] P.S. Noe and V.T. Rhyne. A modification to the SHR-optimal state assignment procedure. *IEEE Transactions on Computers*, pages 327–329, March 1974.

[103] P.S. Noe and V.T. Rhyne. Optimum state assignment for the D flip-flop. *IEEE Transactions on Computers*, pages 306–311, March 1976.

[104] C. Papadimitriou. *Computational complexity*. Addison Wesley, 1994.

[105] C. H. Papadimitriou, J.D. Ullman, and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.

[106] R. Parchman. The number of state assignments for sequential machines. *IEEE Transactions on Computers*, pages 613–614, June 1972.

[107] W. Quine. A way to simplify truth functions. *Amer. Math. Monthly*, 62:627–631, November 1955.

[108] J.-K. Rho and F. Somenzi. Stamina. *Computer Program*, 1991.

[109] V.T. Rhyne and P.S. Noe. On equivalence of state assignments. *IEEE Transactions on Computers*, pages 55–57, January 1968.

[110] V.T. Rhyne and P.S. Noe. On the number of distinct state assignments for a sequential machine. *IEEE Transactions on Computers*, pages 73–75, January 1977.

[111] D. Rosenkrantz. Half-hot state assignments for finite state machines. *IEEE Transactions on Computer-Aided Design*, May 1990.

[112] R. Rudell. Espresso. *Computer Program*, 1987.

[113] R. Rudell. Logic synthesis for VLSI design. *Tech. Report No. UCB/ERL M89/49*, April 1989.

[114] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Transactions on Computer-Aided Design*, CAD-6:727–750, September 1987.

[115] A. Saldanha and R. Katz. PLA optimization using output encoding. In *The Proceedings of the International Conference on Computer-Aided Design*, November 1988.

[116] A. Saldanha, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Satisfaction of input and output encoding constraints. *IEEE Transactions on Computer-Aided Design*, 13:589–602, May 1994.

[117] 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*, pages 953–956, October 1969.

[118] T. Sasao. An application of multiple-valued logic to a design of Programmable Logic Arrays. In *The Proceedings of the International Symposium on Multiple-Valued Logic*, 1978.

[119] T. Sasao. Multiple-valued decomposition of generalized Boolean functions and the complexity of Programmable Logic Arrays. *IEEE Transactions on Computers*, C-30:635–643, September 1981.

[120] T. Sasao. Input variable assignment and output phase optimization of PLA's. In *IEEE Transactions on Computers*, October 1984.

[121] T. Sasao. Multiple-valued logic and optimization of programmable logic arrays. *Computer*, pages 71–80, April 1988.

[122] T. Sasao. Application of multiple-valued logic to a serial decomposition of PLA's. In *The Proceedings of the International Symposium on Multiple-Valued Logic*, June 1989.

[123] T. Sasao. On the optimal design of multiple-valued PLA's. *IEEE Transactions on Computers*, C-38, n.4:582–592, April 1989.

[124] G. Saucier, M. Crastes de Paulet, and P. Sicard. Asyl: a rule-based system for controller synthesis. *IEEE Transactions on Computer-Aided Design*, November 1987.

[125] G. Saucier, C. Duff, and F. Poirot. A new embedding method for state assignment. *The Proceedings of the International Workshop on Logic Synthesis*, May 1989.

[126] G. Saucier, C. Duff, and F. Poirot. State assignment using a new embedding method based on an intersecting cube theory. In *The Proceedings of the Design Automation Conference*, 1989.

[127] R. B. Segal. BDSYN: Logic description translator; BDSIM: Switch level simulator. Master's Thesis M87/33, Electronics Research Lab., University of California, Berkeley, May 1987.

[128] M. Servit and J. Zamazal. Exact approaches to binate covering problem. *Manuscript in preparation*, October 1992.

[129] C.-J. Shi and J. Brzozowski. An efficient algorithm for constrained encoding and its applications. *IEEE Transactions on Computer-Aided Design*, pages 1813–1826, December 1993.

[130] F. Somenzi. Cookie. *Computer Program*, 1989.

[131] F. Somenzi. Gimpel's reduction technique extended to the binate covering problem. *Unpublished manuscript*, 1989.

[132] F. Somenzi. Binate covering formulation of exact two-level encoding. *Unpublished manuscript*, March 1990.

[133] F. Somenzi. An example of symbolic relations applied to state encoding. *Unpublished manuscript*, May 1990.

[134] P. Srimani. MOS networks and fault-tolerant sequential machines. *Computers and Electrical Engineering*, 8(4), 1981.

[135] P. Srimani and B. Sinha. Fail-safe realisation of sequential machines with a new two-level MOS module. *Computers and Electrical Engineering*, 7, 1980.

[136] A. Srinivasan, T. Kam, S. Malik, and R. Brayton. Algorithms for discrete function manipulation. *Proc. Int. Conf. CAD (ICCAD-90)*, pages 92–95, November 1990.

[137] R. Stearns and J. Hartmanis. On the state assignment problem for sequential machines - 2. *IRE Transactions on Electronic Computers*, December 1961.

[138] J. Storey, H. Harrison, and E. Reinhard. Optimum state assignment for synchronous sequential machines. *IEEE Transactions on Computers*, pages 1365–1373, December 1972.

[139] Y. Su and P. Cheung. Computer minimization of multi-valued switching functions. *IEEE Transactions on Computers*, September 1972.

[140] Y. Tohma, Y. Ohyama, and R. Sakai. Realization of fail-safe sequential machines by using a k-out-of-n code. *IEEE Transactions on Computers*, November 1971.

[141] H.C. Torng. An algorithm for finding secondary assignments of synchronous sequential circuits. *IEEE Transactions on Computers*, pages 461–469, May 1968.

[142] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. *The Proceedings of the International Conference on Computer-Aided Design*, pages 130–133, November 1990.

[143] J. Tracey. Internal state assignment for asynchronous sequential machines. *IRE Transactions on Electronic Computers*, August 1966.

[144] D. Varma and E.A. Trachtenberg. Design automation tools for efficient implementation of logic functions by decomposition. *IEEE Transactions on Computer-Aided Design*, 8-8:901–916, August 1989.

[145] T. Villa, L. Lavagno, and A. Sangiovanni-Vincentelli. Advances in encoding for logic synthesis. In *Digital Logic Analysis and Design, G. Zobrist ed.* Ablex, Norwood, 1995.

[146] T. Villa, A. Saldanha, R. Brayton, and A. Sangiovanni-Vincentelli. Symbolic two-level minimization. *Submitted for publication*, 1995.

[147] T. Villa and A. Sangiovanni-Vincentelli. NOVA: State assignment for optimal two-level logic implementations. In *IEEE Transactions on Computer-Aided Design*, pages 905–924, September 1990.

[148] Y. Watanabe and R. K. Brayton. State minimization of pseudo non-deterministic fsm's. In *European Conference on Design Automation*, pages 184–191, 1994.

[149] P. Weiner and E.J. Smith. On the number of state assignments for synchronous sequential machines. *IEEE Transactions on Computers*, pages 220–221, April 1967.

[150] W. Wolf. Recoding-derived bounds for input encoding. Submitted for publication, January 1990.

[151] W. Wolf, K. Keutzer, and J. Akella. A kernel-finding state assignment algorithm for multi-level logic. In *The Proceedings of the Design Automation Conference*, June 1988.

[152] W. Wolf, K. Keutzer, and J. Akella. Addendum to "A kernel-finding state assignment algorithm for multi-level logic". In *IEEE Transactions on Computer-Aided Design*, August 1989.

[153] C-C. Yang. On the equivalence of two algorithms for finding all maximal compatibles. *IEEE Transactions on Computers*, pages 977–979, October 1975.

[154] S. Yang and M. Ciesielski. Optimum and suboptimum algorithms for input encoding and its relationship to logic minimization. *IEEE Transactions on Computer-Aided Design*, January 1991.