Laboratoire de Microélectronique
Faculté de Sciences Appliquées

Université Catholique
de Louvain

State Minimization and State Assignment of Finite State Machines: their relationship and their impact on the implementation

Minimisation d'Etats et Codage d'Etats des Machines à Etats Finis: leur rapport et leur impact sur l'implementation

Promoteurs: **M. Davio** (in memoriam)
A.-M. Trullemans
Jury: **M. Ciesielski**
J.-J. Quisquater
A. Thayse
J. Zahnd

Ney Laert Vilar Calazans

Thèse présentée en vue de l'obtention du grade de Docteur en sciences appliquées

October 18, 1993

Abstract

This work analyzes the relationship between two fundamental problems in the VLSI synthesis of finite state machines, viz. state minimization and state assignment.

First, we express each of the problems as a set of constraint classes. The relationship between the two problems is then determined from the dependencies existing among their respective constraint classes. Second, a framework is proposed to uniformly represent all constraints. Based on this framework, efficient techniques are developed to find a finite state machine VLSI implementation that satisfies a chosen set of feasible constraints. We show here that this problem is a generalization of the state encoding problem of finite state machines.

The main contribution of this thesis is to create a clear connection between two fundamental problems of VLSI sequential synthesis, often regarded as unrelated, through the use of the constrained encoding paradigm. The ultimate result is a method that takes advantage of this connection to improve the final quality of VLSI circuit implementations.

Additionally, we propose a generalized formulation for the Boolean constrained encoding problem, an important constituent of several VLSI design subproblems, and we point out the usefulness of such a general problem statement.

To Karin, with all
of my love

Acknowledgements

To the memory of Prof. Marc Davio, my former research advisor, whose contributions were fundamental to the achievements of this work. I will forever remember his keen observations during the unfortunately few afternoons we worked together, at his house in Nil-Saint-Vincent. I will never forget the ideas exchanged during the contemplation of some cozy sunset through the window pane of his living room, by the end of a working day.

To Dr. Anne-Marie Trullemans, my latter research advisor. She accepted me as a research student and competently ensured the continuation and conclusion of the work started under the direction of Marc Davio.

To Prof. Maciej Ciesielski, member of my jury, and also a good friend. Maciek provided me with invaluable advices on the subject of this thesis in several occasions. The comments he provided on the preliminary version of this volume were thorough and relevant. I owe him a lot, and I hope we can continue to cooperate in the future.

To Prof. Jacques Zahnd and to Dr. André Thayse, members of my jury, and also members of my thesis committee. They have dedicated much of their precious time to deeply criticize the theoretical aspects of this work in more than one opportunity. In particular, Prof. Zahnd provided me with numerous corrections for my often awkward notational conventions, and furnished me with many advices that helped to compensate my lack of mathematical background.

To Dr. Jean-Jacques Quisquater, who accepted to be the President of my jury, and who also contributed with several suggestions to enhance the readability of this final volume.

To the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), not only for the financial support provided during my stay in Belgium, but also for the respect this organism has revealed towards the research activities in Brazil.

To the Pontifícia Universidade Católica do Rio Grande do Sul, my employer back in Brazil, for the economical and intellectual assistance supplied in the last four and a half years.

To my Lab colleagues of yesterday and of today: Paulo Fernandes, Luis Claudio, Frank Vos (in memoriam), Bruno Yernaux, Olivier L'Heureux, Ricardo Jacobi, Professor Jorge Barreto and João Netto. They have eased a lot my adaptation here in this otherwise rainy country which is Belgium.

To "the girls", as my wife and I kindly call them: Ana, Débora, Fátima, Macarena and Tania, which have always had time for an additional cup of tea in the middle of an afternoon, every time the inspiration to work had abandoned me. To Father Edélcio and Brother Francisco, for their friendship. To Vivian and Gilberto, who came later, but quickly became the best of friends.

To Eugênio and Fernando, for all the misery we shared during the last months of our respective thesis.

To Antonio Vivaldi, Johann Sebastian Bach, and their interpreters, for the constant assistance in my hours of deadliest desperation.

To my parents, Paulo e Mariinha. Even if we are more than ten thousand kilometers apart,

they have always played an essential role in this work. During many years they sacrificed much of their own lives to ensure the best education to my brothers and to me. I hope that this thesis will be a little compensation to their immeasurable efforts. I know they will like it, maybe even more than I do.

To Karin, my wife, friend and more than this, for the love she dedicates to me, for the companionship she reveals in the everyday life, and for the pleasure that is to live with her. Karin has unconditionally sustained me during this whole work. She was always ready to encourage me, whenever the difficulties seemed impossible to surmount. She even abdicated of taking a well-deserved rest after her own PhD was over, just to help me finishing mine. I have no words to tell what I feel for her. Thank you, Ka!

Contents

0 Introduction **1**

0.1 Models for the Design Process 1

0.2 Goals and Scope of the Work 3

0.3 Dissertation Outline 6

I Preliminaries 9

1 Related Work **11**

1.1 Combinational Logic Optimization Techniques 11

1.1.1 Two-level Optimization 12

1.1.2 Multilevel Optimization 12

1.2 Sequential Logic Synthesis Techniques 13

1.2.1 The Behavior-driven Approach 13

1.2.1.1 State Minimization 14

1.2.1.2 State Assignment 15

1.2.1.3 Simultaneous State Minimization and State Assignment 18

1.2.2 The Structure-driven Approach 19

1.2.3 Relationship Between Approaches 19

1.2.4 Future Approaches 20

2 An Introductory Case Study **23**

2.1 The Case Study 23

2.1.1 SM-Related Considerations 24

2.1.2 SA-Related Considerations 27

2.1.3 Logic Level Assumptions 29

2.1.4 Three State Assignment Solutions for Machine BEECOUNT 30

2.2 Discussion 33

3 General Definitions **35**
3.1 Binary Relations and Discrete Functions 35
3.2 Finite Automata and Finite State Machines 39
3.3 Representations of Discrete Functions 41
3.3.1 Cubical Representations of Discrete Functions 45
3.3.1.1 Cube Table Schemes for Sets of Discrete Functions 49
3.3.1.2 Cube Table Scheme for Sets of Switching Functions 50
3.3.1.3 Cube Table Mixed Schemes 52

II Constraints: Nature, Generation and Relationship **55**

4 State Minimization Constraints **57**
4.1 State Minimization Definitions 57
4.2 State Minimization Problem Statement 59
4.3 Compatibility and Incompatibility Constraints 60
4.4 Covering Constraints 61
4.5 Closure Constraints 62
4.6 Generation of the SM Constraints 63
4.7 Complexity of the SM Constraints Generation 65

5 State Assignment Constraints **67**
5.1 The FSM Assignment Problem 67
5.2 Input Constraints 73
5.3 Output Constraints 78
5.3.1 Dominance Constraints 78
5.3.2 Disjunctive Constraints 79
5.4 Complexity of Generating the SA Constraints 80
5.4.1 Generation of the Input Constraints 80
5.4.2 Generation of the Output Constraints 81

6 Relationship among SM and SA Constraints **83**
6.1 Extending Constrained Encoding Assignments 84
6.2 State Splitting and Equivalent FSMs 86
6.3 Justifying the Use of Extended Assignments 89
6.4 Violation of SM Constraints by Input Constraints 91

6.5 Violation of Input Constraints by SM Constraints 93
6.6 Conflicts within and with Output Constraints 97
6.7 Conflicts among Output Constraints 97

7 Conclusions on Constraint Nature and Generation 99

III A Unified Framework for SM and SA Constraints 101

8 Pseudo-Dichotomies 103
8.1 The Pseudo-Dichotomy Definition 103
8.1.1 Generality of the Pseudo-Dichotomy Definition 105
8.2 Pseudo-Dichotomies and Encoding 106

9 Pseudo-Dichotomies and Constraint Representation 109
9.1 Representing Constraints with Pseudo-dichotomies 109
9.1.1 Representing SM Constraints 110
9.1.2 Representing SA Constraints 111
9.2 The Pseudo-Dichotomy Framework 112
9.2.1 Building the Local Part 113
9.2.2 Building the Global Part 116

10 Conclusions on the Unified Framework 117

IV Encoding by Constraints Satisfaction 119

11 The Boolean Constrained Encoding Problem 121
11.1 Boolean Constrained Encoding - Statement 121
11.2 The Two-level SM/SA Problem Statement 125
11.2.1 Boolean Constrained Encoding and the PD Framework 126
11.3 Solutions to Constrained Encoding Problems 126

12 The ASSTUCE Encoding Method 129
12.1 Solving the SM/SA Problem 129
12.2 ASSTUCE Method Overview 130
12.2.1 The First Iteration 131
12.2.2 The Subsequent Iterations 135

12.2.3 ASSTUCE Method Discussion 136
12.3 The ASSTUCE Method Data Structures 137
12.3.1 Data Structure for the Global Part 137
12.3.2 Data Structures for the Local Part 137
12.4 ASSTUCE Heuristic Improvements and Extensions 141
12.4.1 Improvements 141
12.4.2 Extensions 142

13 Conclusions on Constraint Satisfaction 143

V Implementation, Results and Final Remarks 145

14 Implementation and Benchmark Results 147
14.1 The ASSTUCE Implementation 147
14.1.1 The ASSTUCE Program Implementation Environment 151
14.2 Benchmark Tests 152
14.2.1 The Benchmark FSMs 152
14.2.2 Benchmark Tests Strategy 153
14.2.3 The Compared Parameters 156
14.2.4 Benchmark Tests with ASSTUCE 157
14.2.4.1 ASSTUCE Benchmark Tests Discussion 157
14.2.5 ASSTUCE versus Complete Encoding Serial Strategy 160
14.2.5.1 ASSTUCE versus Complete Serial Strategy - Discussion 167
14.2.6 ASSTUCE versus Partial Encoding Serial Strategy 167
14.2.6.1 ASSTUCE versus Partial Serial Strategy - Discussion 175
14.2.7 Benchmark Tests - Conclusions 175

15 Overall Conclusions and Future Work 177
15.1 Overall Conclusions 177
15.2 Future Work 180

Bibliography 181

Appendices **191**

A I/O Formats for FSM and Discrete Function Descriptions **211**

B Requirements for an FSM Exploratory Environment **213**

C Manual Pages for ESPRESSO, DIET, NOVA and STAMINA **215**

D Manual Pages for ESPRESSO, DIET, NOVA and STAMINA **217**

List of Figures

0.1 Y-diagram - a model to represent the design process of digital ICs 2
0.2 Simple Data Flow diagram of a VLSI IC design system 4
0.3 Y-diagram for the behavior-driven approach to sequential synthesis 6

1.1 Y-diagram for the structure-driven approach to sequential synthesis 20

2.1 Merge graph for FSM BEECOUNT 26
2.2 One optimal set of entry subsets for machine BEECOUNT 27
2.3 Three Combinational Part PLAs for machine BEECOUNT 32

3.1 Sequential Mealy machine. 41
3.2 Hasse diagram for Example 3.3 46
3.3 Symbolic and positional cube schemes to represent the f discrete function 50
3.4 Behavior of function F using symbolic and positional cube schemes 52

4.1 Initial and final compatibility tables for example 4.1 64
4.2 Merge and compatibility graphs for example 4.1 65

5.1 Original and minimized cube tables for example 5.2 76
5.2 Three assignments and corresponding minimized cube tables for example 5.2 . . 76
5.3 Cube table optimization using dominance constraints 78
5.4 Cube table optimization using disjunctive constraints 80

6.1 Merge graph for example 6.1 85
6.2 Flow table, cube table, minimized cube table and full input constraints for FSM $\mathcal{A}$ 94
6.3 Flow table, cube table, minimized cube table and full input constraints for FSM $\mathcal{A}'$ 95
6.4 Flow table, cube table, minimized cube table and full input constraints for FSM $\mathcal{A}$ 96
6.5 Flow table, cube table, minimized cube table and full input constraints for FSM $\mathcal{A}'$ 96

8.1 State encoding and final PLA for machine LION9 107

11.1 Flow table and input constraints for FSM $\mathcal{A}$ 123

12.1 Symbol array and PD array during first move computation 138
12.2 Bucket list B_0 before the first move 139

14.1 Execution flow for the ASSTUCE program 148
14.2 Theoretical bound versus measured execution time for ASSTUCE 160
14.3 ASSTUCE versus complete encoding - area - C-group 163
14.4 ASSTUCE versus complete encoding - area - I-group 164
14.5 ASSTUCE versus complete encoding - product terms - C-group 165
14.6 ASSTUCE versus complete encoding - product terms - I-group 165
14.7 ASSTUCE versus complete encoding - execution time - C-group 166
14.8 ASSTUCE versus complete encoding - execution time - I-group 166
14.9 ASSTUCE versus partial encoding - area - C-group 170
14.10 ASSTUCE versus partial encoding - area - I-group 170
14.11 ASSTUCE versus partial encoding - product terms - C-group 171
14.12 ASSTUCE versus partial encoding - product terms - I-group 171
14.13 ASSTUCE versus partial encoding - sparsity - C-group 172
14.14 ASSTUCE versus partial encoding - sparsity - I-group 172
14.15 ASSTUCE versus partial encoding - transistor cardinality - C-group 173
14.16 ASSTUCE versus partial encoding - transistor cardinality - I-group 173
14.17 ASSTUCE versus partial encoding - time - C-group 174
14.18 ASSTUCE versus partial encoding - time - I-group 174

15.1 The ASSTUCE approach 178

List of Tables

2.1 Flow table describing the behavior of the FSM BEECOUNT 24
2.2 Flow table for the BEECOUNT FSM after merging states 3 and 4 25
2.3 Three valid state assignments for machine BEECOUNT 30
2.4 Quantitative comparison for three PLA implementations of machine BEECOUNT . 32

3.1 Truth table for Example 3.4 48
3.2 Three cube tables for Example 3.4 49
3.3 Componentwise conjunction of cubes 51
3.4 Componentwise supercube operation 52

4.1 Flow table for example 4.1 64

6.1 Flow table for example 6.1 85
6.2 A valid non-functional, non-injective assignment for example 6.1 86
6.3 Flow table for original FSM in example 6.2 87
6.4 Flow table for the partition augmentation FSM in example 6.2 88
6.5 A functional, injective assignment for closed cover κ of example 6.3 91
6.6 A non-functional, non-injective assignment for the state set S of example 6.3 . . 91

8.1 Flow table for machine LION9 107

12.1 Rules for building the evaluation matrix E 131
12.2 Rules to build E and ν incrementally after a move 138
12.3 Rules to build B_0 and B_1 from E and ν 140

14.1 Characteristics of the MCNC FSM benchmark test set - C-group 153
14.2 Characteristics of the MCNC FSM benchmark test set - I-group 154
14.5 ASSTUCE versus complete encoding strategy for the C-group of FSMs 161
14.6 ASSTUCE versus complete encoding strategy for the I-group of FSMs 162
14.8 ASSTUCE versus partial encoding strategy for the I-group of FSMs 169

Chapter 0

Introduction

The synthesis and the analysis of *very large scale integration* (VLSI) digital *integrated circuits* (ICs) require the extensive use of hierarchical decomposition of the *design process* into *abstraction levels*. This is due to the high degree of complexity achieved by present VLSI digital ICs, which are typically composed by a number of electronic components in the order of 10^5 to 10^6 [73, 96]. Besides the abstraction levels classification, we can frequently rank *design descriptions* according to the nature of the information they convey. This information can be classified into three *domains of description*: physical, structural and behavioral. For example, a schematic diagram depicting an interconnection of logic gates is typically a structural description on the logic level of abstraction, since it tells how to connect logic elements to perform a given task, without concern for the geometrical details of the final circuit layout. On the other hand, the symbolic flow table of a finite state machine is a behavioral description on the logic level. It tells what is the expected behavior of the machine, disregarding both elements' interconnections and geometrical disposition of devices needed to realize the machine as a circuit. Before establishing the scope of the present work, we introduce some abstract models that facilitate the understanding of the IC design process.

0.1 Models for the Design Process

The combination of abstraction levels and domains of description provides a simple, yet useful model for interpreting the digital IC design process. Such two-dimensional model was first proposed by Gajski and Kuhn in [52], and may be described by the so-called Y-diagram, a sample of which appears in Figure 0.1. In this diagram, concentric circles stand for abstraction levels, while radiating segments stand for the domains of description. Each intersection of a circle with a straight line segment represents one distinct design description, like the schematic diagram or the flow table mentioned above. A design process is depicted as a directed graph over the Y-diagram, where the vertex set is the set of design descriptions (corresponding to the set of points determined by the intersections between circles and straight lines). The graph's edge set corresponds to the set of transformations applied to these descriptions.

Some comments about the Y-diagram are helpful for its understanding. First, the center of the diagram corresponds conceptually to the final design, i.e. a description containing all information needed to fabricate the IC. Design tools perform transformations on design descriptions

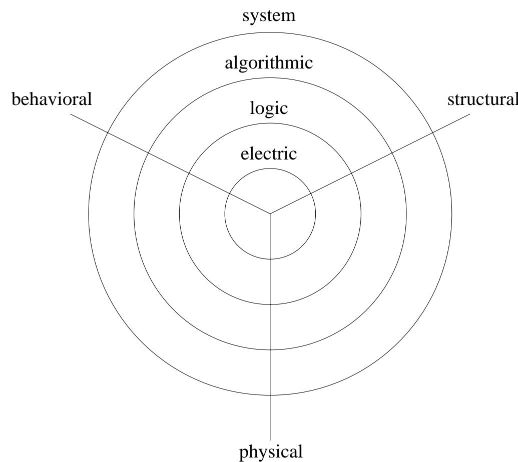


Figure 0.1: Y-diagram - a model to represent the design process of digital ICs

and can thus be identified with edges (or, more generally, paths) in the graph describing the design process. The tools can be classified according to the kind of transformation operated on their input description. For example, analysis tools produce an output description which is not closer to the center of the diagram than the input description. Conversely, synthesis tools produce a description which is not farther from the center of the diagram than their input. Optimization tools correspond to self-loops in the design process graph. An ideal automatic design process would resemble a spiral path going from the behavioral description with the highest level of abstraction to the center of the diagram. In this ideal case, the abstraction level of the successive descriptions would become lower at each synthesis step.

Given the above design process model, we may envisage a general model describing a typical VLSI IC design system. Before doing so, however, we have to partition the Y-diagram design descriptions into three groups: high-level, logic level and low-level descriptions. High-level descriptions correspond roughly to the region between the behavioral and structural axes, and outside the logic circle in Figure 0.1. An example is a hardware description language (HDL) definition of an IC. Low-level descriptions convey a greater amount of electrical or geometrical details, and are thus situated along the physical axis and/or inside the electric circle in Figure 0.1. Examples are the final layout of an IC or its floor plan. Logic descriptions are intermediate between high-level descriptions and low-level descriptions. Examples of the latter are schematic diagrams, describing an interconnection of logic gates, and symbolic flow tables, describing the behavior of an FSM. With each description group, we associate a subsystem of our typical design system, that is the subsystem responsible for manipulating the descriptions in its associated group.

VLSI ICs are sequential circuits[1]. We may accomplish the description of any such circuit by using the finite state machine (FSM) model [72], but this may be impractical in many cases. Alternatively, we may model the circuit as a network of communicating blocks, each of them described as an FSM. We assume here a high-level design subsystem that produces as output such a network, extracted from some high-level circuit description.

To the high-level design subsystem follows the logic level design subsystem, which transforms a network of communicating FSMs into a netlist of library elements. We call the problem of

[1] We remind that we may consider combinational circuits as 1-state sequential ones.

designing general purpose sequential circuits ***sequential logic design***, whenever the circuits are initially described in the logic level of abstraction, and no special purpose technique is available to implement them more efficiently. For instance, there are several specific techniques to design FSMs that describe arithmetic functions such as adders or multipliers. There are two reasons that justify the application of such techniques. First, the widespread use of the blocks they are designed for in VLSI IC design, and second, the performance gain achieved by using these techniques with regard to using general purpose ones.

After logic design comes the low-level design subsystem, which transforms the netlist of library elements into a geometrical detailed description of the IC implementing the behavior initially described. Figure 0.2 describes our typical VLSI IC design system using a Data Flow Diagram notation [53]. Note that the user's interference is considered in the diagram and that it may occur at any step of the design process. This representation is in accordance with the current trend of allowing user intervention during the execution of automated synthesis procedures. Also, note that this oversimplified model of a design process is useful for our local use only. High-level and low-level issues were intentionally overlooked, and only a very simple model of the logic level is retained.

0.2 Goals and Scope of the Work

The present work concerns the sequential logic design step of the IC design process, indicated in Figure 0.2 by the shadowed rounded box. Accordingly, all design descriptions we manipulate herein (flow tables, Boolean functions, encoded PLAs, etc.) belong to the logic level of abstraction. The sequential logic design step can be partitioned into two main tasks: *sequential logic synthesis* and *sequential logic analysis*. In order to allow such a bipartition of this design step, we may consider that design errors correction is a form of optimization, and that design optimization is a synthesis activity.

A finite state machine is a convenient model to represent a sequential circuit in the logic level of abstraction. The complete task of automatically designing a system of general purpose communicating FSMs, requires the use of a considerable number of synthesis and analysis design tools to perform tasks such as:

- FSM/system behavioral simulation and/or formal verification;
- FSM decomposition;
- FSM/system state minimization;
- FSM/system state assignment;
- logic optimization;
- test pattern generation;
- test structures insertion, etc.

The task of building and integrating such a set of tools comprises the concurrent work of several researchers [38]. To render the present work feasible, we have built this dissertation

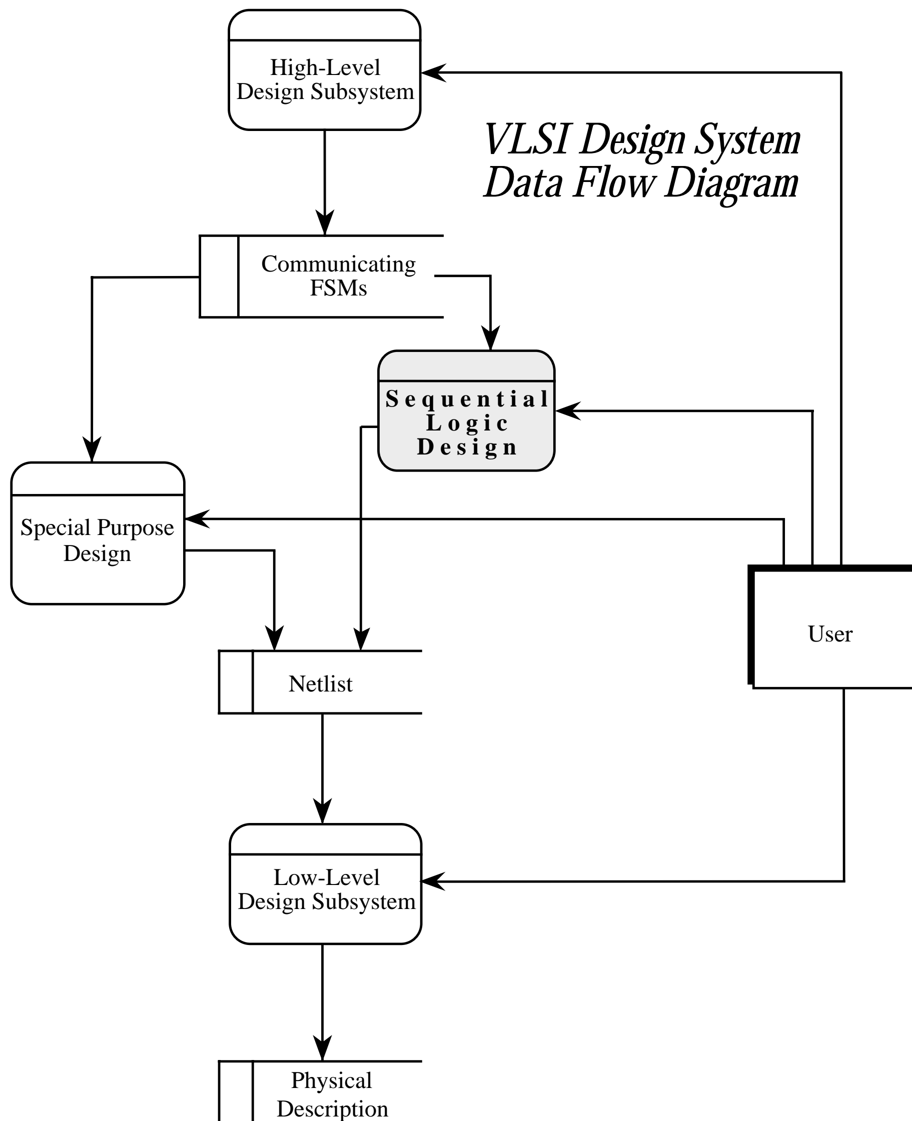


Figure 0.2: Simple Data Flow diagram of a VLSI IC design system

around two major topics of the sequential logic synthesis of FSMs, viz. "state minimization" and "state assignment".

The main objective of the present work is to study the relationship between *state minimization* (SM) and *state assignment* (SA) of FSMs. State assignment is also called *state encoding*. We use both terms interchangeably in the rest of this work. SM and SA are the two main tasks to accomplish during the logic synthesis of FSMs. Intuitively, we know that these tasks are not independent, and too strong a state minimization can actually deteriorate the performance level accessible to the best available state assignment technique. In fact, we define here a new problem. This new problem is called the *two-level* SM/SA *problem*. The main original contribution of this dissertation in the practical sense, is a method to solve the two-level SM/SA problem using a constrained encoding approach. This method considers state minimization during the solution of the state assignment problem.

Note that we address the general problem of minimizing incompletely specified FSMs. Minimizing completely specified machines is a special case situation, which can be easily dealt with, since there are algorithms with worst-case running time $\mathcal{O}(n \log n)$ to solve it exactly, with n representing the number of states in the original machine [66]. We are interested here in real-world machines. Thus, references to state minimization for the rest of this dissertation regard the general problem, unless it is explicitly stated otherwise. This is a sound choice, for real machines are seldom completely specified. Another basic choice we make is to limit attention to *synchronous* FSMs. Although most of the constraint formulations provided here are independent of clock considerations, a generalization effort will later be needed to show their validity for asynchronous circuits in general. We have also limited attention to single FSM circuits.

Besides state minimization and state assignment, *combinational logic optimization techniques* will be extensively used here. We will apply these techniques to generate some of the constraint classes associated with the SA problem, as well as to provide bounds for evaluating the involved cost functions. These techniques will additionally allow us to measure the quality of the obtained results.

The SM and SA problems are part of the general sequential logic synthesis task, as we defined in the beginning of the present Section. This task can be stated as the transformation of some non-physical logic description of a sequential circuit into some *implementable* design description that is *optimal* in some sense. We will restrict attention to transformations that start in the behavioral domain (e.g. a symbolic flow table of an FSM) and which end in the structural domain (e.g. the personality matrix of a PLA). Also, we assume our final target to be a two-level (in particular, a sum-of-products) design description, in order to reduce the problem's complexity. Figure 0.3 illustrates our approach, which we call a *behavior-driven* approach to sequential synthesis, because the starting point is a behavioral description. In Chapter 1, we introduce other ways to regard the sequential logic synthesis task, which can be found in recent publications. There, we will also compare the tradeoffs associated with these approaches and ours.

A secondary objective here is to compare the two possible strategies to solve the SM and SA problems. These are the simultaneous state minimization and assignment (or *simultaneous strategy*), and the application of minimization procedures followed by assignment procedures (or *serial strategy*).

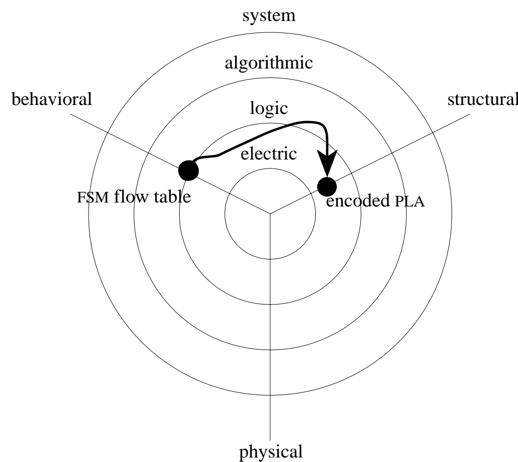


Figure 0.3: Y-diagram for the behavior-driven approach to sequential synthesis

0.3 Dissertation Outline

The main original contribution of this dissertation in the practical sense, is a method to solve the two-level SM/SA problem, i.e. a method to solve the state assignment while considering the state minimization problem.

We divide this document into five parts.

Part I, Preliminaries, comprises three Chapters. Chapter 1 provides a justification for the work, by comparing previous approaches to sequential logic synthesis with our proposition. Chapter 2 gives an informal overview of the main ideas through the presentation of a case study of FSM minimization and encoding. The goal of Chapter 3 is to formally introduce the fundamental concepts we use in the rest of the work.

In order to establish the relationship between the SM and the SA problems, we decompose them into relations we call *constraints*. Parts II to IV are dedicated to describe the manipulation of these constraints and constitute the core of the dissertation.

Part II analyzes in detail the nature of the constraints, and presents techniques for their generation from an FSM behavioral description. In Chapter 4, the constraints yielded by state minimization are considered, while Chapter 5 tackles the constraints obtained from the decomposition of the state assignment problem. The relationship between SM and SA constraints are the object of Chapter 6, which contains several of the original findings of the present work. This part ends with a set of conclusions about the SM and SA constraints, depicted in Chapter 7.

Part III proposes a new, unified framework for representing all relevant constraints described in Part II. It begins with the introduction of some additional terminology in Chapter 8. Chapter 9 sets up the unified framework, showing how to accommodate each relevant class of constraints into it. At this point, we obtain the fusion of both SM and SA into a single problem, i.e. that of satisfying a set of constraints represented inside the framework. This is what we call the two-level SM/SA problem. Chapter 10 gathers a set of conclusions about Part III.

Part IV explains how to solve the constraint satisfaction problem determined by the unified framework proposed in part III. Chapter 11 provides a new statement of the *Boolean constrained*

encoding problem, and discusses the genericity of this new statement. In the same Chapter, we formally state the two-level SM/SA problem and show that it is an instance of the Boolean constrained encoding problem. In Chapter 12, we describe a method to solve the two-level SM/SA problem. A set of conclusions about this part is drawn in Chapter 13.

Part V closes the work. Chapter 14 describes the characteristics of a prototype program that implements the method proposed in Chapter 12, and compares the results obtained by its execution with those achieved by applying some previous approaches to solve the SA and SM problems. Finally, Chapter 15 gathers the main contributions of the study, a set of final conclusions and plans for future work in the subject.

Part I

Preliminaries

Chapter 1

Related Work

We now review the basic approaches to the solution of the sequential logic synthesis problem. The main objective of the present Chapter is to position our work with regard to previous publications, as well as to provide a justification for the proposal included herein.

The sequential logic design and the combinational logic design problems rely on the theoretical findings of discrete function mathematics [35, 114]. One way to solve the former is to model a sequential circuit as a *Mealy* FSM [83]. In a Mealy FSM model, the FSM output and transition functions are gathered in a lumped combinational network, while the state information is kept in a synchronous register, which interposes a time barrier in the feedback lines characterizing the sequentiality of the implementation. In this way, the original design problem is made simpler [64], since timing problems such as *races* and *hazards* are easily eliminated. If the FSM specification is an already binary encoded description, we solve the problem through the application of *adequate* combinational synthesis techniques to the combinational part. On the other hand, if the original description is stated symbolically, there is a prior step left, namely the obtainment of some *optimal* binary encoding of the machine's combinational network from the symbolic information. This step comprises the consideration of the sequential aspects of the circuit, and includes the search for a solution to both state minimization and state assignment problems, as well as of input and output assignments, if these are also stated symbolically.

Providing a thorough view of the field of sequential logic synthesis is out of the scope of this work. Instead, we concentrate in the description of the outstanding contributions in theoretical and practical aspects of the problem. We give special attention to works that have generated effective synthesis tools.

The next Section discusses briefly combinational logic optimization techniques. Next, appears a review of sequential synthesis research related to the state minimization and the state assignment problems.

1.1 Combinational Logic Optimization Techniques

Combinational (logic) optimization has received continued attention for a long time, and is still regarded as containing open questions, even though many methods exist that solve restricted versions of the problem. Most methods start with a behavioral description, manipulating discrete functions according to Boolean or algebraic techniques. However, some methods deal

directly with structural information, since this allows the use of more concrete cost functions to evaluate the optimization results.

The first successful methods dedicated themselves to the search of a *two-level* solution (in particular, under the form of either a *sum-of-products* or a *product-of-sums*) to the optimization problem. This simplification is due to the existence of area-efficient layout counterparts for two-level representations, like ROMs and PLAs [49]. The two-level representations assumption eases the task of combinational optimization, since the solution space is significantly reduced. However, this is still a very complex problem, for which the extensive use of heuristic optimization procedures is fundamental. Recently, several efforts appeared to address the more general problem of optimization of factored forms, also called *multilevel optimization*. The following sections discuss two-level and multilevel optimization tools.

1.1.1 Two-level Optimization

A first successful heuristic two-level minimizer tool was MINI, developed at IBM [65]. On the academia side, we have the tools PRESTO [21] and ESPRESSO [16], originated from the research in the University of California at Berkeley. ESPRESSO has been since then extended to provide exact logic minimization capability [100], and to allow multiple-valued logic minimization [101]. This tool has become an industrial *de facto* standard, and it is widely used for various aspects of combinational, as well as sequential synthesis. Additional efforts concentrated in improving the ESPRESSO results in specific aspects, such as the program McBOOLE developed at the McGill University of Montreal [30], which performs exact two-level minimization comparing favorably with ESPRESSO for small to medium-sized examples (up to 20-input, 20-output variables), or as the ASYL system, developed at the INPG of Grenoble [104], which capitalizes on a totally different, rule-based approach to the combinational optimization problem.

With the exception of ASYL, all of the above tools use versions of a *canonical recursive paradigm* [29] to decompose the initial problem into a set of simpler functions. Morreale [86] applied this paradigm to the determination of prime and irredundant normal forms of switching functions. The paradigm relies upon the application of the well-known Shannon expansion to switching function decomposition. The tools cited above introduced heuristic techniques to cut the complexity of the canonical recursive paradigm, the most important of which are based on the concept of *unate discrete functions* [35].

For a comprehensive review of two-level combinational optimization techniques we may consult [16].

1.1.2 Multilevel Optimization

One early tool to perform multilevel logic optimization was LSS [32], developed at IBM. This tool applied local transformations to a netlist of gates, using a rule-based approach. Meanwhile, as the first publications about the theoretical [14] and practical [15] aspects of the two-level minimizer ESPRESSO appeared, Brayton and McMullen introduced concepts connected to the multilevel logic minimization of combinational functions using algebraic operations [18]. These concepts, which are called *algebraic kernel* and *algebraic* or *weak division*, were later used as the basis to the implementation of at least three tools dedicated to multilevel combinational

optimization: the IBM's Yorktown Silicon Compiler [13], MIS from the University of California at Berkeley [19], and BOLD from the University of Colorado at Boulder [11].

Other research efforts started with these concepts and went further in the research of good solutions for the problem of multilevel optimization. The SOCRATES tool from General Electric [57] added a rule-based expert system to customize the generic solution obtained from the multilevel logic synthesis step, adapting the results to different layout strategies and several technologies. The *transduction method*, from the University of Illinois at Urbana, takes an input description under the form of a schematic diagram of 2-input NOR gates, and apply transformations over it on the structural domain. The transformations are based on the computation of don't cares for each gate output and on the concept of *permissible functions* developed by Muroga et al in [87]. Another tool, PHIFACT, developed by the Philips Research Laboratory at Brussels, generalizes the kernel and division concepts, obtaining their more powerful and more complex Boolean counterparts [120].

Multilevel logic synthesis poses a harder problem than its two-level counterpart, since the cost functions used to evaluate the optimization results correlate less well to physical parameters such as circuit surface, timing and dissipated power. This explains the success of structural methods like transduction and the one proposed in the LSS tool, which are inherently closer to the final design description than the behavioral methods. Despite the large number of publications on the subject, multilevel synthesis is still a little understood problem. We refer to [12, 17] for a broad review of the literature on the subject.

1.2 Sequential Logic Synthesis Techniques

We may classify the approaches to sequential (logic) synthesis into two families, according to the domain of description to which belongs the initial circuit specification. The first family, whose members we call *behavior-driven* approaches, was introduced in Chapter 0, using Figure 0.3. The second family consists of approaches where the initial circuit description is a structural one, and these approaches are accordingly named *structure-driven*. In this Section we discuss the research related to both families. We also provide some comments on how to combine these approaches to obtain a better solution for the sequential logic synthesis problem, and point out some present research directions.

The focus of the discussion is on state minimization and state assignment of single FSMs. A complete review should include considerations of FSM decomposition and the consequent communicating FSMs issues. This is out of the scope of this work. A more thorough review of sequential logic synthesis exists in [5].

1.2.1 The Behavior-driven Approach

This approach was presented in Chapter 0, and it is illustrated in Figure 0.3. The idea is to start with a purely behavioral description of an FSM, producing a lower level structural description of it. Its main advantage is to allow a smooth transition from high-level to logic level descriptions, because FSMs are naturally identified during the compilation of HDL circuit descriptions [107]. The next sections discuss the SM and SA problems in this approach.

1.2.1.1 State Minimization

The state minimization of incompletely specified FSMs experimented a period of effervescence in the 60's, when Paull and Unger, and Grasselli and Luccio set up the basic theory [91, 56], its popularity declining in the next two decades. One proof of this last fact is the present lack of state minimization programs in most commercial and academic FSM synthesis systems. Examples of these are the OCTTOOLS [48], from the University of Berkeley and the FSM Synthesizer System, from the AT&T Bell Laboratories [112]. It is as if the designers of those and similar systems simply had taken for granted that the generation of FSM specifications without redundant states was an easy task. Quite recently, some works have shown [8, 69] that this is not the case. They proposed state minimization tools and applied them to a widely available set of benchmark FSMs, which are part of the MCNC[1] benchmarks [118]. Some of these machines turned out to be, after a state minimization step, a one-state FSM. This implies that we can generate the functionality of this FSM using a combinational circuit alone. However, several modern synthesis systems, e.g. the ones cited above, when trying to implement the original specification of these machines do produce a complete sequential circuit, with a combinational part realizing the transition and output functions, and registers to store the present state. This lack of insight of present systems may result in waste of precious semiconductor surface. A more important result is that, in the present version of the MCNC benchmarks, more than 40% of the machines (22 out of 53) may be implemented with less states than the original specification.

The general theory of the SM problem superseded previous formalization efforts, like those in [55]. Paull and Unger provided the main results in [91]. They defined the concepts of *compatibles*, *incompatibles*, *maximal compatibles* and *closed sets of compatibles* (see Section 4.1 for definitions of these terms). The SM problem was stated by them as a *closure-covering* problem (following the terminology of Davio et al [37]). Grasselli and Luccio [56] contributed with an important enhancement, by proving that only a restricted subset of all compatibles (namely, the *prime compatibles*, to be defined in Section 4.1) should be considered while looking for an optimum solution. Most modern works on the field restrict themselves to the proposition of efficient techniques to solve the original problem formulated by these authors [69]. Note that there are two complex subproblems at the heart of the solution of the SM problem, viz. finding the complete set of maximal compatibles (resp. prime compatibles), and searching for a closed cover of compatibles (or maximal compatibles, or prime compatibles). These problems are NP-hard, since the NP-complete problems of clique finding and set covering [54] can be reduced to the first and the second, respectively. The whole SM problem was already shown to be NP-complete [95]. Several works suggest heuristics or quick exact techniques to generate the maximal or prime compatibles, and then use some general or particular technique to solve the covering problem [69, 59, 94]. Since the number of prime compatibles or even of maximal compatibles can be very large, other authors propose techniques that avoid their generation, and still obtain a good, although not optimum, solution to the problem [7, 9].

Since the mathematical background of the SM problem is very well established, and given that the associated cost functions (number of states) relate well enough to physical parameters (circuit area, propagation delay and dissipated power), no alternative formulation was proposed to the general theory, even if the number of publications on the subject in the last years is quite abundant. See [97] for a significant review of the works published between 1959 and 1985, and

[1]The FSMs included in this benchmark set come from industrial and academic environments, and are intended to be a representative sample of present typical designs.

[69] for an account of the most important achievements on the subject during this period. The present theoretical formulation of the state minimization problem is thus largely satisfying for most purposes. We will see that this is not the case for the state assignment problem.

However, there is at least one point where the SM general theory is defective. The minimized machine, solution of the SM problem, can still be (and it often is) incompletely specified. In this case, some degree of freedom arises in the choice of the state transition and/or output functions to implement. Choosing the best suited functions to use in order to optimize the physical parameters is called the *mapping problem* [59]. This problem is not addressed by the theory. Here, considerations of other synthesis steps, like state assignment, can be advantageous. A first approach to the solution of the mapping problem is provided in [59].

1.2.1.2 State Assignment

Given an FSM symbolic description, the *assignment problem* of FSMs comprises the solution of at least three encoding problems, before producing an implementable design:

- input assignment;
- output assignment;
- state assignment.

Hartmanis and Stearns proposed the first thorough, systematic approach to the FSM assignment problem in [63]. They developed an *algebraic structure theory* of sequential machines, based on the mathematical theory of partitions. Also, they used the *algebra of partition pairs* introduced by Krohn and Rhodes [74] to develop tools for the analysis of the *information flow* in FSMs. Finally, they suggested techniques to synthesize machines with *reduced functional dependence* assignments or *decomposition driven* assignments, based on this algebra. Although this is the best established theoretical approach to the subject, the structure theory is not considered the most suitable one, due to:

1. the computational complexity of implementing it;
2. the fact that reduced functional dependence demands more algebraic structure than most FSMs have [61];
3. the non-existence of a direct correlation between the target cost function, functional dependence minimization, and area/delay characteristics of practical circuits.

As a consequence, the straightforward application of these techniques conducts frequently to poor practical results in state assignment [90]. Yet, it remains a basic reference for further research in the subject.

Among the many practical methods for working out solutions to the SA problem, we can cite the one described by Humphrey that establishes general conditions to respect when looking for a good state assignment without using exhaustive enumeration [67]. The first of these conditions states that

Condition 1.1 (First Condition of Humphrey) *All states that are next states of another, under some input value, must be assigned adjacent binary codes, in order to increase the minimization likelihood in the final combinational part.*

Based on this same argument, Armstrong proposed in [4] an algorithm to solve the state assignment problem for synchronous sequential machines. He was the first to formulate SA as a graph embedding problem, using the gate count of the final implementation as cost function to minimize. Liu proposed a state assignment technique for asynchronous circuits in [79] that also respects Humphrey's first condition. Papachristou and Sarma suggested a state assignment method for synchronous circuits based on the same principles of the Liu method [90]. They obtained an important result for synchronous FSM synthesis, which is the observation that, for these machines, assignments based on the findings of Liu associate a single *Boolean cube* with a set of states whose next state under a given input is unique, and that this cube is disjoint from all other cubes associated with the other next states under the same input. We will see below that the *Liu assignments*, as they are denominated by Unger in [114], are fundamental in modern state assignment methods.

Other authors have used the Hartmanis and Stearns' paradigm, trying to eliminate its weaknesses [70, 47]. Saucier et al [104] proposed artificial intelligence techniques for performing state assignment, while Amann and Baitinger [3] suggested the consideration of lower level structural issues, like the use of counter-based ROM/PLA architectures, to reduce the cost of the FSM implementation. A complete review of the published propositions would be too long to suit this dissertation. We would rather cite Marc Davio's [33] comment regarding the large number of different techniques available today for tackling the SA problem:

The diversity of the approaches encountered so far probably attests our ignorance, or at least the lack of sufficiently deep problem understanding.

Only recently, the first significant improvement to this situation emerged, and this since the advent of the structure theory of Hartmanis and Stearns. The *symbolic optimization method* developed by de Micheli et al [43, 39], provided an alternative paradigm for dealing with various aspects of the encoding problem. Before discussing symbolic optimization, let us recall that the synchronous sequential implementation of an FSM combinational part has as inputs the primary inputs plus the present state feedback lines coming from the state register (i.e. the sequential part). The outputs are the primary outputs plus the next state lines that feed the state register inputs. Thus, solving the input and output assignment problems for the combinational part solves also the state assignment problem.

In a first work [43], de Micheli et al developed a method called *disjoint minimization* that is useful to solve the state assignment problem by reducing it to an *input encoding problem*. The program KISS [42] implements this method. In [39, 40] the method is extended to take into account both input and output influences. The extended version is called *symbolic minimization*[2]. Both methods rely upon a two-step scheme: an encoding independent minimization step generates constraints for a subsequent encoding step. The encoding independent step groups inputs and present states according to the needs of a Liu assignment. In fact, the KISS program output is an assignment of this kind. The most significant finding of de Micheli et al is twofold:

[2] In later works, the denomination symbolic minimization is adopted to design the technique described in the first work. We adopt this later convention.

1. they showed how to compute a tight upper bound for the product term cardinality of the final two-level implementation, from the encoding independent step;
2. they showed that constraining codes to have the shortest possible length in bits is frequently a bad choice if the cost function to minimize is final circuit area.

In these methods, one uses the length of the encoding (number of bits) as cost function to minimize, while trying to respect a set of constraints. If the final encoding satisfies all constraints generated in the first, encoding independent step, the implementation is sure to comply with the predicted product term cardinality upper bound.

However, the outputs influence on symbolic minimization was not fully captured by the techniques in [39, 40]. Devadas and Newton [45] proposed the concept of *generalized prime implicants*, that was used to model more thoroughly the output encoding problem. They developed exact solutions, with very low computational efficiency. In another work, Ciesielski et al [26] suggested exact techniques to generate all possibly useful elementary output constraints, into which the *generalized prime implicants* decompose. The efficiency of the technique is not discussed in the reference. Any method considering the outputs' influence relies upon the generation of a set of potentially useful output constraints, from which a subset is used later. The initial set can be very big, as can be concluded from an analysis of the exact techniques to generate them. In general, the input constraints are considered as the fundamental ones, since they provide bounds on the length of the final encoding, while output constraints are used to achieve further enhancement of the result.

More recently still, the use of the *pseudo-dichotomy* concept provided a tighter link between symbolic optimization and the structure theory. Tracey [111] introduced the pseudo-dichotomy concept to deal with the SA problem in asynchronous sequential circuits. Some modern works [119, 26, 103] have taken advantage of this partition-like structure to represent and manipulate the SA constraints generated by symbolic optimization. Their approach is to translate the constraints to pseudo-dichotomies, and then to solve the associated pseudo-dichotomy covering problem. Prime pseudo-dichotomies are defined, and used to reduce the solution search space for the subsequent covering step. In a newer work, Shi and Brzozowski suggest a greedy technique that uses pseudo-dichotomies, but avoid the generation of primes [106]. Their results compare favorably with previous approaches in terms of both final area and encoding length. Additionally, their implementation runs much faster.

Some methods [4, 47] fix the encoding length to the minimum possible value, and then try to encode the states so as to optimize the resulting combinational part. Methods using symbolic minimization try to reduce the encoding length while respecting the generated constraints.

The symbolic minimization techniques and the derived constrained encoding approach showed that, if area estimate is the cost function to minimize, there are relevant constraints to be considered during state encoding other than minimum code length. Symbolic methods may be subdivided into *(complete) constrained encoding* and *partial constrained encoding*. In complete constrained encodings, the codes generated satisfy *all* SA constraints, and the encoding length is a cost function to minimize. The programs KISS [42] and DIET [119] generate this kind of encoding. Partial constrained encoding consists in preestablishing a code length (minimum or not), and then trying to find an encoding with this code length that satisfies as many constraints as possible. In general, partial constrained encoding gives better results than complete constrained encoding in terms of PLA area, since it provides a better control over the

final code length. The program NOVA [115], for example, uses partial constrained encoding. However, for machines with a relatively low number of constraints, complete constrained encoding performs better, and that is why the program ENCORE [106] allows the choice between these two methods.

In state assignment, there is too vast a number of publications to cover them all in this work. We could not even find a reasonably comprehensive review in the available literature. Limited, yet broad reviews on the subject can be found in [5, 6].

1.2.1.3 Simultaneous State Minimization and State Assignment

We stated, in Chapter 0, that there are two strategies to solve the SM and SA problems, viz. the simultaneous strategy and the serial strategy. Traditionally, one uses the serial strategy. However, some works have proposed the use of the latter to provide a more global view of the FSM synthesis design process. No conclusive result exists, up to now, to show which strategy performs best. Still, we intuitively feel that taking into account the SA constraints while doing state minimization can only further the insight on the characteristics of both problems.

Hartmanis and Stearns studied the relationship between the SA and SM problems in [62, 63], showing that a state minimization phase that does not take into account the subsequent state assignment may obscure some potentially good implementations of the machine. They even suggested the use of *state splitting* techniques for coping with the problem of highly minimized descriptions, where little structural information is available. State splitting is the converse of state minimization, in the sense that it "unreduces" a machine in order to obtain a best realization.

To our knowledge, only three works have suggested the simultaneous strategy to date. In the first of these, Hallbauer [60] proposes a method based on pseudo-dichotomies that avoid races in asynchronous circuits, and which tries to perform state minimization while heuristically reducing the encoding length. The second work is due to Lee and Perkowski [75], and suggests one exact method to tackle synchronous FSMs. Their method employs a branch-and-bound technique to reduce the search in the solution space. In a third work Avedillo [8] presents a heuristic method in which the encoding is generated incrementally, and which may create incompletely specified codes for the states in the original description. The subsequent combinational logic minimization step can, in this way, merge compatible states such that the equivalent of state minimization is performed.

No theoretical findings on the relationship between the SM and SA problems is provided in any of these works. Besides, the method of Hallbauer has not been submitted to benchmark tests, its efficiency being thus hard to evaluate. The Lee and Perkowski's method is feasible only for very small machines (no more than sixteen states). The Avedillo's method has been extensively tested using a subset of the MCNC FSM benchmark set, comprising machines with no more than 32 states. Although reasonably efficient, the results of this method are poorer than those obtained with a serial strategy proposed in the same work.

These facts have driven the ideas behind the work presented here. We wanted to investigate more thoroughly the possibility of providing a theoretical framework in which the SM and SA problems can be conveniently compared, as well as to propose a more efficient method to approach the simultaneous strategy. The advent of high-level synthesis tools imply the occurrence of automatically generated FSMs. Such FSMs are expected to increase the needs for

redundancy removal in the logic level design phase [107].

Let us recall that this work is grounded on a constraint satisfaction formulation of the SM and SA problems. Previous publications have suggested similar formulations for both the SM problem [37] and the SA problem [45]. However, these approaches have always considered only one of the problems at a time. Here we envisage this formulation as a natural way of establishing a relationship between the two problems. Our main objective is to decompose both of them into comparable sets of entities, such that the final solution takes into account the tradeoffs arising between the SM and SA problems. This is possible only if we employ the simultaneous strategy.

1.2.2 The Structure-driven Approach

This is a modern approach to FSM synthesis, derived from the application of the concept of *retiming* [76] to the synthesis of synchronous circuits. The basic idea of retiming is to manipulate a structural description of a synchronous circuit, such as a netlist, as a whole. No previous extraction of the memory elements is done. In this way, we obtain a more thorough optimization of the sequential network. Retiming allows the accomplishment of transformations where registers, as well as gates, are moved, extracted, merged or decomposed.

A first work by Malik et al [80] proposed a limited method of retiming and resynthesis that could optimize circuits containing pipelined sections. Other works generalized the method. In [41], de Micheli extends the multiple level optimization techniques from [19] to the synchronous domain using the retiming concept. At the same time, he provides algorithms based on these operations to reduce the global cycle time of a synchronous circuit. A general theoretical framework for synchronous optimization is proposed in [31] to solve the problem based on *recurrence equations*, but the reported practical results are worse than those obtained with combinational optimization techniques. More recently, Lin [77] proposed a generalization of the well-known *algebraic kernel* concept of combinational optimization [18]. His extension of the combinational techniques based on *synchronous kernels* obtained very good results, in terms both of area reduction and timing performance.

The advantage in the structure-driven approach is that, unlike the behavior-driven approach, area and timing metrics can be directly evaluated at each step of the computation, and not only estimated. Figure 1.1 depicts the corresponding Y-diagram.

1.2.3 Relationship Between Approaches

Although the behavior-driven and the structure-driven approaches are frequently stated as alternative [41], in fact they are not. Instead, we consider them as complementary. With the present trend of using high-level behavioral descriptions as input to design systems [93, 92], it is going to be quite difficult to avoid the use of the FSM behavioral models in a first step. This is due to the fact that direct translation of HDL descriptions into netlists may generate too big initial descriptions. Thus, the behavior-driven approach can produce an intermediate description, which can then be retimed and resynthesized in a subsequent design step.

On the other hand, the pure structure-driven approach can be useful alone in resynthesizing and retiming circuits initially described as a schematic diagram or netlist. This may

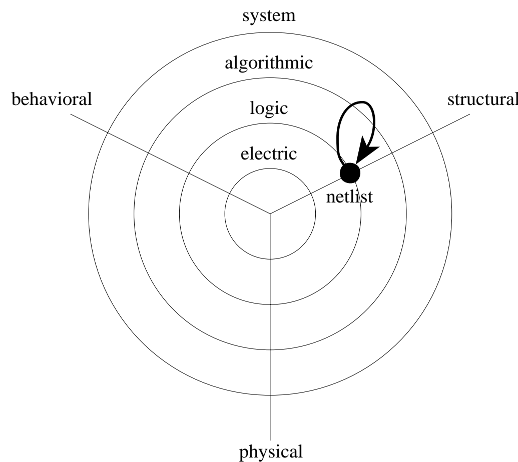


Figure 1.1: Y-diagram for the structure-driven approach to sequential synthesis

happen every time an old design is to be reused or enhanced, as well as on conservative design environments.

1.2.4 Future Approaches

New advances in sequential logic analysis may, in a near future, produce significant changes in the way sequential logic synthesis is performed. These advances rely on two concepts introduced by recent research works. Let us briefly discuss these concepts.

The first concept is an efficient canonical form for representing combinational switching functions by means of binary decision diagrams [2]. This form is called *ordered binary decision diagram* (OBDD), and it was introduced in [23] by Randal Bryant from the Carnegie-Melon University. This canonical form has been successfully used to enhance the performance of combinational and sequential verification algorithms [81, 28]. Although very efficient, OBDDs present two problems. First, we have to find the optimal total ordering of input variables to construct the canonical form from it, together with the function description. Although this problem is NP-complete [50], various works have suggested good heuristic techniques to circumvent the problem, e.g. [51, 24]. Second, there exists a class of functions for which an OBDD has an exponential size on the number of input variables for any total ordering of input variables, as is the case for the integer multiplication function.

The second concept is the use of *implicit* descriptions of FSMs in sequential verification algorithms [10]. Implicit representations of FSMs describe the output and next state functions of a machine by means of well-formed algebraic expressions [35]. They are opposed to *explicit* representations, where these functions are described in the tabular or the equivalent graph forms. Note that implicit representations:

- can be obtained directly from high-level specifications;
- are intrinsically capable of handling functions which are bigger (due to their compactness) than those handled by explicit representations;
- have a direct translation to the OBDD representation.

Due to the above reasons, they may advantageously substitute explicit representations in the behavioral approach. However, this can happen only if implicit representations prove to be practical in solving the traditional problems of sequential synthesis, which is still an open research issue.

Chapter 2

An Introductory Case Study

The goal of the present Chapter is to introduce the subject of this work through an FSM logic synthesis case study. Along with the discussion of the synthesis process applied to the case study, we informally uncover the main concepts to be treated in the remaining of this thesis. The presentation of most formal definitions will take place later, in Chapter 3. Meanwhile, the use of some otherwise inexact terms and concepts will favor the intuitive understanding of the problems; these terms will be presented here inside single quotes (‘’) when they are referred to for the first time, instead of the ***bold slanted*** typeface we adopt in the thesis to introduce new terms. This convention is used to avoid confusion when we later define the terms formally.

Section 2.1 presents the case study. It begins with an introduction of the SM and SA problems using a hypothetical serial strategy to solve them. The Section includes a brief discussion, pointing out some deficiencies of this and other serial strategies, resulting from assumptions that have to be made when working in the logic level of abstraction. In the same Section, three distinct practical solutions for the case study are displayed. At the end of the Chapter, Section 2.2 unveils some relevant questions that will be answered alongside the rest of this volume.

2.1 The Case Study

The FSM we use here as case study comes from the already mentioned MCNC benchmarks [118], and is named BEECOUNT. Table 2.1 displays the behavior of this synchronous FSM. As usual in synchronous flow tables, columns and rows are associated with input values and states, respectively. As is the case for all MCNC FSM benchmarks, the inputs and outputs are already binary encoded. States are represented symbolically as integers. BEECOUNT has eight inputs (encoded in three bits), four outputs (encoded in four bits), and seven states. Each entry of the flow table describes a possible transition of the machine, where the column and row specify the present input and state of the FSM, and the contents of the entry comprise the next state and the output associated with the transition, respectively. Next state and/or output values may be unspecified in some entries, which constitutes a don’t care condition for the transition and/or output functions of the FSM, and this is indicated by the use of the “-” sign.

Now, we want to study the relationship between the SM and SA problems using this FSM behavior description. The objective of doing so is to show that it is possible to take advantage

Table 2.1: Flow table describing the behavior of the FSM BEECOUNT

state \ input	000	100	010	110	001	101	011	111
0	0,0101	1,0101	2,0101	-,-	0,1010	0,1010	0,1010	0,1010
1	0,0101	1,0101	0,0101	3,0101	0,1010	0,1010	0,1010	0,1010
2	0,0101	0,0101	2,0101	5,0101	0,1010	0,1010	0,1010	0,1010
3	-,-	1,0101	4,0101	3,0101	0,1010	0,1010	0,1010	0,1010
4	0,0110	-,-	4,0101	3,0101	0,1010	0,1010	0,1010	0,1010
5	-,-	6,0101	2,0101	5,0101	0,1010	0,1010	0,1010	0,1010
6	0,1001	6,0101	-,-	5,0101	0,1010	0,1010	0,1010	0,1010

of their relationship during the search for an optimal two-level implementation of FSMs. Translated to practical considerations for this case study, and imposing all restrictions we stated in Section 0.2, this means we want to answer the following question: "How can we optimally assign codes to the states of machine BEECOUNT?" Or, stated in more detail: "How can we obtain a state encoding for BEECOUNT such that a two-level circuit implementing it will be optimum with regard to some cost function taking into account the three minimization criteria of occupied area, propagation delay and power consumption?" We now examine the conditions that permeate through this question, and that we repute important to consider during the encoding. We begin with the conditions connected to the SM problem only.

2.1.1 SM-Related Considerations

First, we note that there are no two identical rows in the flow table for the BEECOUNT FSM. However, consider states 3 and 4. They have identical rows, except for the entries corresponding to the inputs 000 and 100. Because the difference is due only to the existence of unspecified entries in these columns, we may change the unspecified entries to contain any state/output pair, without modifying the 'specified' behavior. In particular, the entry corresponding to state 3 and input 000 may hold the same contents as the entry corresponding to state 4 and input 000, while the entry corresponding to state 4 and input 100 may hold the same contents as the entry corresponding to state 3 and input 100. If we perform both changes in the flow table, the first two rows become identical, and we say that states 3 and 4 are then 'equivalent'. If a pair of states can be made equivalent without changing the specified behavior, we say they are 'compatible'.

With the notions of equivalence and compatibility we associate the existence of redundancy in the machine, because they imply that there are two states we cannot 'distinguish', as long as we limit attention to the external (i.e. input/output) behavior of the FSM. Since this is the behavior addressed by FSM users, any transformation of the machine specification that does not affect it is valid.

The concept of FSM valid transformations is *very important*, since these are the only transformations allowed during the FSM synthesis process. In the scope of this work, we are interested in dealing with 'encoding transformations' only, and the central concept here is that of 'valid state assignment' to be introduced in Chapter 5. In fact, generalizing this last concept will allow the simultaneous consideration of both SM and SA problems.

Back to our example, after making the states 3 and 4 equivalent we can simply merge the flow table rows associated with them into a single one, thus reducing the total number of states of the FSM. Additionally, we must provide a new designation for the row resulting from the merging process (for instance, $\{3,4\}$), and change every reference to any of the original states by this designation. The result appears in Table 2.2. The merging process may produce a smaller combinational part, because every encoded output[1] potentially depends on less input values, and because the lower bound on the number of bits needed to encode the outputs may become smaller. Nonetheless, the final functions are 'denser', since the number of unspecified entries in the flow table is reduced (in our example it passes from 5 to 3), which can render the minimization of the combinational part harder to perform.

Table 2.2: Flow table for the BEECOUNT FSM after merging states 3 and 4

state \ input	000	100	010	110	001	101	011	111
0	0,0101	1,0101	2,0101	-,-	0,1010	0,1010	0,1010	0,1010
1	0,0101	1,0101	0,0101	{3,4},0101	0,1010	0,1010	0,1010	0,1010
2	0,0101	0,0101	2,0101	5,0101	0,1010	0,1010	0,1010	0,1010
{3,4}	0,0110	1,0101	{3,4},0101	{3,4},0101	0,1010	0,1010	0,1010	0,1010
5	-,-	6,0101	2,0101	5,0101	0,1010	0,1010	0,1010	0,1010
6	0,1001	6,0101	-,-	5,0101	0,1010	0,1010	0,1010	0,1010

We may generalize the concepts of state pair equivalence (resp. compatibility) and state merging, to sets with an arbitrary number of states. A set of states is called an 'equivalent' (resp. 'compatible') if its elements are pairwise equivalent (resp. compatible). Back to the original flow table, we may depict the compatibility relations within the machine using a graph, called 'merge graph'. In this graph, each distinct vertex represents a distinct FSM state, and an edge connects a pair of nodes if and only if the two associated states are compatible. Labeled edges correspond to 'conditionally compatible' pairs of states, the label representing a set of pairs of states that need to be compatible so that the states associated to the nodes be themselves compatible. The merge graph for BEECOUNT is showed in Figure 2.1. Note, for instance, that states 1 and 0 are compatible if states 0 and 2 are also compatible, and vice versa. Thus, they are conditionally compatible. However, states 1 and 2 are not compatible, even though they assert no conflicting output value for any input value. Their incompatibility comes from the fact that there is at least a sequence of inputs (for example 110 followed by 100, followed by 000) that may distinguish if the FSM is initially in one of these two states.

In the merge graph, we point out that the *maximal complete subgraphs* [27] (also called *maximal cliques*) represent the compatibles of maximum size that contain some state. These are accordingly named 'maximal compatibles'. For the BEECOUNT FSM, the set of all maximal compatibles is:

$$\{\{0,1\}, \{0,2\}, \{3,4\}, \{5,6\}\}.$$

Noting that the particular external behavior of every state in the original description must be contemplated in the FSM description after a set of state merging steps, we may informally

[1] Recall that combinational part outputs comprise the output lines *and* the next state lines. The same comment is valid *mutatis mutandis* for the inputs.

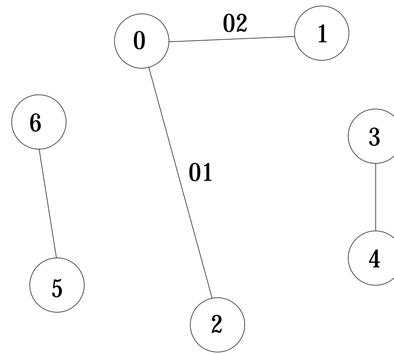


Figure 2.1: Merge graph for FSM BEECOUNT

enunciate the goal of state minimization[2]: *The solution of the* SM *problem can be achieved if we compute a set of compatibles with minimum cardinality, such that each state of the original* FSM *belongs to at least one of its elements, i.e. if we can find a 'minimum cover of compatibles'.* A maximally reduced equivalent FSM can then be constructed from this set, by associating a state to each compatible, defining a new flow table according to the row merging rule introduced before. An inspection of the merge graph of our example FSM allows the extraction of all 'valid covers' for this example. A cover is valid if all conditions imposed by compatible pairs of states in it are respected. It is thus called a 'closed cover' of compatibles. For our example, there are only 8 such covers, namely:

1. {{0}, {1}, {2}, {3}, {4}, {5}, {6}};
2. {{0}, {1}, {2}, {3,4}, {5}, {6}};
3. {{0}, {1}, {2}, {3}, {4}, {5,6}};
4. {{0}, {1}, {2}, {3,4}, {5,6}};
5. {{0,1}, {0,2}, {3}, {4}, {5}, {6}};
6. {{0,1}, {0,2}, {3,4}, {5}, {6}};
7. {{0,1}, {0,2}, {3}, {4}, {5,6}};
8. {{0,1}, {0,2}, {3,4}, {5,6}};

First, we observe that in no closed cover representation we may have the compatible {0,1} without having the compatible {0,2}, since the former implies the latter (and vice versa, since the latter also implies the former). Second, we notice that there is only one *minimal* closed cover, formed by the set of maximal compatibles, but this does not account for the general case. The set of all maximal compatibles *always* represents a cover of the state set, but it is not necessarily minimum, nor unique. Indeed, there is no need for a minimum cover representation to have any maximal compatible at all [72]. However, any compatible is contained in at least one maximal compatible. Thus, the set of all maximal compatibles is a 'reasonable' starting point for the search of the minimum covers, since we can generate any minimum cover from it.

[2]This statement is not only informal, but also incomplete, but will serve our purposes in this Chapter.

A question arises as to how we can choose *the* best cover with regard to our minimization criteria. In this particular example, the minimum cover is unique, and gives a good approximation of the best implementation. However, solving the SM does not necessarily guarantee the satisfaction of any of the three criteria of minimum area, propagation delay and dissipated power. More important, since the minimum closed cover is not unique in general, there must be ways to choose among all minimum closed covers. To select the best one, we may consider the conditions imposed by the state assignment step.

2.1.2 SA-Related Considerations

We have mentioned in Section 1.2.1.2 how the first condition of Humphrey (Condition 1.1) has led to the development of modern state assignment methods. In particular, the symbolic minimization method [43] relies upon this condition to generate constraints on the assignment of state codes for a PLA implementation of an FSM combinational part. We highlight below the functioning of the method as applied to machine BEECOUNT. The objective is to show how the 'input constraints' may help us in the search for the optimum implementation of the FSM.

It is well known [16] that one of the most critical factors influencing the cost of a two-level implementation is its 'product term' cardinality, which is associated with the concept of 'cubes'. Thus, an FSM state encoding method viewing the obtainment of optimal area results for two-level implementations should be able to reduce the product term cardinality.

The symbolic minimization method identifies subsets of entries in the flow table that are identical in some aspect (next state, output or both) and whose corresponding inputs and present states, together, 'can' form a 'valid' product term in the final PLA. The method looks for a set of entry subsets that is optimal, in the sense that the least possible number of such subsets is obtained, while ensuring that every relevant entry is included in some subset. For machine BEECOUNT, the result of applying this procedure to the original flow table of Table 2.1 appears in Figure 2.2. From these subsets the method generates the constraints to be respected by the state encoding, as we will discuss next.

state \ input	000	100	010	110	001	101	011	111
0	0,0101	1,0101	2,0101	-, -	0,1010	0,1010	0,1010	0,1010
1	0,0101	1,0101	0,0101	3,0101	0,1010	0,1010	0,1010	0,1010
2	0,0101	0,0101	2,0101	5,0101	0,1010	0,1010	0,1010	0,1010
3	-, -	1,0101	4,0101	3,0101	0,1010	0,1010	0,1010	0,1010
4	0,0110	-, -	4,0101	3,0101	0,1010	0,1010	0,1010	0,1010
5	-, -	6,0101	2,0101	5,0101	0,1010	0,1010	0,1010	0,1010
6	0,1001	6,0101	-, -	5,0101	0,1010	0,1010	0,1010	0,1010

Figure 2.2: One optimal set of entry subsets for machine BEECOUNT

Let us first explain what we mean by 'valid' product term. Consider the subset of entries

associated with the second input column (100) and the three lines corresponding to states 0,1, and 3 in Figure 2.2. To implement this partial behavior optimally, we need only one product term in the encoded two-level implementation, because all three entries share a single output pattern. The product term has to account for all three entries associated with the subset, and for no other entry in the flow table. To do so, it suffices to impose that the binary codes we assign to states 0, 1, and 3 be such that the 'smallest cube' containing the codes (also denominated the 'supercube' of the codes) of these three states does not contain the code of any other state in the machine.

We may informally show the soundness of this reasoning by contradiction. Suppose that the supercube of the codes contains codes of other states as well. Remember that a product term in a PLA produces a single particular output pattern. In our example, using such a supercube as part of the product term generating the output pattern unique to the entries of the subset, would associate this same pattern with states outside the subset. This may indeed change the specified behavior of the machine, and must thus be avoided. For example, if we assign code 0011 to state 0, code 0001 to state 1, and code 0000 to state 3, the smallest binary cube accounting for these states is $00--$. This encoding allows that a single PLA product term ($10\ 00--$) generate the entry subset in question if the code 0010 is not assigned to any state contained in the subset $\{2, 4, 5, 6\}$. In this case, $10\ 00--$ would be a 'valid' product term.

As long as the conditions imposed by every subset of entries are respected during the state encoding step, we may ensure that the final PLA will have at most as many product terms as there are subsets in the table. The goal of the state encoding using symbolic techniques becomes then, as we have already cited in Section 1.2.1.2, the search for either:

- the minimum length encoding that respects all constraints, if we use complete constrained encoding;
- the encoding with a given length (often, the absolute minimum necessary) that satisfies the largest number of constraints, if we use partial constrained encoding.

The satisfaction of the constraints we have just presented coerces the encoding of states, so that the use of such codes in the generation of outputs of the FSM combinational part (i.e. as inputs to this part) leads to a minimized machine. They are accordingly called 'input constraints', and each of them is represented as the subset of present states associated with some subset of entries encircled in Figure 2.2. The example constraint we used to explain the concept of valid product term is then represented by the state set $\{0, 1, 3\}$, and satisfying it implies assigning codes to states 0, 1 and 3 such that the supercube of these three codes does not intersect any state code other than those.

Now, we come back to the selection of the best two-level PLA implementation for BEECOUNT, started in the previous Section. We have eight distinct closed cover representations, each corresponding to a distinct FSM structure that may replace machine BEECOUNT. If we assume that the symbolic minimization method leads to good solutions for the SA problem, we may evaluate which of the covers leads to the best FSM implementation as follows.

1. generate all machines associated to each closed cover representation;
2. submit each machine to the symbolic minimization process;

3. generate an encoding for each machine by respecting the conditions imposed by the respective symbolic minimization;
4. in each machine description, substitute each state symbol by the associated state code and submit the 'encoded' machine to logic minimization, obtaining a final two-level logic implementation of the FSM.

The best solution will be that associated with the two-level logic implementation presenting the best area, delay and power characteristics.

2.1.3 Logic Level Assumptions

Most of the available techniques to solve the SM and SA problems rely upon a few basic assumptions. The first one is that state minimization leads to smaller FSM implementations. However, as we pointed out in Section 2.1.1, state minimization produces denser machines, which are harder to minimize. We will see experimentally, in Part V, that performing state minimization to its maximum extent may lead to larger PLAs than performing no state minimization at all, although this is not the general case. The second common assumption is that to reduce the area of the final PLA after state assignment, we have to reduce the code length to its minimum. Again, in Part V we will find machines for which a code length two to three times larger than the minimum necessary lead to a final PLA smaller than those produced by "good" minimum length encodings.

On the other hand, the two above assumptions rely on the fact that smaller PLAs are preferable to bigger ones. But this simplification does not account for at least three facts.

The first one is the use of topological optimization methods in low-level synthesis. In fact, topological methods, comprising PLA partitioning [68] and simple [58] and/or multiple [38] folding, capitalize on the initial PLA sparsity to attain silicon surface gains from 20 to 70% over the original area estimate [38].

The second fact is that the propagation delay time in PLAs depends strongly on the occupied surface, as well as on the form factor, which should ideally be close to the unity. If topological optimization is overlooked, the form factor is determined after state assignment, and its optimization imposes a trade-off between the code length and the number of products. This trade-off could be accounted for during the state assignment step.

The third fact is the consideration of the power dissipated by the module, which is rarely addressed at the logic level. At this level of abstraction, however, we may roughly estimate power consumption based on the number of transistors of the final PLA. In the last Section of this Chapter, these considerations will be used to direct the discussion about useful cost functions to use during the search for a solution to the SM and SA problems.

In Section 2.1.2, we have noticed the existence of a relationship between the SM and SA problems, namely that the constraints derived from the SA problem can be used to evaluate the multiple minimum solutions arising in the SM problem. In so doing, we are able to eliminate some of these solutions, i.e. those identified as leading to bad implementations according to our final minimization criteria. The method suggested above is useful to introduce the problem we want to treat, but it is not a practical one to apply to 'large' FSMs, since it requires the generation of all possible solutions of the SM problem, followed by the selection of the best one

among these. However the obtainment of a solution to the SM problem is already NP-complete [95]. We refer to Zahnd [122] for an exact method describing how to engender all “irredundant covers” of compatibles. By irredundant cover, Zahnd means a cover where no proper subset of it is also a cover. In particular, all minimum covers are irredundant.

In Chapter 6 we will disclose other useful relationships between the SM and SA problems, while in Parts III and IV we propose an efficient method to cope with the complexity of both problems at once. In the next Section, we present three solutions for the SA problem of machine BEECOUNT. These solutions will be confronted with respect to the satisfaction of the optimization criteria of area, propagation delay, and power consumption, in view of all cost functions suggested above.

2.1.4 Three State Assignment Solutions for Machine BEECOUNT

From the discussion in the last Section, we note that state encoding could hardly be classified as a one-solution problem, since many cost functions may interfere in the evaluation of the quality of a solution to this problem. Often, these cost functions conflict with each other, thus making optimality strongly dependent upon which cost functions are retained to compute the quality. Table 2.3 shows three valid state assignments for the states of machine BEECOUNT. Again, the assignments are valid in the sense that all three preserve the external behavior predicted by the initial specification.

Table 2.3: Three valid state assignments for machine BEECOUNT

state	NOVA code	ASSTUCE code	compatible	STAMINA + NOVA code
0	000	0-	{0,1}	01
1	101	00	{0,2}	00
2	100	01	{3,4}	10
3	110	10	{5,6}	11
4	010	10		
5	011	11		
6	111	11		

We obtained the encodings in Table 2.3 by running either an SA program, or a combination of SM and SA programs.

The first column, labeled NOVA **code**, presents an encoding obtained with NOVA [116], an SA program using a partial constrained encoding approach. It is in fact the best performing program we could find using this approach. The second column, labeled ASSTUCE **code**, presents an encoding obtained with ASSTUCE, the program developed in the scope of this thesis. ASSTUCE may employ either complete or partial constrained encoding approaches. The solution presented in the Table resulted from the use of the partial approach, to allow a fair comparison with NOVA.

Both programs were asked to assign minimum length codes to the machine, but the length of the respective codes are distinct. The explanation for this fact relies in the fact that NOVA is not capable of identifying the existence of compatibility classes in the original description. It thus assigns a code of length 3 to BEECOUNT, which is the minimum for a machine with

7 states. ASSTUCE, on the other hand, can identify the existence of such classes, since it implements a simultaneous strategy to solve the SM and SA problems, and is thus able to assign codes to BEECOUNT so that the equivalent of state minimization can be performed during the logic minimization phase. Both, NOVA and ASSTUCE assign codes to the states of the original machine description.

A third encoding appears in the column labeled STAMINA + NOVA **code**. This last encoding was obtained according to a serial strategy to solve the SM and SA problems. First, an SM program, STAMINA [59] was run on BEECOUNT, generating an exactly minimized version of this machine, where states correspond to the compatibles shown in the Table. After this, NOVA was used to assign the minimized machine, obtaining a code of length 2, the minimum for a 4-state FSM.

Details about the method employed by ASSTUCE will be found in Chapter 12, while the program implementation will appear in Chapter 14.

A comparison of the three solutions is useful to reveal the main distinctions between our approach and any other constrained encoding method we can find in the available literature. To our knowledge, all constrained encoding programs to date generate encodings that are *injections* from the FSM state set into a subset of the binary codes of a given length. ASSTUCE, on the other hand, engenders state codes that are not necessarily distinct for distinct states, and which additionally can be incompletely specified. This is fundamental to allow full-fledged performance of a simultaneous strategy to solve the SM and SA problems for incompletely specified FSMs.

To justify this, consider states 0,1 and 2 of machine BEECOUNT. In order to encode these states so that the minimum closed cover of BEECOUNT can be obtained, we have to guarantee that once encoded, these states lead to the compatibles $\{0, 1\}$ and $\{0, 2\}$. To do so, the code assigned to state 0 must be 'compatible' with the codes assigned to states 1 and 2. However, the codes for states 1 and 2 cannot be compatible, because these are incompatible states. One way to maintain the correct external behavior, while respecting the compatibility relations, is to use an incompletely specified code for state 0, and arrange for states 1 and 2 to have incompatible codes, but both compatible with the code for state 0. ASSTUCE assigns codes in this way: for state 0, it assigns the code 0−, and for states 1 and 2 the codes 00 and 01, respectively.

We also advance that ASSTUCE does not perform state minimization explicitly, i.e. we cannot in general devise the final compatibles from the encoding, since codes are incompletely specified. State minimization is done by the combinational logic minimization tool, which groups compatible state codes, but only if this leads to a better implementation, according to its less abstract minimization criteria.

We have used the three state assignments depicted in Table 2.3 to obtain the corresponding minimal two-level combinational part PLAs of the associated FSM implementations. After substituting the encoding in the corresponding symbolic description, we obtained an encoded FSM, represented as a PLA personality matrix. This personality matrix was submitted to ESPRESSO, which generated the minimized personality matrices appearing in Figure 2.3.

The personality matrix in the ESPRESSO format has two sides: **inputs**, including the primary input columns and the present state columns, and **outputs**, including the next state columns and the output columns. Each 1 (resp. 0) in the **inputs** represents a transistor in the AND plane of the PLA, connected to the value of one primary input or present state (resp. to its

```
Three FSM Combinational Parts PLAs after Encoding and Minimization

        NOVA                   ASSTUCE            STAMINA+NOVA
inputs    outputs       inputs  outputs       inputs  outputs
010 --0  010 0000       000 10  00 0110       000 11  00 1001
100 --1  100 0000       000 11  00 1001       1-- 00  01 0000
0-0 111  000 1001       1-0 11  11 0000       -10 10  10 0000
-00 110  000 0110       110 --  10 0000       000 10  01 0110
100 -0-  001 0000       -10 10  10 0101       1-0 11  11 0000
110 0--  001 0000       --1 --  00 1010       110 --  10 0000
-10 0--  010 0000       0-0 0-  01 0101       -00 0-  01 0101
1-0 -0-  100 0000       -10 -1  01 0101       --1 --  01 1010
-10 1-0  100 0101       1-0 --  00 0101       -10 --  00 0101
1-0 -11  011 0101                             100 --  01 0101
--1 ---  000 1010
--0 -0-  000 0101
--0 0--  000 0101
```

Figure 2.3: Three Combinational Part PLAs for machine BEECOUNT

complement). Each 1 in the **outputs** corresponds to a transistor in the OR plane of the PLA. For detailed information about the ESPRESSO format refer to Appendix A.

A visual inspection of Table 2.3 and Figure 2.3 provides us already with qualitative information about the results of the three solutions. Table 2.4 shows some quantitative data available from these Figures.

Table 2.4: Quantitative comparison for three PLA implementations of machine BEECOUNT

Parameter \ Solution	NOVA	ASSTUCE	STAMINA + NOVA
Code length	3	2	2
Number of states	7	4	4
Number of product terms	13	9	10
Number of transistors	68	50	54
Area estimate	247	144	160
Sparsity estimate	72.47	65.27	66.25
Form factor estimate	1.46	1.78	1.6

It is interesting to note that the ASSTUCE solution has a bounded but unspecified number of states. We know only that states 3 and 4 (resp. 5 and 6) are implemented always as a single one since their codes are both identical and contain no don't care. The final implementation may even choose to implement states 0 and 1 and/or 0 and 2 as distinct ones, since state 0 has an incompletely specified code. This freedom is absent in other approaches, and it implies the observation that the number of states of the final implementation is completely irrelevant, as long as the correctness of the FSM's external behavior is guaranteed. The other quantitative data show that, although ASSTUCE does not effectuate explicit state minimization, it has obtained an overall result that is not only better than ordinary state assignment programs results, but also better than the result obtained by a serial strategy. One explanation for this fact is that ASSTUCE allows more degrees of freedom to subsequent synthesis steps, due to the use of codes

that are specified only where necessary. An extensive comparison of various approaches to the solution of SM and SA problems will be provided in Part V.

2.2 Discussion

The case study showed that to develop a method leading to the solution of both SM and SA problems, we have to carefully study their relationship, in order to access cost functions that guarantee "optimum" FSM implementations. The main point to be clarified here is *how* we define an optimum FSM implementation. Ideally, such an implementation has the least area, the least delay, and the least power consumption. In practice, these often conflicting goals must be traded-off during state minimization and encoding. In the rest of this work, we search for techniques to answer simultaneously the following set of questions:

> Given an FSM symbolic specification, how can we assign codes to its states such that we achieve:
>
> 1. the least number of distinct codes, which allows for extensive state minimization, and reduces the final PLA size?
> 2. the most sparse codes, to permit the greatest possibility of minimization for the final combinational part, allowing enhanced topological minimization, and reduced power consumption?
> 3. codes with the least length in bits, to obtain small PLAs, and to generate the least number of outputs for the combinational part of the machine?

To engender short length codes, we need to minimize the 'size' of the cubes used to encode states. To allow sparse codes to be obtained, we must maximize the size of the same cubes, which conflicts directly with the third item above. To reduce the number of distinct codes, any compatible pair of states should receive equal, or at most compatible codes, but this again conflicts with the requirement for the last item, because the use of compatible codes imply that the code length may have to be increased. Sparse codes favor topological minimization, but may still produce larger final areas, if the code length grows without bounds to favor the sparsity. The fastest PLAs have a form factor close to unity. However, the PLA form factor depends on the number of inputs, outputs and product terms directly, and only indirectly on the structure of the state encoding. Thus, it is quite hard to control the form factor during state minimization/encoding. Power consumption is also hard to estimate, even though we know that the sparser the PLA, the less it consumes, given the same area and the same form factor.

Chapter 3

General Definitions

In this Chapter, we define some basic concepts needed in the rest of this dissertation. We assume the familiarity with the elementary definitions of finite sets theory, lattices and enumeration orders, as well as with the *Cartesian product* definition.

The concepts introduced herein are related to discrete and switching functions, and to finite state machines. They do not constitute a complete review of any of these subjects, and they were based in the terminology of some classical textbooks. Rutherford presents a comprehensive set of algebraic terms definitions in [102]. Discrete and switching functions receive an in-depth formal treatment by Davio, Deschamps and Thayse in [35]. On the other hand, our formal concepts regarding finite state machines were based on the expositions of Zahnd in [122] and Kohavi in [72]. The notation we use is distinct in minor points from that proposed in each of the cited works, due to the coherence requirements across the various domains. Although most concepts presented are applicable to sets in general, any mention to sets below implies finite, non-empty sets, except when explicitly stated otherwise.

The next Section introduces discrete functions, and defines formally some terms abundantly used in next Chapters. Section 3.2 defines FSMs, while Section 3.3 discusses the compact representation we will use to describe discrete functions, namely the *cube tables*.

3.1 Binary Relations and Discrete Functions

Discrete functions are the fundamental mathematical concept we will later use to formalize the SM and SA problems. Because functions derive directly from the binary relation concept, this latter is the starting point for the definitions, preceded by a brief introduction of some set notations we adopt. The definitions in this Section were adapted from [35], except when indicated otherwise.

Definitions 3.1 (Set related definitions) *Given a set S, $\mid S \mid$ denotes the number of elements in S, or the* **cardinality** *of S. The notation $\mathcal{P}(S)$ stands for the set of all subsets of S, also called the* **powerset** *of S. The* **empty set** *is noted below by the symbol $\emptyset$.*

Definitions 3.2 (Binary relation) *A* **binary relation** *from a set S to a set T, is a triple $\langle S, T, r \rangle$, where*

1. *S is the* **domain** *of r, noted $dom(r)$;*

2. *T is the* **codomain** *of r, noted $cod(r)$;*

3. *r is a subset of the Cartesian product $S \times T$, called* **graph** *of the relation.*

The graph of the relation is noted $r : S \longrightarrow T$. When no confusion can arise, the binary relation $\langle S, T, r\rangle$ is represented by its graph r. A binary relation associates with every element s of S, a subset of T, noted $r(s)$, and called the **image** *of s by r. The image of the subset S' of S by r is defined by*

$$r(S') = \bigcup_{s \in S'} r(s).$$

A binary relation of the type $\langle S, S, r\rangle$ is called a **binary relation on a set**.

Properties 3.1 (Binary relations) *It is useful to describe some of the properties that a binary relation $\langle S, T, r\rangle$ may enjoy:*

1. *it is* **completely specified** *if $r(s) \neq \emptyset$, for every $s \in S$;*

2. *it is* **onto** *if $r(S) = T$;*

3. *it is* **functional** *if $|\, r(s)\,| \leq 1$, for every $s \in S$;*

4. *it is* **one-to-one** *if for every two distinct elements $s \in S, s' \in S$ we have $r(s) \cap r(s') = \emptyset$.*

Properties 3.2 (Binary relations on a set) *The special case of binary relations on a set $\langle S, S, r\rangle$ may have, in addition, other properties:*

1. *it is* **reflexive** *if $s \in r(s)$, for every $s \in S$;*

2. *it is* **symmetric** *if $s \in r(t) \Longrightarrow t \in r(s)$, for every $s \in S, t \in S$;*

3. *it is* **antisymmetric** *if $s \in r(t) \wedge t \in r(s) \Longrightarrow s = t$, for every $s \in S, t \in S$;*

4. *it is* **transitive** *if $s \in r(t) \wedge t \in r(u) \Longrightarrow s \in r(u)$, for every $s \in S, t \in S, u \in S$.*

A reflexive and transitive binary relation is called a **preorder relation**. *An antisymmetric preorder relation is an* **order relation**. *A symmetric, reflexive binary relation is a* **compatibility relation**. *A transitive compatibility relation is an* **equivalence relation**.

Definition 3.3 (Cover) *Given two sets S and T, a binary relation $\langle S, T, c\rangle$ is called a* **cover** *of T iff*

$$\bigcup_{s \in S} c(s) = T.$$

A cover is thus an onto binary relation. The images $c(s)$ of the elements $s \in S$ are called the **classes** *of c.*

Definition 3.4 (Partition) *Given two sets S and T, a binary relation $\langle S, T, \pi \rangle$ is a* **partition** *of T iff it is onto and one-to-one. Stated less abstractly, $\langle S, T, \pi \rangle$ is a partition iff it is a cover of T and*

$$\forall(s \in S, s' \in S), \quad s \neq s' \Longrightarrow \pi(s) \cap \pi(s') = \emptyset.$$

The image $\pi(s)$ of an element s of S is called a **block** *of partition π. Since π is also a cover, its blocks are also called* classes*. A partition can then be defined as a cover where the classes are mutually disjoint.*

Definitions 3.5 (Function) *A* **partial mapping** *or* **partial function** *is a functional binary relation. A* **mapping***, or* **complete function** *is a functional and completely specified binary relation. A* **surjection** *is an onto mapping. An* **injection** *is a one-to-one mapping. A* **bijection** *is a one-to-one and onto mapping. Where no confusion arises, the term* **function** *stands for the more general partial function definition.*

Notice that the definition of complete function comes as a special case of the partial function definition.

We are now able to define discrete functions, which is the class of functions more relevant to the present work.

Definition 3.6 (Discrete function) *A function*

$$f : S \longrightarrow L$$

is a **discrete function** *when S and L are finite, non-empty sets.*

In many cases, the domain of a discrete function is the Cartesian product of n finite sets S_i:

$$S_{n-1} \times \ldots \times S_1 \times S_0 = \mathop{\times}_{i=n-1}^{0} S_i.$$

If $S_{n-1} = \ldots = S_1 = S_0$, we use the exponential notation S^n as a shorthand for $\mathop{\times}_{i=n-1}^{0} S_i$. The discrete function

$$f : \mathop{\times}_{i=n-1}^{0} S_i \longrightarrow L$$

is then represented by $f(\mathbf{x})$, with $\mathbf{x} = (x_{n-1}, \ldots, x_1, x_0)$, where each variable x_i takes its value from the set S_i.

Definition 3.7 (General discrete function) *A discrete function of the form*

$$f : \mathop{\times}_{i=n-1}^{0} S_i \longrightarrow L^m$$

is called a **general discrete function***. Function f is interpreted as a set of discrete functions of the form*

$$f_j : \mathop{\times}_{i=n-1}^{0} S_i \longrightarrow L, \quad 0 \leq j \leq m-1.$$

As a consequence of the last definition, all properties and results obtained for discrete functions are directly applicable to general discrete functions, since general discrete functions *are* discrete functions.

Some particular kinds of discrete functions deserve special attention in the scope of the present work. We introduce first the notion of *sequences*, a concept that has a close connection with the modeling of the temporal behavior of sequential circuits. Our definition is borrowed directly from Zahnd [122], with only minor notational differences.

For every integer $n \geq 1$, we designate by $\mathbf{n}$ the set of integers k such that $1 \leq k \leq n$. Thus,

$$\mathbf{1} = \{1\},\ \mathbf{2} = \{1,2\},\ \mathbf{3} = \{1,2,3\},\ \text{ etc.}$$

Definitions 3.8 (Sequence) *Let A be a finite non-empty set and n be an integer such that $n \geq 1$. The discrete complete function*

$$x : \mathbf{n} \longrightarrow A$$

is called a **sequence** *of length n over A. Sequences like x above are represented by an expression with the generic form $[x(1)x(2)\dots x(n)]$, which is called a* **word**[1] *of length n over A. Alternatively, the shorthand notation $[x(1,n)]$ is used.*

A finite, non-empty set is referred to as an ***alphabet***, when the need to consider sequences over this set arises. The elements of the alphabet may accordingly be called ***letters***. The set of all sequences of length n over an alphabet A is designated here by $A^{[n]}$. The set of sequences of arbitrary length over the alphabet A, i.e. the infinite set $A^{[1]} \cup A^{[2]} \cup A^{[3]} \cup \dots$, is noted A^+. For instance, if $A = \{a, b\}$:

$$\begin{aligned} A^{[1]} &= \{[a],[b]\} \\ A^{[2]} &= \{[aa],[ab],[ba],[bb]\} \\ A^{[3]} &= \{[aaa],[aab],[aba],[abb],[baa],[bab],[bba],[bbb]\} \end{aligned}$$

Definition 3.9 (Integer function) *An* **integer function** *is a function*

$$g : \mathop{\times}_{i=n-1}^{0} \{0,1,\dots,m_i - 1\} \longrightarrow \{0,1,\dots,r-1\}.$$

Evidently, integer functions are discrete functions. Additionally, observe that any discrete function $f : \mathop{\times}_{i=n-1}^{0} S_i \longrightarrow L,\ \mid L \mid = r$, can be expressed as an integer function, by defining an adequate set of injections as follows:

1. define an injection of the set S_i onto the set of integers $\{0,1,\dots,m_i - 1\}$, for $i = 0,1,\dots,n-1$;
2. define an injection of the set L onto the set of integers $\{0,1,\dots,r-1\}$.

[1] The notation used to represent sequences is coherent with the *value vector* representation of discrete functions, to be introduced in Section 3.3.

Definition 3.10 (Binary function) *A* ***binary function*** *is an integer function of the form*

$$f: \bigtimes_{i=n-1}^{0} \{0, 1, \ldots, m_i - 1\} \longrightarrow \{0, 1\}.$$

A binary function is thus a *two-valued* function of *multiple-valued or multivalued* variables.

Definition 3.11 (Switching function) *A* ***switching function****, also called* ***Boolean function****, is an integer function*

$$f: \{0, 1\}^n \longrightarrow \{0, 1\}.$$

The set $\{0, 1\}$ will appear very often in the discussion of subsequent Chapters. We shall accordingly denote it by the shorthand $\mathcal{B}$ in what follows. The above definitions are already sufficient to formally introduce FSMs, our main concern in this Chapter. Other definitions related to discrete functions will be introduced later, in Section 3.3, during the discussion on discrete functions representations.

3.2 Finite Automata and Finite State Machines

Definition 3.12 (Finite automaton) *A* ***finite automaton*** *is an algebraic structure of the form* $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$ *where:*

1. $I = \{i_{p-1}, i_{p-2}, \ldots, i_0\}$ *is the* ***input alphabet****;*
2. $S = \{s_{q-1}, s_{q-2}, \ldots, s_0\}$ *is the finite* ***state set****;*
3. $O = \{o_{r-1}, o_{r-2}, \ldots, o_0\}$ *is the* ***output alphabet****;*
4. δ *is a discrete function:*

   $$\delta: \; I \times S \longrightarrow S;$$

   called ***next state*** *or* ***transition function*** *of* $\mathcal{A}$*; given a pair* $(i_j, s_k) \in I \times S$*, if* $\delta(i_j, s_k)$ *is specified,* $s_l = \delta(i_j, s_k)$ *is the* ***next state*** *of the automaton* $\mathcal{A}$ *corresponding to the input* i_j *and to the* ***present state*** s_k*;*
5. λ *is a discrete function:*

   $$\lambda: \; I \times S \longrightarrow O,$$

   called ***output function*** *of* $\mathcal{A}$*; given a pair* $(i_j, s_k) \in I \times S$*, if* $\lambda(i_j, s_k)$ *is specified* $o_m = \lambda(i_j, s_k)$ *is the* ***output*** *of the automaton* $\mathcal{A}$ *corresponding to the input* i_j *and to the* ***present state*** s_k*;*

The pair $(\delta(i_j, s_k), \lambda(i_j, s_k))$ *is called a* ***transition*** *of automaton* $\mathcal{A}$*.*

When both δ and λ are complete functions, the automaton is ***complete*** or ***completely specified***; otherwise, it is ***partial*** or ***incompletely specified***.

The first observation we can make about the above definition of finite automata is that it could be more general, by letting δ and λ be defined as binary relations. This more general case is treated extensively by Zahnd in [122]. By constraining δ and λ to be functions, we limit the finite automaton model, which is then capable of describing *standard type sequential machines* only, according to the terminology introduced in [122]. However, we are interested here in manipulating sequential machines that present a *deterministic* behavior (as defined by Kohavi in [72]), and the finite automaton definition we provide is capable of representing any deterministic sequential machine specification [72].

Second, our definition corresponds to the model known as a ***Mealy machine*** [83]. Another possibility would be to define automata based on the ***Moore machine*** model [85], where the output function depends on the state set only. As long as we do not admit null length input sequences (which is exactly what is implied by the sequence definition above), the two models are completely equivalent, i.e. any behavior described using one model can be described using the other, and vice versa [72].

An FSM is considered here as one possible implementation (abstract or concrete) of the finite automaton algebraic definition. For simplicity reasons, however, no distinction between these terms is to be made in this work. An FSM can be represented in several ways. The most common explicit representations are the tabular form called *flow table*, and the graph form called *flow graph*. In the present work, we rely mostly on tabular representations. One example is the flow table, another is the *cube table*, to be introduced in Section 3.3.

Rows in the ***flow table*** correspond to states of an FSM, while columns are associated with the FSM inputs; the table entry corresponding to a pair (input, present state), noted (i_j, s_k), is filled in with the transition $(\delta(i_j, s_k), \lambda(i_j, s_k))$. An example of flow table appeared already in Chapter 2, where Table 2.1 described the behavior of the FSM used as a case study. Again, unspecified entries, also called don't care conditions, are identified by dashes (-) in flow tables.

A finite automaton describes an iterative process; as such, it can receive distinct hardware implementations:

1. a ***combinational implementation***, as the cascade connection of cells realizing the next state and output functions;

2. a ***sequential synchronous implementation***, as the loop connection of a combinational circuit CC realizing the next state and output functions with a suitable clocked memory;

3. a ***sequential asynchronous implementation***, similar to the previous implementation, but where combinational and sequential elements are interspersed in the structure, with no global clock to control the evolution of the states;

4. a ***mixed implementation***, arising from trading-off the above "pure" implementations, and usually resulting in a flat or hierarchical network of communicating hardware blocks.

In this text, we always refer to the implementation of an automaton as a sequential synchronous one. The sequential synchronous implementation is sketched for a Mealy machine in Figure 3.1. This is the hardware model we assume in the rest of this work. In what follows, we call CC the ***combinational part*** of the FSM.

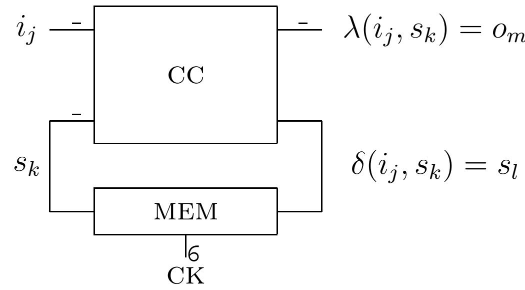


Figure 3.1: Sequential Mealy machine.

The finite automaton concept introduced in this Section is not directly implementable as a digital circuit, as Figure 3.1 may suggest. The practical aspects to be accounted for in the necessary "translation" process are explored from Chapter 5 on.

3.3 Representations of Discrete Functions

The main benefit of assuming a synchronous implementation as goal is that some complex problems of sequential logic design are made simpler. Notably, a synchronous implementation simplifies the consideration of delay problems, concentrating the complexity of the sequential logic design task on the implementation of the purely combinational part[2]. The discrete functions δ and λ describe the input/output behavior of the combinational part. The combinational part is then a discrete function as well, according to our definitions, and it would be useful to have at our disposal ways of efficiently representing such functions. In this Section, we develop a *cubical notation* to represent discrete functions that is adequate to our purposes. We presuppose the knowledge of common representation forms such as truth tables, for discrete functions in general, and Boolean expressions, for Boolean functions.

Let f be a discrete function of the form

$$f : S \longrightarrow L, \text{ where } S = \mathop{\times}_{i=n-1}^{0} S_i,$$

and where the cardinalities of S and L are $m = \prod_{i=n-1}^{0} | S_i |$ and r, respectively. We will assume every discrete function mentioned in this Section to be of this form, except when noted otherwise.

We define now a simple form for representing discrete functions.

Definition 3.13 (Value vector) *Let us choose an adequate enumeration order* [35] *of the elements of S, by defining an injection j from S onto the set of integers* $\{0, 1, \ldots, m-1\}$.

[2]This fact does not imply that *all* sequential aspects of the FSM can be overlooked. The sequential characteristics of the problem must be used to guide the combinational part synthesis, in the form, e.g. of cost functions.

Define an integer function $g : \{0, 1, \ldots, m-1\} \longrightarrow L$*, such that* $g(j(s)) = f(s)$*. Function* g *is represented by a vector of* m *values of the form*

$$[g(e)], \quad \text{with } e = 0, 1, \ldots, m-1; \;\; g(e) \in L, \forall e.$$

Given an enumeration order, this is the **value vector** *of function* f*, noted* $[f_e]$*.*

Example 3.1 (Value vector) Let f be a discrete function from the set $S = \{0, 1, 2, 3, 4\}$ into the set $L = \{0, 1\}$, where $f(x) = 0$ if x is even, and $f(x) = 1$ otherwise. Assume the natural enumeration order of the integer elements of S. With this enumeration order, the value vector that describes the behavior of f is $[f_e] = [0\ 1\ 0\ 1\ 0]$.

∎

It should be clear from Definition 3.13 that, given an enumeration order on S, the associated value vector uniquely describes f. This representation is but a compact version of the traditional truth table representation of discrete functions.

Davio, Deschamps and Thayse discuss the lattice [102] structure of discrete functions in [35]. A global overview of their work is out of the scope of this work. However, two of the concepts they define are invaluable to understand our definitions, the *lattice exponentiation function* and the *cube function*.

Let us choose an enumeration order for the elements of the codomain L of function f. We do so by defining an injection from L to the set of integers $\{0, 1, \ldots, r-1\}$. L is then a totally ordered set under this enumeration. The algebraic structure $\langle L, \vee, \wedge, 0, r-1 \rangle$ is a lattice under the disjunction ($\vee$) and conjunction ($\wedge$) operations, having a least element 0, and a greatest element $(r-1)$. Davio, Deschamps and Thayse define the disjunction and conjunction of two discrete functions f and g, and of a discrete function f and an element l of the codomain L, as the componentwise extensions of the lattice operations in L. These extensions are easily made explicit with the help of the value vector representation:

$$[(f \vee g)_e] = [f_e \vee g_e], \;\; [(f \wedge g)_e] = [f_e \wedge g_e], \;\; \text{and}$$

$$[(f \vee l)_e] = [f_e \vee l], \;\; [(f \wedge l)_e] = [f_e \wedge l].$$

Definition 3.14 (Lattice exponentiation function) *Let* $\mathbf{x}$ *denote the vector of variables* $(x_{n-1}, \ldots, x_1, x_0)$*, taking values on* S*, with* $S = \underset{i=n-1}{\overset{0}{\times}} S_i$*; given a variable* $x_i \in \mathbf{x}$ *and a subset* $C_i \subset S_i$*, we define the* **lattice exponentiation function** $x_i^{(C_i)}$ *as the discrete complete function*

$$x_i^{(C_i)} = \begin{cases} r-1 & \textit{iff } x_i \in C_i \\ 0 & \textit{otherwise.} \end{cases}$$

From this definition it should be clear that $x_i^{(S_i)} = r-1$ and that $x_i^{(\emptyset)} = 0$, for all i. The lattice exponentiation concept allows us to express any of the $r^{|S_i|}$ distinct functions of a single variable taking values on the set S_i. To verify this statement we present the special case of switching functions as an example.

Example 3.2 (Switching functions) Consider a switching function

$$h : \mathcal{B}^n \longrightarrow \mathcal{B}.$$

Remember that $\mathcal{B} = \{0, 1\}$ (cf. last paragraph of Section 3.1). Let $\mathbf{x}$ be the vector of n Boolean variables $(x_{n-1}, \ldots, x_1, x_0)$. Then, for any x_i, all possible distinct lattice exponentiation functions are:

$$x_i^{(\emptyset)} = 0; \quad x_i^{(\{0\})} = \overline{x_i}; \quad x_i^{(\{1\})} = x_i; \quad x_i^{(\mathcal{B})} = 1.$$

The horizontal bar over a symbol, like in $\overline{x_i}$ above stands for the Boolean complement operation. One often refers to the lattice exponentiation functions $x_i^{(\{0\})} = \overline{x_i}$ and $x_i^{(\{1\})} = x_i$ as ***literals*** or ***Boolean literals***.

■

By ***lattice expression*** we understand a *well-formed expression* [35] made up of three kinds of symbols:

1. lattice elements, i.e. elements of the set $\{0, 1, \ldots, r-1\}$;
2. variables of the form $x_i^{(C_i)}$;
3. binary lattice operations (disjunction and conjunction).

Any discrete function, as demonstrated in [35], can be represented by a lattice expression.

Definitions 3.15 (Cube function) *A* ***cube function*** *or simply a* ***cube*** *is a discrete complete function $c : S \longrightarrow L$, where the values $c(\mathbf{x})$ are computed by the expression*

$$c(\mathbf{x}) = l \wedge \bigwedge_{i=n-1}^{0} x_i^{(C_i)}, \qquad l \in L, C_i \in S_i;$$

The lattice element l is called the ***weight*** *of the cube. If $c(\mathbf{x})$ is such that $\forall C_i, \mid C_i \mid = 1$, then $c(\mathbf{x})$ is a* ***minterm***. *Given two cubes, $c(\mathbf{x}) = l \wedge \bigwedge_{i=n-1}^{0} x_i^{(C_i)}$, $l \in L, C_i \in S_i$, and $d(\mathbf{x}) = m \wedge \bigwedge_{i=n-1}^{0} x_i^{(D_i)}$, $m \in L, D_i \in S_i$, their* ***supercube*** *is a cube*

$$p(\mathbf{x}) = (l \wedge m) \wedge \bigwedge_{i=n-1}^{0} x_i^{(C_i \cup D_i)}.$$

The supercube definition is immediately extendable to sets of cubes with cardinality bigger than two. The cubes $c(\mathbf{x})$ and $d(\mathbf{x})$ are ***disjoint*** *if there is no $\mathbf{v} \in S$ for which $c(\mathbf{v}) = l$ and $d(\mathbf{v}) = m$, or if $l = 0$ or $m = 0$. The* ***size*** *of a cube $c(\mathbf{x})$ is $\mid \{\mathbf{x} \mid c(\mathbf{x}) = l\} \mid$, if $l \neq 0$, otherwise the size is 0.*

The ***satisfying set*** *of c is the set of Boolean vectors $\{\mathbf{x} \mid c(\mathbf{x}) = l\}$, if $l \neq 0$, otherwise it is the empty set. The satisfying set of a cube is noted $sat(c)$. Every element in this set is said to* ***satisfy*** *the cube function c. A* ***switching cube function*** *is a cube whose domain is $S = \mathcal{B}^n$ and whose codomain is $L = \mathcal{B}$, for some integer n.*

The satisfying set, together with the weight completely define a cube. Very often, cubes are manipulated through their satisfying sets, specially in switching functions where the weight can be assumed to be 1. When no confusion may arise, references to the cube c as a set stand for the satisfying set of some cube c.

The concept of cube function as stated above was introduced by Davio and Bioul in [34]. The cube $c(\mathbf{x})$ takes the value l iff for all i, $x_i \in C_i$, and takes the value 0 otherwise. If at least one of the C_i sets is empty, the cube assumes the value 0 everywhere, and it is accordingly called an ***empty cube***. On the other hand, if $C_i = S_i$ for every i, the cube $c(\mathbf{x})$ is the ***constant cube*** $c(\mathbf{x}) = l;\ \ l \in L$.

To illustrate the cube concept, we revisit the switching functions example.

Example 3.2 (Continued) Non-trivial switching cubes (i.e. those distinct from the empty and constant cubes) correspond to a conjunction, or ***product*** of literals. The only lattice element relevant in these cubes is 1, the greatest element in the lattice $\mathcal{B}$, which can accordingly be dropped. For instance, the switching cube function $c(\mathbf{x})$ corresponding to the lattice expression $1 \wedge x_2^{(\{0\})} \wedge x_1^{(\{0,1\})} \wedge x_0^{(\{1\})}$ can be represented by the product of literals $\overline{x_2} \wedge x_0$.

■

The interpretation of a cube function depends strongly on the underlying structure of the domain set. If the domain S of a cube c is not considered as a Cartesian product, $sat(c)$ can be *any* subset of S. However, if S is considered as a Cartesian product $S = S_1 \times \ldots \times S_m$, $sat(c)$ must be one set of the form $X_1 \times \ldots \times X_m$, where for $i = 1, \ldots, m$, X_i is a subset of S_i. Thus, if the structure of the domain is not specified, the cube function concept is ambiguous [123]. In what follows we always specify explicitly this Cartesian structure.

If a discrete function is expressed by a disjunction of cube functions, the resulting representation is called a ***disjunctive normal form***[3].

Cubes and the disjunctive normal form are used in [35] to develop a *canonical disjunctive form*, having the form of a special disjunction of cubes. Any discrete function may be represented in both disjunctive normal form and canonical disjunctive form. The dual concepts of cubes, *anticubes*, and of the disjunctive forms, *conjunctive normal form* and *canonical conjunctive form*, are also extensively discussed in the same work. The canonical forms have a direct mapping to hardware implementations in the form of read-only memories or ROMs, while the normal forms can be associated with any two-level hardware implementation, and particularly to PLA implementations [36].

Before discussing cubical representation of discrete functions, we introduce a tabular notation due to Zahnd [123], that is adequate to represent discrete functions whenever the Cartesian structure of the domain is irrelevant.

Definitions 3.16 (Function tabular specification) *Given sets S and L, a* ***function tabular specification*** *with* ***domain*** *S and* ***codomain*** *L, which is also called simply* ***function specification*** *is a set of pairs of the form*

$$\Gamma = \{(A_1, z_1), \ldots, (A_n, z_n)\}$$

[3]We follow here the terminology of [35], which distinguishes *normal form* from *canonical form*. Given a function, a canonical form of it is a normal form that is unique.

such that for $i = 1, \ldots, n$ we have $A_i \subset S$, $z_i \in L$, and such that for $i, j = 1, \ldots, n$, we have

$$z_i \neq z_j \Longrightarrow A_i \cap A_j = \emptyset. \tag{3.1}$$

The elements (A_i, z_i) are the **rows** *of Γ. We say that a complete function $f : S \longrightarrow L$* **satisfies** *the specification Γ (or the table Γ) if, for all $i = 1, \ldots, n$ we have*

$$f(s) = z_i, \text{ for all } s \in A_i. \tag{3.2}$$

The specification Γ is a tabular representation of the set of complete functions $f : S \longrightarrow L$ that satisfy it. Due to (3.1), this set is never empty. In the special case where the union of the sets $A_1, \ldots, A_n$ is the set S, there is only one complete function $f : S \longrightarrow L$ that satisfies the table.

Definitions 3.17 (Equivalence between function specifications) *Given Γ and Γ', two function specifications of the form $S \longrightarrow L$, we say that Γ is* **finer** *than Γ' if every complete function $f : S \longrightarrow L$ that satisfies Γ satisfies Γ' as well. We may also say that Γ' is* **coarser** *than Γ. This determines a binary relation which is reflexive and transitive, i.e. a preorder relation on the set of function specifications of the form $S \longrightarrow L$. We say that Γ and Γ' are* **equivalent** *if each specification is finer than the other, or alternatively, if every complete function satisfying one specification satisfies also the other. This clearly defines an equivalence relation (reflexive, symmetric and transitive) on the set of specifications with domain S and codomain L.*

A function specification $\Gamma = \{(A_1, z_1), \ldots, (A_n, z_n)\}$ is said to be **reduced** *if it does not contain two rows (A_i, z_i), (A_j, z_j) such that $A_i \neq A_j$ and $z_i = z_j$. It is clear that if this last case arises, we can always build an equivalent specification by substituting these two rows by $(A_i \cup A_j, z_i)$. It is also obvious that if Γ is reduced, every table equivalent to Γ has at least as many rows as Γ. On the other hand, given a specification Γ', there is* one and only one[4] *reduced specification Γ equivalent to Γ'. We will say that Γ is obtained* **by reduction** *from Γ'. This reduction process is what* we *call* ***symbolic minimization***.

3.3.1 Cubical Representations of Discrete Functions

Cube representations are adapted to compactly express two-level forms of discrete functions. In this Section, we introduce cube tables, a representation form that can describe sets of discrete functions that share a domain interpreted as a Cartesian product. The whole set of represented discrete functions can always be seen as a single discrete function whose codomain is the Cartesian product of the codomains of all functions in the set, and can thus be treated using the ordinary function tabular specification defined in the last Section. However, the manipulation of discrete partial functions is enhanced if the set of functions notation is retained, and the use of cubes allows a smooth passage from tabular symbolic representations to tabular encoded representations, and that is what we need it for. Before introducing cube tables, we discuss a graphical interpretation for cubes.

The satisfying set of a cube function characterizes its behavior, describing completely the set of elements of the domain that evaluate to the weight of the cube. We may use Hasse

[4]We must keep in mind that this will not be the case in the cube tables, to be defined in Section 3.3.1.

diagrams [22] to represent ordered sets in general, lattices, and of course, satisfying sets of cube functions in particular.

Definition 3.18 (Hasse diagram) *Given an ordered set $\langle S, \leq \rangle$ [102], a* ***Hasse diagram*** *represents $\langle S, \leq \rangle$ as a directed graph. Each element of S is represented by a vertex in the diagram; the vertices are connected by edges in such a way that there is a path from an element a to an element b iff $a \leq b$. To achieve this result with the fewest possible edges, an edge is drawn from a to b iff:*

1. *$a \leq b$ and*
2. *there is no c such that $a \leq c \leq b$.*

Example 3.3 (Hasse diagrams of switching cubes) Let $\langle \mathcal{B}^4, \leq \rangle$ be an ordered set where $\leq$ is the partial ordering $\leq$: $\{(a, b) \,|\, a \vee b = b\}$, $\vee$ being the componentwise Boolean disjunction operation. The Hasse diagram representing this set is shown in Figure 3.2. The direction of the edges is not shown, being implicit [35] that they go from a lower to an upper position.

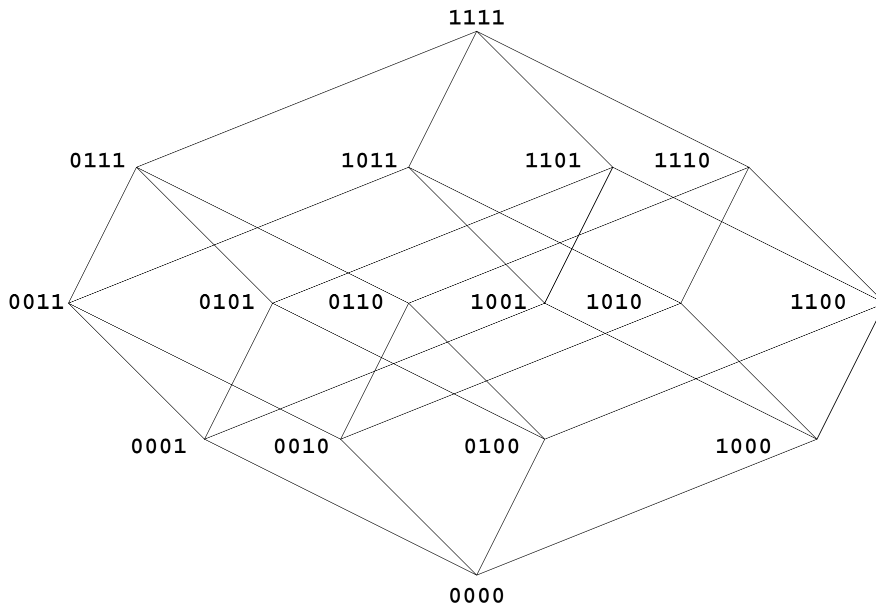


Figure 3.2: Hasse diagram for Example 3.3

Every switching cube function has a satisfying set that is a complete subgraph in the Hasse diagram (if we ignore the directed nature of the graph) of the domain $\mathcal{B}^n$, under the $\leq$ relation or equivalent [35]. For example, each vertex in the diagram corresponds to a minterm cube function, and the supercube of two cubes is the smallest complete subgraph that contains every element in each satisfying set of the two cubes.

∎

Now, let $S = S_1 \times S_2 \times \ldots \times S_m$ be a Cartesian product, $L_1, \ldots, L_n$ be sets and $F = \{f_1, \ldots, f_n\}$ be a set of discrete functions of the form $f_i : S \longrightarrow L_i, (i = 1, \ldots, n)$. Remember that $\mathbf{n} = \{1, \ldots, n\}$ represent the set of positive integers between 1 and n.

Definitions 3.19 (Cube table) *A* ***cube table*** *is a set*

$$\Omega = \{(C_1, D_1), \ldots, (C_q, D_q)\}$$

of pairs (C_j, D_j) where the C_js are satisfying sets of non-empty cubes c_j with domain S, where $D_j = \{d_1^j, \ldots, d_n^j\}$, with each element d_i^j being one of:

1. *the empty set $\emptyset$;*
2. *the set L_i;*
3. *a singleton $\{l_i\}$ contained in L_i,*

and where the following condition holds:

$$d_i^j \neq d_i^k \Longrightarrow C_j \cap C_k = \emptyset, \quad \text{for all } i \in \mathbf{n} \text{ and for all } i, j \in \mathbf{k}.$$

The set of C_js is called the ***input part*** *of Ω, while the set of D_js is called the* ***output part*** *of Ω.*

The last requirement in the above definition expresses the conditions a cube table must obey so that it represents a set of discrete functions. This definition allows the simultaneous representation of a number n of discrete functions of a Cartesian product S, using a single set of cubes. Unspecified values in partial functions are represented by the situations where one element d_i^j is the set L_i in the output part. The empty set in the output part of the cube table is needed to model the situation where a cube is irrelevant to the representation of a subset of the n functions. Since the n functions represented are defined as partial, and any partial function may be seen as a set of complete functions, a cube table can be viewed as a set of n sets of complete functions.

Cube tables are identical to function tabular specifications if the Cartesian product structure of the involved domain and codomain is ignored, which is adequate when dealing with the theoretical aspects of discrete functions.

Definition 3.20 (Cube table satisfaction and equivalence) *The set $F = \{f_1, \ldots, f_n\}$*[5] *of discrete functions* ***satisfies*** *a cube table Ω if*

$$\forall (i \in \mathbf{n}, (C_j, D_j) \in \Omega), \quad d_i^j = \{l_i\} \Longrightarrow \text{for all } s \in C_j \text{ we have } f_i(s) = l_i.$$

Conversely, a cube table Ω ***satisfies*** *the set F if*

$$\forall (i \in \mathbf{n}, s \in S), \quad f_i(s) = l_i \Longrightarrow \exists (C_j, D_j) \in \Omega \text{ such that } s \in C_j, d_i^j = \{l_i\}.$$

The cube table Ω ***represents*** *F if Ω satisfies F and F satisfies Ω. A necessary condition for this to happen is that for all $i \in \mathbf{n}, s \in S$,*

$$\begin{aligned} f_i(s) \in F \text{ is unspecified} \Longrightarrow \; & \text{either } \exists (C_j, D_j) \in \Omega \text{ such that } s \in C_j, d_i^j = L_i \\ & \text{or } \quad \nexists (C_j, D_j) \in \Omega \text{ such that } s \in C_j, d_i^j \neq \emptyset. \end{aligned}$$

[5] Note that the term *satisfaction* applies here to partial functions in general, not only to complete functions.

Let Ω and Ω' be two cube tables describing a set of n functions with domain $S = \{S_1, \ldots, S_m\}$ and respective codomains L_i for $i = 1, \ldots, n$. We say that Ω is **finer** *than Ω' if every set of functions F satisfying Ω satisfies Ω'. Alternatively we say that Ω' is* **coarser** *than Ω. This defines a preorder relation on the set of cube tables of this form. Ω and Ω' are* **equivalent** *if each is finer than the other, which defines an equivalence relation on the set of cube tables with domain S and respective codomains L_i.*

One cube table $\Omega = \{(C_1, D_1), \ldots, (C_q, D_q)\}$ is said to be **reduced** *if there is no equivalent cube table Ω' with less elements than Ω.*

A cube table that satisfies a set of functions does not necessarily represents the set, since it can have less unspecified entries in the associated truth table than the set of functions that satisfies it. The importance of the satisfaction concept is that it tells in what conditions we may substitute a set of functions by a cube table, whenever only the specified behavior of the functions is relevant. In case all functions in F are completely specified, the concepts of satisfaction and equivalence are clearly synonyms. In general, a reduced cube table is not unique, due to the requirement that the input parts be cubes, not any subset of S. Let us illustrate cube tables by an example.

Example 3.4 (Discrete functions and cube tables) Let $I_1 = \{a, b, c\}$, $I_2 = \{d, e\}$, $O_1 = \{g, h\}$ and $O_2 = \{i, j, k\}$ be sets, $S = I_1 \times I_2$ and $F = \{f_1, f_2\}$ be a set of discrete functions such that $f_1 : S \longrightarrow O_1$ and $f_2 : S \longrightarrow O_2$. The behavior of F is depicted in Table 3.1, under the form of a truth table.

Table 3.1: Truth table for Example 3.4

S		F	
I_1	I_2	f_1	f_2
a	d	h	k
a	e	h	i
b	d	h	k
b	e	g	-
c	d	-	k
c	e	g	j

Table 3.2, on the other hand, depicts three cube tables that satisfy F, and introduces the notation we use to represent a cube table. Each row in each of the cube tables corresponds to an element of the cube table, with the input part represented to the right, using set notation. Each C_j is the Cartesian product of the sets in the input part of some row. The output part, on the other hand, uses a distinct notation. Since every d_i^j in D_j is the empty set, L_i or a singleton of L_i, we omit the braces of the set notation. In both, input and output parts, unspecified entries "-" represent the set L_i associated with the column. Any subset of S not explicitly assigned a value by the cube table corresponds to an unspecified condition as well. Thus, unspecified conditions can be represented either explicitly or implicitly. For example, consider the cube table 2 in Table 3.2. Function f_1 is not specified for the element $(c, d) \in S$, but this is only implicit in cube table 2, while the fact that f_2 is not specified for the element $(b, e) \in S$ is explicitly depicted in the fourth row of cube table 2.

Table 3.2: Three cube tables for Example 3.4

1				2				3			
C_j		D_j		C_j		D_j		C_j		D_j	
I_1	I_2	f_1	f_2	I_1	I_2	f_1	f_2	I_1	I_2	f_1	f_2
$\{a\}$	-	h	$\emptyset$	$\{a,b\}$	$\{d\}$	h	k	$\{a,b,c\}$	$\{d\}$	h	k
-	$\{d\}$	$\emptyset$	k	-	$\{d\}$	$\emptyset$	k	$\{b,c\}$	$\{e\}$	g	j
$\{a\}$	$\{e\}$	$\emptyset$	i	$\{a\}$	$\{e\}$	h	i	$\{a\}$	$\{e\}$	h	i
$\{b\}$	$\{d\}$	h	$\emptyset$	$\{b\}$	$\{e\}$	g	-				
$\{b\}$	$\{e\}$	g	-	$\{c\}$	$\{e\}$	g	j				
$\{c\}$	$\{e\}$	g	j								

Cube tables 1 and 2 satisfy F, they are both satisfied by F and are thus equivalent. We then conclude that cube table representations are not canonical. Cube table 3 satisfies F but is not satisfied by it, and it is neither equivalent to cube table 1 nor to cube table 2. Indeed, cube table 3 represents a set of two discrete complete functions, and is an instance of what we define below as a *complete cube table.*

∎

Definition 3.21 (Complete cube tables) *The cube table Ω is* ***complete*** *or* ***completely specified*** *if for all $s \in S$, there exists a subset of Ω defined as*

$$\Upsilon = \{(C_j, D_j) \mid s \in C_j\},$$

and for all $i \in \mathbf{n}$ there is an integer j such that $D_j \in \Upsilon$, $d_i^j \in D_j$ and $\mid d_i^j \mid = 1$.

Complete cube tables are related to circuit implementations of discrete functions. Every circuit intended to implement the behavior of a cube table implements in fact a complete cube table that satisfies the set of functions from which the cube table was obtained.

We will employ four schemes to represent sets of discrete functions using cube tables. The first two are the ***symbolic scheme*** and the ***positional cube scheme***. These are general schemes, since both may represent any set of discrete functions. The first of them represents the set elements of the domain and codomain directly, using set notation. The second is equivalent to the first, but the domain and codomain elements are assigned 1-hot Boolean codes. This encoding allows the use of multiple-valued logic techniques to manipulate the set of functions. The ***switching scheme*** is useful to represent sets of switching functions only, while the last, called ***mixed scheme***, allows the representation of sets of functions where part of the specification is already in the form of Boolean codes and part is in symbolic form.

3.3.1.1 Cube Table Schemes for Sets of Discrete Functions

Example 3.4 introduced the symbolic scheme for cube tables. The positional cube scheme implies the use of bit vectors to represent cubes, as well as output values. Su and Cheung suggested the positional cube scheme to represent multiple-valued functions [109]. The particularities of this scheme will be introduced in Example 3.5. We refer to [109] for a formal statement of the positional cube scheme.

Example 3.5 (Discrete function cube table) Let $I = \{\mathtt{inp0}, \mathtt{inp1}, \mathtt{inp2}, \mathtt{inp3}, \mathtt{inp4}\}$ be a set of input symbols and $O = \{\mathtt{out0}, \mathtt{out1}, \mathtt{out2}\}$ be a set of output symbols. Define a discrete function

$$f : I \longrightarrow O,$$

and let the behavior of f be displayed in Figure 3.3, using a cube table. Figure 3.3(a) shows the symbolic scheme, while Figure 3.3(b) illustrates the equivalent positional cube scheme.

```
        Symbolic Scheme        Positional Cube Scheme

      {inp0,inp2}  {out1}          10100 010
         {inp3}       -            00010 111
         {inp1}    {out2}          01000 001

               (a)                     (b)
```

Figure 3.3: Symbolic and positional cube schemes to represent the f discrete function

In Figure 3.3(b), the domain and codomain of f are represented by $|\, I\,|$-bit and $|\, O\,|$-bit vectors, respectively. Enumeration orders must be defined on the elements of both sets, prior to establishing the representation. Herein, the enumeration will be either obvious from the naming of the elements or given explicitly. An element name ending with the integer value v is represented by a 1 in the $v - th$ position. Don't cares are accordingly represented by an all 1s bit vector.

∎

3.3.1.2 Cube Table Scheme for Sets of Switching Functions

A specific cube table scheme for discrete switching functions is justified by the efficiency it provides in manipulating such functions.

Consider the general switching function

$$h : \mathcal{B}^n \longrightarrow \mathcal{B}^r.$$

Remember that a switching cube $c(\mathbf{x})$ can be represented either by 0 (the empty cube) or by a product of v distinct literals ($0 \leq v \leq n$), where each literal is either x_i or $\overline{x_i}$, with every literal corresponding to a position in $\mathbf{x}$ (cf. Example 3.2). We may represent this function compactly if we build its cube table as follows:

1. Associate the symbol 1 with the literal x_i and the symbol 0 to the literal $\overline{x_i}$. With the lattice exponentiation $x_i^{(\{0,1\})} = 1$ associate the don't care symbol '-'. Depict the satisfying set of a cube of h as the vector of length n that represents this cube in this encoding. Such a vector represents a set C_j for a cube table representing h;
2. the output part D_j is also represented by a vector containing the elements 0,1 and '-', with length r, but the interpretation of the symbols is distinct. Let the component switching functions of h be $h_{r-1}, \ldots, h_1, h_0$ and:

(a) insert a 1 on the right side of a cube table row in position h_i if the component h_i assumes the lattice value 1 whenever the cube associated with the row evaluates to 1;

(b) insert a 0 on the same position if the cube in question does not participate in the lattice expression describing the behavior of the component h_i;

(c) insert a '-' if the component h_i may assume either 1 or 0 whenever the corresponding cube evaluates to 1, i.e. a don't care.

(d) if no cube evaluating to 1 in a given input configuration has a right side specifying 1 or '-' for component h_i, h_i assumes the lattice value 0, which is thus a default value.

Such a vector corresponds to the set D_j in the cube table.

The first item above suggests a three-valued representation for cubes that is adequate to represent any single switching function.

Definition 3.22 (Three-valued switching cube representation) *Any non-null switching cube function $c : \mathcal{B}^n \longrightarrow \mathcal{B}$ can be represented by a* ***three-valued*** *vector $q_{n-1} \dots q_0$, where $q_i \in \{0, 1, -\}$, as explained in item 1 above. Let two cubes, c_1 and c_2, be represented in this notation. These cubes are disjoint iff there is at least a position q_i where c_1 and c_2 are distinct and both are distinct from '-'.*

As an example of three-valued switching cube representation, consider the cube of Example 3.2, $c(x_2, x_1, x_0) = \overline{x_2} \wedge x_0$. In the three-valued representation we have $c(x_2, x_1, x_0) = 0 - 1$.

We define a componentwise ***conjunction*** operation $\wedge$ over the set $\{0, 1, -\}$ in Table 3.3.

Table 3.3: Componentwise conjunction of cubes

$\wedge$	0	1	-
0	0	$\emptyset$	0
1	$\emptyset$	1	1
-	0	1	-

It can be verified that the three-valued representation of the conjunction of two cubes c_1 and c_2 results from the componentwise conjunction of the three-valued representations of cubes c_1 and c_2. If any position of the resulting cube evaluates to the empty set, the resulting cube is the null cube.

We also define the componentwise ***supercube*** operation, in Table 3.4.

Again, the three-valued representation of the supercube of c_1 and c_2 results from the application of the componentwise supercube operation to the three-valued representations of c_1 and c_2.

Herein, we shall use the three-valued representation to note either the cube itself or its satisfying set.

Table 3.4: Componentwise supercube operation

sup	0	1	-
0	0	-	-
1	-	1	-
-	-	-	-

3.3.1.3 Cube Table Mixed Schemes

The goal of introducing the above representations is to permit the manipulation of mixed descriptions that often arise in FSM logic synthesis. We highlight the benefits of mixed representations in the Example below.

Example 3.6 (Mixed cube table for discrete functions) Consider the following two finite sets $I = \{\texttt{inp0}, \texttt{inp1}, \texttt{inp2}\}$ and $O = \{\texttt{out0}, \texttt{out1}, \texttt{out2}\}$, and $S = \mathcal{B}^4 \times I$. Define a set of discrete functions $F = \{f_1, f_2, f_3, f_4\}$ such that $f_1 : S \longrightarrow O$, $f_2 : S \longrightarrow \mathcal{B}$, $f_3 : S \longrightarrow \mathcal{B}$, and $f_4 : S \longrightarrow \mathcal{B}$, the behavior of F being described by Figure 3.4.

```
   Mixed symbolic/switching        Mixed positional cube/switching

     Input            Output           Input     Output
      Part             Part             Part      Part

  1-00 {inp0,inp2}  out2 100          1-00 101  001 100
  111- {inp0,inp2}    -  011          111- 101  111 011
  101- {inp0,inp1}  out1 11-          101- 110  010 11-
  1-01      -       out0 101          1-01 111  100 101
  0---   {inp0}     out0 111          0--- 100  100 111

              (a)                               (b)
```

Figure 3.4: Behavior of function F using symbolic and positional cube schemes

Note that F is a set discrete function that has the domain specified as a Cartesian product presenting two-valued, as well as multiple-valued components, and that the elements of F have codomains that are either multiple-valued (f_1) or two-valued (the others). Figure 3.4(a) depicts a mixed symbolic/switching cube table that satisfies F, with the multiple-valued components displayed as in the symbolic scheme. Figure 3.4(b) shows the same cube table using a mixed positional cube/switching scheme.

The compactness of using the cube notation is evident from the Figure (e.g. F would require a 48-row truth table to specify its behavior). Most computationally efficient methods devised to manipulate two-level forms of sets of discrete functions employ some variation of the mixed positional cube/switching scheme to represent functions internally. Since the symbols used to represent two-valued and multiple-valued information are the same, a distinction must be made between them to avoid confusion. In our examples, we separate distinct-valued information in distinct column groups, and we also separate the input part from the output part, since the same symbols have distinct meanings in each part.

Note that the output parts of both cube tables in the Figure are *disjoint*, in the sense that no two rows have intersecting output values. This characteristic is a desirable one, otherwise a conflicting specification *may* arise. If the output part is disjoint, the input part has to be, too. Otherwise the cube table cannot describe a set of functions. For example, if the fourth row in Figure 3.4(a) were "`0001 {inp0,inp2}`" instead of "`1-01 -`", the Figure could not represent a function, since there would be two distinct output values associated with the input "`0001 {inp0}`", namely "`{out0} 101`" and "`{out0} 111`". We can also verify that all elements of F are partial functions, once there are elements of the common domain with no output value associated with it for any function, e.g. "`0000 {inp2}`".

∎

The advantages of using the above cube representations should be clear by now. They in fact allow the use of efficient Boolean manipulation techniques for treating discrete functions. At the same time, the use of positional cube schemes permits encoding multiple-valued data in binary form without loosing track of the symbolic nature of the information.

Part II

Constraints: Nature, Generation and Relationship

Chapter 4

State Minimization Constraints

In this Chapter, we analyze the SM problem, from the standpoint of a state assignment constraint formulation. The main objective is to provide a formal definition of a set of elementary constraint types that is sufficient to uniquely specify a generic instance of the SM problem, having in sight the use of such constraints to guide the state encoding design step. After identifying the constraints involved in the SM problem, we model each constraint kind as a binary relation, the elements of which are constraints of a given kind, represented in the most elementary form.

The specific definitions required to state the SM problem appear in Section 4.1. We stress that our intent in that Section will be to introduce the related terminology, not to provide a thorough insight of the problem, which can be found in textbooks like [72, 122], except for the *class set* concept suggested by Grasselli and Luccio in [56]. Section 4.2 contains the SM problem statement. The three next Sections define the constraint types that are sufficient to unequivocally describe any instance of the problem. To conclude the Chapter, Section 4.6 outlines the classical method of SM constraints generation, and Section 4.7 discusses the complexity of the method.

4.1 State Minimization Definitions

Let $\mathcal{A}$ be an FSM $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$. Consider a word of length n over the input alphabet I, noted $[i(1,n)] \in I^+$, where I^+ is the set of sequences of arbitrary length over the alphabet I, and an arbitrary state $s \in S$ of machine $\mathcal{A}$. We define

$$\lambda([i(1,n)], s) = \lambda_i(s) = \lambda(i_n, \delta(i_{n-1}, \delta(i_{n-2}, \ldots, \delta(i_1, s) \ldots))).$$

Thus, $\lambda_i(s)$ corresponds to the output value after the word $[i(1,n)]$ is applied to the machine $\mathcal{A}$ in the initial state s. Since we assume here deterministic FSMs, every $\delta(i_i, s_i)$ above is assumed to be specified.

Definitions 4.1 (Machine simulation and reduction) *Given two* FSM*s, $\mathcal{A}$ and $\mathcal{A}'$, with $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$ and $\mathcal{A}' = \langle I, T, O, \delta', \lambda' \rangle$, we say that machine $\mathcal{A}'$* ***simulates*** *machine $\mathcal{A}$ iff for every state $s \in S$, there exists a state $t \in T$ such that $\lambda_i(s) = \lambda'_i(t)$ for any word $i \in I^+$, whenever $\lambda_i(s)$ is specified. The* FSM *$\mathcal{A}'$ is a* ***reduction*** *of $\mathcal{A}$ if $\mathcal{A}'$ simulates $\mathcal{A}$ and $|T| \leq |S|$. $\mathcal{A}'$*

is a ***minimal reduction*** *or* ***minimization*** *of $\mathcal{A}$ iff there is no other reduction $\mathcal{A}'' = \langle I, U, O, \delta'', \lambda'' \rangle$ of $\mathcal{A}$ such that $|U| < |T|$.*

The interpretation of the concept is clear. If $\mathcal{A}'$ simulates $\mathcal{A}$, then machine $\mathcal{A}$ initially in state s can be replaced by machine $\mathcal{A}'$ initially in state t, without changing the input/output behavior expected from $\mathcal{A}$. Since to every $s \in S$ corresponds some $t \in T$, the replacement is possible for any initial state. We remark that the machine simulation definition is immediately applicable for two instances of a single machine. It corresponds to the special case where $\mathcal{A} = \mathcal{A}'$. However, we can draw a stronger relationship among the states of a machine.

Definitions 4.2 (State compatibility) *Consider an FSM $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$. Two states $s_1 \in S$ and $s_2 \in S$ are* ***compatible*** *iff for every word $i \in I^+$, whenever both $\lambda_i(s_1)$ and $\lambda_i(s_2)$ are specified, we have $\lambda_i(s_1) = \lambda_i(s_2)$. We denote that states s_1 and s_2 are compatible by writing $s_1 \sim s_2$. A subset of states $C \in S$ is a* ***compatibility class*** *or a* ***compatible*** *of $\mathcal{A}$ iff the states in C are pairwise compatible. A compatible C is a* ***maximal compatibility class*** *or a* ***maximal compatible*** *of $\mathcal{A}$ iff there is no other compatible D of $\mathcal{A}$ such that $C \subset D$.*

Let us interpret the compatibility concept intuitively. Suppose we have a machine $\mathcal{A}$ containing two compatible states s_1 and s_2. Suppose also that the machine is in one of these two states. Then, it is impossible to distinguish from the input/output (I/O) behavior alone, if the machine is in one or another of these two states, and this for any input word[1] $i \in I^+$.

Under certain conditions that we will discuss next, a compatible may represent a state in a minimization of $\mathcal{A}$. In fact, in any minimization of $\mathcal{A}$ each state corresponds to a judiciously chosen compatible [122]. As a result, compatible states may be distinguishable or not in the final implementation of the FSM, depending on how the grouping of compatible states into compatibility classes is performed.

Definitions 4.3 (Implied compatibles, closed sets and covers) *Given a compatible C of an FSM $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$, and an input letter $i \in I$, let D be the set $\{c \in C \mid \delta(i, c) \text{ is specified}\}$. The set*

$$C' = \{c' \mid c' = \delta(i, d) \quad \text{for some} \;\; d \in D\}$$

is a compatible of $\mathcal{A}$, as shown in [72]*. Then, C is said to* ***imply*** *C' under the input letter i, which is represented by the notation $C[\,i\,] \rightarrow C'$. We say that C' is a* ***closure condition*** *imposed on C by $\mathcal{A}$. A* ***closed set of compatibles*** *of $\mathcal{A}$ is a set K of compatibles such that for every $k \in K$ and for every $i \in I$*

$$k[\,i\,] \rightarrow k' \quad \text{and} \;\; \exists k'' \in K \mid \; k' \subseteq k''.$$

Putting it into words, every compatible k' implied by some compatible k in a closed set (of compatibles) is contained in some element of the closed set. Given a closed set of compatibles K of $\mathcal{A}$, a ***closed cover of compatibles*** *of $\mathcal{A}$ is a cover*

$$v : K \longrightarrow S,$$

[1] In the special case of completely specified machines (not treated here), this condition implies that the outputs associated with two compatible states are *always* identical, for all input word $i \in I^+$. In this case, the compatibility relation becomes an equivalence relation.

of S such that a pair $(k, s) \in K \times S$ is in the graph v of the cover iff $s \in k$. This closed cover is **minimum** *iff for any other closed set of compatibles K' of $\mathcal{A}$ associated to a closed cover of compatibles $v' : K' \longrightarrow S$ of S, we have $|K| \leq |K'|$.*

Definitions 4.4 (Class sets) *A* **class set** *P_C implied by a compatible C is the subset of compatibles implied by C, each one of which:*

1. *has more than one element;*
2. *is not contained in C;*
3. *is not contained in any other compatible in P_C.*

If a given compatible has an implied class set that is empty, the states in it are **unconditionally compatible**. *Otherwise, the states are* **conditionally** *or* **transitively compatible**.

Note that the class set concept expresses the non-trivial closure conditions associated to C. In other words, any domain of a closed cover of compatibles containing C must, for each compatible $P \in P_C$, contain a compatible D such that $P \subset D$.

Definitions 4.5 (Prime compatibles) *A compatible C is* **excluded** *by a compatible D iff:*

1. *$D \supset C$;*
2. *$P_D \subseteq P_C$, where P_D, P_C are the class sets of D and C, respectively.*

A **prime compatibility class** *or* **prime compatible** *of an* FSM *$\mathcal{A}$ is one compatible that is not excluded by any other compatible of $\mathcal{A}$.*

Grasselli and Luccio [56] showed that any FSM has at least one minimum closed cover of compatibles whose domain contains only prime compatibles. Whenever there is a need to obtain a single minimum closed cover of compatibles, restricting the search space to prime compatibles can lead to significant reductions in the computation resources required during the search.

4.2 State Minimization Problem Statement

In view of the definitions presented in Section 4.1, we may now discuss the formal statement of the FSM SM problem.

Problem Statement 4.1 (FSM state minimization) *In view of the definitions presented in Section 4.1, this problem is characterized by:*

INSTANCE: An FSM *$\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$, as described in Definition 3.12, and an integer k.*

QUESTION: Does an FSM *$\mathcal{A}'$ exist, with $\mathcal{A}' = \langle I, S', O, \delta', \lambda' \rangle$, such that $\mathcal{A}'$ is a minimization of $\mathcal{A}$ and $|S'| \leq k$?*

Pfleeger already showed in [95] that the above *decision problem* [54] is NP-complete. Nevertheless, its solution can be stated simply, using the following two-step procedure.

Given an FSM $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$,

1. *find a minimum closed cover of compatibles* $v : K \longrightarrow S$, *of* $\mathcal{A}$*;*
2. *from* v*, construct a new machine* $\mathcal{A}' = \langle I, S', O, \delta', \lambda' \rangle$*, that is a minimization of* $\mathcal{A}$*. Each element in* S' *corresponds to one compatible in* K*; also, define* δ' *and* λ' *based on* δ *and* λ*, respectively, such that* $\mathcal{A}'$ *simulates*[2] $\mathcal{A}$.

In the following Sections, we decompose the SM problem into the set of elementary constraints arising from the problem statement, as well as from the definitions in the previous Section. After investigating the different kinds of constraints arising from the structure of an FSM, a method for generating these constraints is presented, which we refer to as the *compatibility table method*.

4.3 Compatibility and Incompatibility Constraints

From the state compatibility definition in Section 4.1, we may draw two binary relations on the set of states of an FSM.

Definitions 4.6 (Compatibility and Incompatibility relations) *Given a finite state machine* $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$*, the* ***compatibility relation*** *of* $\mathcal{A}$ *is the binary relation* $\langle S, S, \theta \rangle$*, where*

$$\theta = \{(s,t) \mid s,t \in S \ \text{ and } \ s \sim t\}.$$

The ***incompatibility relation*** *of* $\mathcal{A}$ *is the binary relation* $\langle S, S, \iota \rangle$*, where*

$$\iota = \{(s,t) \mid s,t \in S \ \text{ and } \ s \not\sim t\}.$$

We note that the compatibility and incompatibility relations carry the same information under dual forms. It is also immediate to verify that the compatibility relation θ is reflexive and symmetric, but not necessarily transitive, whereas the incompatibility relation is symmetric, but not necessarily transitive, and it is never reflexive. Any pair (s,t) in θ (resp. ι) corresponds to a pair of compatible (resp. incompatible) states, and constitutes an ***elementary compatibility*** (resp. ***incompatibility***) ***constraint*** of machine $\mathcal{A}$.

Addressing the compatibility relation as a set of constraints is one of the original propositions in the present thesis. Previous works have never treated the state compatibility relation as a set of constraints, since given a subset of states of an FSM, they either form a compatible or not, depending only on the structure of the machine. However, we are interested in encoded FSM implementations, and how these implementations relate to the state minimization problem.

[2]Given v, the construction of δ' and λ' is straightforward, as can be seen from step 4 of *Process* A in [91]. A practical example illustrating the application of this step appeared in Chapter 2, in the process of building Table 2.2 from Table 2.1, and from the compatible pair of states $\{0, 1\}$.

In this sense, the fact that states are compatible imposes constraints in the subsequent state encoding step, so that respecting these constraints can lead to implementations that take into account state minimization.

Suppose that v is a minimum closed cover of compatibles of $\mathcal{A}$. From the point of view of state minimization, a compatibility constraint designates that states s and t may be in the same compatible of the domain of v. An incompatibility constraint (s,t), on the other hand, means that s and t may never be present in any compatible of the FSM, nor in any compatible of the domain of v. From the point of view of *state encoding*[3], a compatibility constraint (s,t) implies that s and t need not be encoded with *disjoint codes* in any *valid encoding* of machine $\mathcal{A}$. An incompatibility constraint obliges that disjoint codes be assigned to s and t in every valid encoding of $\mathcal{A}$.

Given an FSM $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$, the cardinality of θ depends on the nature of functions δ and λ. However, we may easily compute the bounds within which $|\theta|$ varies. By Definition 4.2, $(s,s) \in \theta$ for every $s \in S$. The minimum cardinality of θ is thus $|S|$, corresponding to a situation where every two distinct states are incompatible. The upper bound occurs when every two states are compatible, corresponding to $|\theta| = (|S|)^2$.

Consider now the incompatibility relation ι. Since it is not reflexive, the lower bound for ι is the empty set, when all states of the FSM are pairwise compatible, and the machine can be implemented as a combinational one. If all possible pairs of distinct states belong to ι, there is no compatible in the FSM, in which case the machine is already reduced, and it is thus itself the solution of the SM problem. The cardinality of ι is thus comprised between 0 and

$$2 \binom{|S|}{2} = (|S|)^2 - |S| .$$

This happens due to two facts:

1. ι is symmetric: ((s,t) in ι implies (t,s) in ι, for every s, t in S, generating the 2 multiplicand);
2. (s,s) is never in ι, for any s.

4.4 Covering Constraints

A set of compatibles of an FSM can be a solution of the SM problem if, among other conditions, it is the domain of some cover of the set of states of the FSM. We have already defined a closed cover of compatibles in Section 4.1. Here we define *covering relations* overlooking the closedness requirements.

Definition 4.7 (Covering relations) *Given an* FSM $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$ *and a set of compatibles* $K \in \mathcal{P}(S)$[4] *of* $\mathcal{A}$*, the* **covering relation** *corresponding to* K *is the binary relation* $\langle K, S, \gamma \rangle$*, where*

$$\gamma = \{(k,s) \mid k \in K \text{ and } s \in k\}.$$

[3]See Chapter 5 for a discussion of state encoding, as well as of the related definitions.

[4]Recall that the notation $\mathcal{P}(S)$ stands for the powerset of S (cf. Definitions 3.1).

The set K is the domain of a cover of S iff $\forall s \in S,\ \exists (k,s) \in \gamma$. In this case, each pair in γ is called an ***elementary covering constraint*** *of machine $\mathcal{A}$.*

From the point of view of SM, a covering constraint (k, s) means that if K is the domain of a closed cover of S, the compatible k corresponds to a state in some machine that simulates $\mathcal{A}$, and this state accounts for the behavior of state s in the original machine.

The number of covering constraints in a solution of the SM problem for a machine with $|S|$ can be very large, since the cardinality of K is bounded by $\mathcal{O}(2^{|S|})$. However, if we limit attention to minimum solutions of the SM problem, the cardinality of the set of compatibles K cannot be larger than $|S|$, and it is at least a singleton. The lower bound on the cardinality of a covering relation graph γ that corresponds to such a cover of S is thus $|S|$, and it is obtained in two trivial cases:

1. when K is a singleton (meaning that all states are compatible);
2. when $|K|=|S|$ and every element $k \in K$ has unit cardinality (corresponding to a machine without any compatible pair of states).

To attain an upper bound for the cardinality of this covering relation, we consider a set K containing the maximum number of elements, $|S|$, each element having the largest possible cardinality, i.e. $(|S| - 1)$. This corresponds to $|\gamma| = (|S|)^2 - |S|$.

4.5 Closure Constraints

Class sets, as introduced in Section 4.1, express the non-trivial closure conditions related to compatibles of an FSM. If we restrict the attention to compatible pairs of states only, we can still represent all non-trivial closure conditions imposed by compatibles of this FSM without loss of generality [72]. This can be done by defining a binary relation on its set of state pairs.

Definition 4.8 (Closure relation) *Let $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$ be an FSM and θ be the graph of the compatibility relation extracted from the description of $\mathcal{A}$, according to Definition 4.6. The* ***closure relation*** *of machine $\mathcal{A}$ is a binary relation $\langle \mathcal{P}(S), \mathcal{P}(S), \sigma \rangle$*

$$\sigma = \{(\{s,t\}, \{k,l\}) \mid (s,t), (k,l) \in \theta,\ s \neq t \ \textit{and}\ \{k,l\} \in P_{\{s,t\}}\},$$

where $P_{\{s,t\}}$ denotes the class set of the set $\{s,t\}$. The elements of the graph σ are called ***elementary closure constraints*** *imposed by $\mathcal{A}$ on the compatible state pairs.*

Note that σ contains only non-trivial constraints, since it is built based on the class set concept. Indeed, due to the class set definition, every element in the class set of a pair of compatibles $P_{\{s,t\}}$ contains exactly two states.

From the standpoint of SM, a closure constraint $(\{s,t\}, \{k,l\})$ means that if the states s and t are present in the same compatible of the domain of a minimum closed cover of compatibles of $\mathcal{A}$, the states k and l must also be in the same compatible of the same set. From the point

of view of SA, it means that assigning non-disjoint codes to s and t forces the codes assigned to k and l also to be non-disjoint.

If, for a given FSM, the graph σ is empty, any cover of compatibles is automatically closed, and the search for optimal solutions of the SM problem is simplified. The bounds on the number of distinct elementary closure constraints for a machine with $|S|$ states are obtained as follows: the lower bound is obviously 0; the upper bound is obtained for a relation σ where every possible pair of distinct states is present and imply all other pairs. The number of distinct pairs is

$$\binom{|S|}{2} = \frac{|S|^2 - |S|}{2}.$$

Since σ is not necessarily symmetric $(\{s,t\},\{k,l\})$ is distinct from $(\{k,l\},\{s,t\})$. Thus, $|\sigma|$ is at most

$$(\frac{|S|^2 - |S|}{2})((\frac{|S|^2 - |S|}{2}) - 1) = \frac{|S|^4 - 2|S|^3 - |S|^2 + 2|S|}{4}.$$

4.6 Generation of the SM Constraints

We introduce now the method due to Paull and Unger [91] to generate the compatibility, incompatibility and closure constraints at once. This method relies upon the construction of a *compatibility table*. Given a cover, the generation of the associated covering constraints is straightforward, and will not be addressed here, because we are not interested in explicitly generating covers of compatibles. The generation of this cover, based on the compatibility and closure constraints is the problem that Pfleeger showed to be NP-complete [95].

Example 4.1 Consider an FSM $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$ with:

$$I = \{001, 010, 011, 100, 101, 110, 111\},\ S = \{0, 1, 2, 3, 4, 5, 6, 7\},\ O = \{0, 1\}$$

and δ, λ defined in Table 4.1 as a flow table. Note that the input and output fields are already binary encoded.

The ***compatibility table*** contains one entry for each pair of distinct states in the machine. The objective of the method is to iteratively fill the entries of the table with the necessary conditions for each pair of distinct states to be compatible, if any.

The first step consists in searching the flow table for state pairs that are distinguishable by an input sequence of unit length. If a pair is distinguishable in this step, their elements are incompatible states. To denote this fact, we insert a cross ($\times$) in the corresponding entry. If the states are not distinguishable with a sequence of unit length, we fill the corresponding entry with all class sets for the pair. If no such class set exists for a given pair of states, the states in it are fully compatible, which is denoted by a check mark ($\surd$) in the compatibility table. The result of this first step for our example is shown in Figure 4.1(a).

After the first step, transitively incompatible pairs were left uncrossed. For instance, we can see that the pair $(6, 7)$ would form a compatible iff the pairs $(0, 4)$, $(1, 3)$ and $(2, 3)$ were

Table 4.1: Flow table for example 4.1

state \ input	001	010	011	100	101	110	111
0	0,0	-,-	6,0	3,1	1,0	0,-	-,-
1	1,0	6,1	0,-	-,-	0,-	0,1	-,-
2	1,0	6,1	0,1	-,-	-,-	-,-	7,0
3	1,-	3,-	0,-	-,-	1,-	3,-	0,1
4	1,0	2,-	-,1	5,1	4,1	7,0	-,-
5	0,1	3,0	6,1	1,0	1,-	3,-	0,1
6	-,-	3,-	-,-	1,-	1,0	-,-	0,-
7	-,-	2,1	-,-	3,1	-,-	7,0	4,0

compatible. Examining the entry associated to the first of these pairs, we note that $(0, 4)$ is an incompatible pair. Thus, $(6, 7)$ is also an incompatible pair, and its entry in the compatibility table can be crossed. This process continues, working over the initial compatibility table until an iteration over all entries is executed without any new entry being crossed. The final compatibility table obtained for our example appears in Figure 4.1(b).

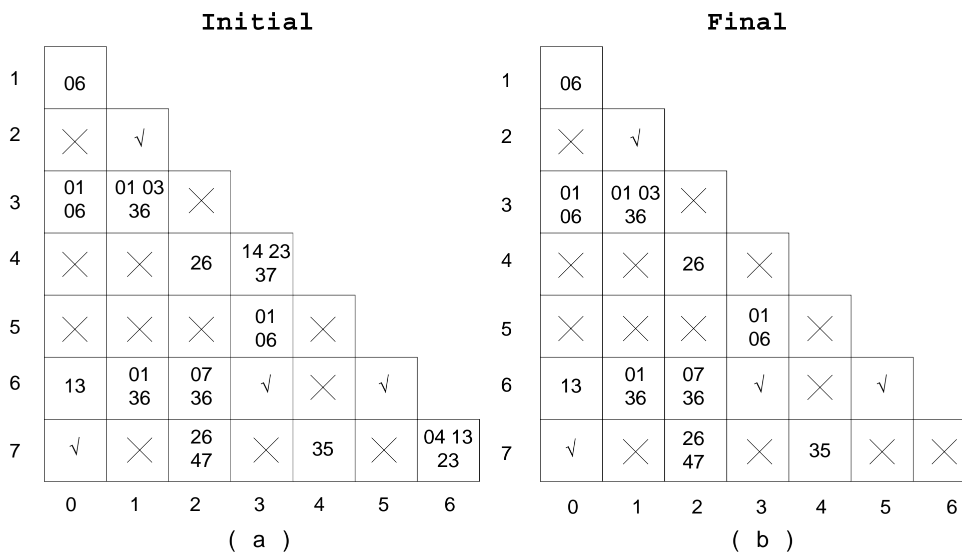


Figure 4.1: Initial and final compatibility tables for example 4.1

In the final table, each crossed entry corresponds to an elementary incompatibility constraint, while each pair of states inside some entry represents an elementary closure constraint. This information can be visualized through graphs. A labeled undirected graph, called ***merge graph*** (MG) is used to express the compatibility constraints. Each vertex in MG is associated to a state, and an edge exists between two vertices if the associated states are compatible, the trivial self-loops being omitted. Edges are labeled with the non-trivial conditions needed to guarantee the compatibility between the states. Another graph, called ***compatibility graph*** (CG) is useful in dealing with the closure constraints. This is a directed graph where each vertex represents a (non-trivial) compatible pair of states. An edge goes from a source vertex to a sink

vertex iff the sink vertex corresponds to a needed non-trivial condition for the source vertex pair to be compatible. Figure 4.2(a) and Figure 4.2(b) present the MG and CG graphs for our example, respectively.

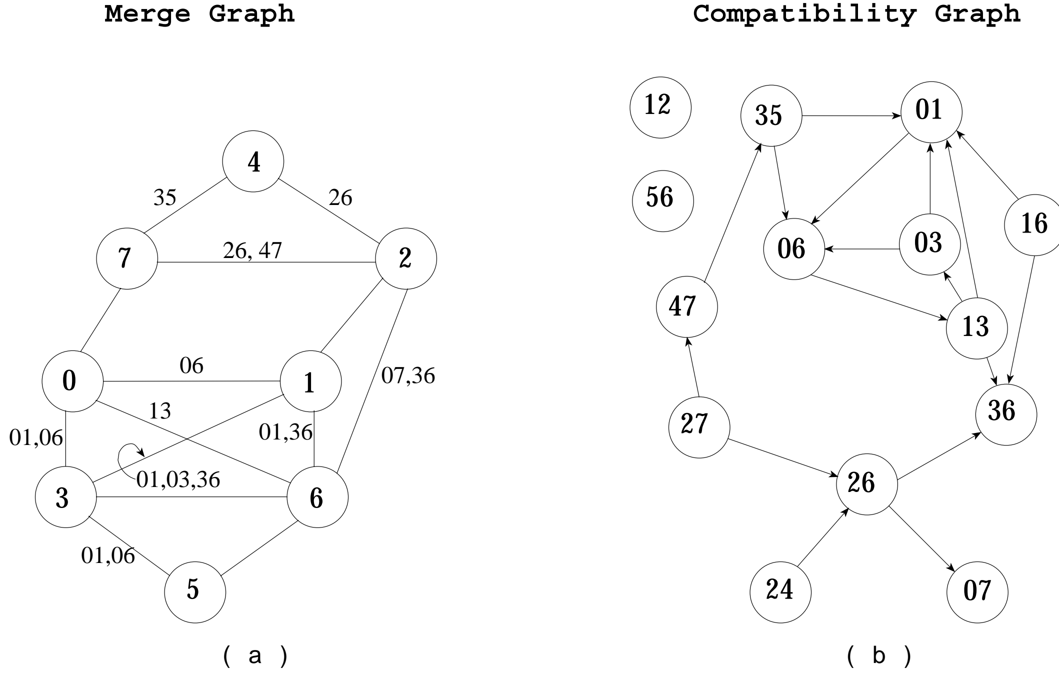


Figure 4.2: Merge and compatibility graphs for example 4.1

■

4.7 Complexity of the SM Constraints Generation

Let $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$ be an FSM and let $p = \mid I \mid$ and $q = \mid S \mid$ be the respective cardinalities of the input alphabet and of the state set of $\mathcal{A}$. To give an exact measure of the hardness of the above method, we now proceed to a worst-case analysis of its space and time complexity, to prove that they are $\mathcal{O}(p.q^2)$ and $\mathcal{O}(p.q^4)$, respectively. We also introduce some comments on how the running time of such algorithms can be reduced in practice.

The space complexity is easily computed, since, given the flow table, we build a single compatibility table that can be used during the application of the method. This table has exactly $\frac{q^2-q}{2}$ entries, and each entry contains a list of at most p pairs. The space complexity is then derived directly from the product of these quantities, which gives us $\mathcal{O}(p.q^2)$.

The time complexity is a little harder to compute. The first step of the method consists in generating the first version of the compatibility table using the flow table. Every pair of distinct states has to be inspected for output compatibility over all inputs. For a machine with p inputs and q states, this means $p\frac{q^2-q}{2}$ tests executed in the worst case.

Now, we have to calculate how many additional iterations are executed in the worst case, as well as how much computation is done at each iteration. This worst case occurs, for instance, in

a machine with no compatible pairs, but where all pairs of states are transitively incompatible except one[5]. In this case, the first step ends up with a table where only one crossed entry is present.

Suppose now that the structure of the flow table is such that only one new cross is added at each new step, which is possible, and corresponds to the maximum number of iterations of the method. Suppose also that this cross is obtained only when doing the last possible test at each iteration: this corresponds to a maximum amount of work executed at each iteration. Such a situation can be obtained, for instance, if each entry depends on p distinct pairs, all but one lexicographically inferior to the pair under analysis, considering we process entries in lexicographic order. We can then measure the number of operations needed in this worst case. Since each entry in a compatibility table can have at most p pairs inside it, the cost of each iteration is exactly $p(\frac{q^2-q}{2}) - j$, where j is the number of the iteration, varying from 0 (in the first step) to $\frac{q^2-q}{2} - 1$ (in the last step). Thus, the total number of steps in the worst-case situation is given by the following summation:

$$\begin{aligned}
\sum_{j=0}^{\frac{q^2-q}{2}-1} (p\frac{q^2-q}{2} - j) &= p\frac{q^2-q}{2} \sum_{j=0}^{\frac{q^2-q}{2}-1} 1 - \sum_{j=0}^{\frac{q^2-q}{2}-1} j \\
&= p(\frac{q^2-q}{2})^2 - \frac{(\frac{q^2-q}{2}-1)\frac{q^2-q}{2}}{2} \\
&= p\frac{q^4-2q^3+q^2}{4} - \frac{q^4-2q^3-q^2+2q}{8}
\end{aligned}$$

This result shows that we can design algorithms to implement the method whose running time is bounded by $\mathcal{O}(p.q^4)$. Nevertheless, some heuristic techniques can be added to the method to make it run faster. For example, we can solve the above worst case in only two steps, by adequately ordering the states and using a look-up table (implemented by a hash table, e.g.) to keep the incompatible pairs information up-to-date. A good heuristic ordering strategy is to test first states which are either fully compatible or incompatible with many other states. This ordering can be performed immediately after building the initial compatibility table.

[5]No FSM may have only transitively incompatible pairs of states. At least one has to be fully incompatible such that this condition is propagated to all other pairs.

Chapter 5

State Assignment Constraints

The hardware implementation of FSMs relies upon the existence of two-state devices. A transistor configured to work in either ON or OFF states provides a typical elementary device used in such implementations. Two-state devices can directly manipulate information if and only if the information has a binary character. The elements of the sets defining an FSM $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$, given generally as symbolic information, must thus be translated into *binary* (also called *Boolean*) codes, to allow the construction of a hardware version of $\mathcal{A}$. In this way, a fundamental step in the process of FSM synthesis is the encoding of the input alphabet I, the output alphabet O, and the state set S. As a consequence of this assignment process, the discrete functions δ and λ must accordingly be replaced by adequately designed switching functions.

In the next Section, we characterize the FSM assignment problem in general, and the state assignment problem in particular. Succeeding Sections propose the decomposition of the SA problem into its constituent constraint sets, discuss techniques available to generate such constraints and study the complexity of these techniques.

5.1 The FSM Assignment Problem

Definitions 5.1 (Assignment) *Given an integer n, an* ***assignment*** *or* ***encoding*** *of a set S is a mapping or complete function*

$$e : S \longrightarrow \mathcal{P}(\mathcal{B}^n),$$

where $\mathcal{P}(\mathcal{B}^n)$ stands for the powerset of $\mathcal{B}^n$. The integer n is called the ***length*** *of the assignment. For every $s \in S$, the image $e(s)$ is called the* ***code*** *of the element s. Two codes $e(s)$, $e(t)$ are* ***disjoint*** *iff $e(s) \cap e(t) = \emptyset$, otherwise the codes are* ***intersecting****. An assignment that is an injection and where codes are pairwise disjoint is an* ***injective encoding****, in which case*

$$\forall (s, t \in S),\ s \neq t \Longrightarrow e(s) \cap e(t) = \emptyset.$$

Any assignment that is not injective is called ***non-injective****.*

A ***cube assignment*** *or* ***cube encoding*** *of S is an assignment e of S where every code $e(s)$ is the satisfying set of some cube, i.e. there is a switching cube $c(\mathbf{x})$ over $\mathcal{B}^n$ such that*

$$c(\mathbf{x}) = 1 \Leftrightarrow \mathbf{x} \in e(s).$$

Let the codes of a cube assignment e be represented as three-valued vectors $\mathbf{v} = v_{n-1} \dots v_0$, and the cardinality of S be q. The three-valued column vectors obtained by selecting all elements v_i for some i, over all codes $e(s), s \in S$, have the form $(v_{i,0} \dots v_{i,q-1})^T$, and are called ***columns*** *of the encoding. The exponent T notation indicates that this corresponds to a column of the encoding (i.e. the transpose of the row vector).*

A ***functional assignment*** *or* ***functional encoding*** *of S is an assignment where every code is a singleton. Accordingly, a functional assignment is redefined as a function*

$$e : S \longrightarrow \mathcal{B}^n,$$

without loss of generality. Any assignment that is not functional is called ***non-functional****.*

The concept of assignment is limited here to fixed-length codes. We may thus refer to the encoding length also as the ***code length***. In the rest of this work, we restrict attention to functional and/or cube encodings. Correspondingly, Boolean codes will be represented as either Boolean tuples or as three-valued vectors, respectively. We adopt herein the simplified notation $\mathbf{x} = x_{n-1} \dots x_1 x_0$ to represent a tuple of the set $\mathcal{B}^n$, and call it a ***Boolean vector***.

Definition 5.2 (Discrete function satisfaction) *Let $f : S \longrightarrow L$ be a discrete function and $f_e : \mathcal{B}^p \longrightarrow \mathcal{B}^q$ be a general switching function. We say that f_e* **satisfies** *f under the two assignments $\varphi : S \longrightarrow \mathcal{B}^p$ and $\psi : L \longrightarrow \mathcal{B}^q$ if these assignments are such that*

$$\forall (s \in S) \;\; f(s) = l \Longrightarrow f_e(\varphi(s)) = \psi(l).$$

We may now define a kind of finite automaton directly related to the FSM assignment problem.

Definition 5.3 (Assigned finite automaton) *Given three integers l, m, and n, an* **assigned** *or* **encoded finite automaton** *is an algebraic structure of the form $\mathcal{A}_e = \langle \mathcal{B}^l, \mathcal{B}^m, \mathcal{B}^n, \delta_e, \lambda_e \rangle$ where:*

1. *$\mathcal{B}^l$ is the* **input alphabet***;*
2. *$\mathcal{B}^m$ is the finite* **set of states***;*
3. *$\mathcal{B}^n$ is the* **output alphabet***;*
4. *δ_e is a general partial switching function:*

   $$\delta_e : \; \mathcal{B}^l \times \mathcal{B}^m \longrightarrow \mathcal{B}^m;$$

   called **next state** *or* **transition function** *of $\mathcal{A}_e$; given the pair $(\mathbf{i}, \mathbf{ps}) \in \mathcal{B}^l \times \mathcal{B}^m$, suppose that $\delta_e(\mathbf{i}, \mathbf{ps})$ is specified; then, $\mathbf{ns} = \delta_e(\mathbf{i}, \mathbf{ps})$ is the* **next state** *of the automaton $\mathcal{A}_e$ after receiving the input $\mathbf{i}$, when in the present state $\mathbf{ps}$;*

5. *λ_e is a general partial switching function:*

$$\lambda_e : \mathcal{B}^l \times \mathcal{B}^m \longrightarrow \mathcal{B}^n,$$

called ***output function*** *of $\mathcal{A}_e$; given a pair $(\mathbf{i}, \mathbf{ps}) \in \mathcal{B}^l \times \mathcal{B}^m$, suppose that $\lambda_e(\mathbf{i}, \mathbf{ps})$ is specified; then, $\mathbf{o} = \lambda_e(\mathbf{i}, \mathbf{ps})$ is the* ***output*** *of the automaton $\mathcal{A}_e$ after receiving the input $\mathbf{i}$, when in the present state $\mathbf{ps}$.*

The pair $(\delta_e(\mathbf{i}, \mathbf{ps}), \lambda_e(\mathbf{i}, \mathbf{ps}))$ is called a ***transition*** *of automaton $\mathcal{A}_e$.*

Definition 5.4 (Automaton satisfaction) *Given two finite automata $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$ and $\mathcal{A}' = \langle I', S', O', \delta', \lambda' \rangle$, we say that $\mathcal{A}'$* ***satisfies*** *$\mathcal{A}$ if it is possible to define three functions:*

$$\begin{aligned} \varphi &: I \longrightarrow I'; \\ \xi &: S \longrightarrow S'; \\ \psi &: O \longrightarrow O', \end{aligned}$$

such that for all $i \in I$, $s \in S$, δ' and λ' obey to the conditions

$$\delta(i, s) \ \textit{specified} \implies \delta'(\varphi(i), \xi(s)) = \xi(\delta(i, s)), \tag{5.1}$$

$$\lambda(i, s) \ \textit{specified} \implies \lambda'(\varphi(i), \xi(s)) = \psi(\lambda(i, s)). \tag{5.2}$$

A necessary and sufficient condition for an assigned finite automaton $\mathcal{A}_e$ to implement an FSM $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$ is that it satisfies $\mathcal{A}$.

Given the above definitions, we may envisage a simple procedure to transform a symbolic automaton into an encoded automaton that satisfies the initial specification and is directly implementable using two-state devices. This procedure is based on the simplifying assumption that to each distinct input, state or output value of the original specification we make correspond one distinct input, state or output code in the encoded automaton, respectively. This clearly implies the use of functional injective encodings.

In this case, the first requirement imposed on $\mathcal{A}_e$ is the minimum length of the Boolean codes used to represent the sets I, S, O. The lower bounds for the integers l, m and n are,

$$l \geq \lceil \log_2(|I|) \rceil, \quad m \geq \lceil \log_2(|S|) \rceil, \quad n \geq \lceil \log_2(|O|) \rceil, \tag{5.3}$$

respectively, where the notation $\lceil \rceil$ designates the smallest integer not smaller than the expression between $\lceil$ and $\rceil$.

After determining the integers l, m, n using inequalities (5.3), it suffices to choose *any* three functional injective assignments:

$$\begin{aligned} \varphi &: I \longrightarrow \mathcal{B}^l; \\ \xi &: S \longrightarrow \mathcal{B}^m; \\ \psi &: O \longrightarrow \mathcal{B}^n. \end{aligned}$$

Function φ is called an ***input encoding*** or ***input assignment***, function ξ is called an ***state encoding*** or ***state assignment***, and function ψ is called an ***output encoding*** or ***output assignment***.

Last, functions δ and λ need to be replaced by suitable general switching functions δ_e and λ_e. Given the above assignments and the automaton satisfaction definition, the obtainment of such switching functions is immediate. For every $i \in I$, $s \in S$, we define δ_e, λ_e as

$$\delta_e(\varphi(i), \xi(s)) = \xi(\delta(i, s)) \text{ and}$$

$$\lambda_e(\varphi(i), \xi(s)) = \psi(\lambda(i, s)).$$

Together, these functions specify the input-output behavior of the combinational circuit (CC) in the sequential synchronous implementation of Figure 3.1. Additionally, we may easily show that the encoded automaton generated by this procedure satisfies the initial specification, by noting that such encodings are just a renaming of the original values. Now, consider the general FSM assignment problem statement below:

Problem Statement 5.1 (FSM assignment) *Given the FSM $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$, compute the three assignments φ, ξ and ψ, and three integers l, m, n leading to an encoded FSM $\mathcal{A}_e = \langle \mathcal{B}^l, \mathcal{B}^m, \mathcal{B}^n, \delta_e, \lambda_e \rangle$ that satisfies $\mathcal{A}$ and "optimizes" the combinational circuit CC of the corresponding sequential synchronous hardware implementation.*

The simple procedure we just provided meet all requirements of the above problem statement, except for the (crucial) optimization requirement, which requires additional comments:

1. the word "optimizes" should of course receive an accurate definition; such a definition is application dependent, and it can put the emphasis on one of the following criteria: area of the circuit CC, its input-output delay or its dissipated power. In practical situations, we shall usually adopt some acceptable trade-off among these often conflicting goals;
2. while simple to formulate, the FSM assignment problem is an extremely difficult one to solve exactly; the natural strategy is then to concentrate on restricted situations, easier to deal with:
   (a) first, one often considers subproblems of the general problem; most of the available literature is devoted to the SA subproblem: one assumes that the input and output signals are given by their binary representations, and deal with a mixed symbolic-Boolean representation. In this case, the only unknown assignment is the state assignment ξ. Such a situation is frequently encountered in the design of the controller in an algorithmic state machine [36], when the data path design has already been completed. Since we are interested in studying the relationship between the SM and SA problems, this is the subproblem we consider in this work;
   (b) one may consider restricted optimization goals: most of the available techniques are oriented towards area minimization;
   (c) one may also introduce extraneous conditions that will guide the solution, by allowing an accurate and simple definition of the optimization goal(s); such conditions can be:

  - of architectural nature: we can for example assume that the combinational circuit will be implemented as a PLA, and then measure the size of this PLA by the corresponding number of product terms. We can also assume that the circuit will be implemented using standard cells from an appropriate library, and then measure the size of the corresponding realization by the number of literals in the associated multilevel Boolean network. The advent of VLSI programmable devices has pushed researchers to consider the efficient implementation of FSMs using these components. Here, the area-based cost functions become useless, and new ways of evaluating the implementation cost must be provided;
  - of topological nature: we can indeed simplify the problem by limiting the design search space to a particular class of circuits, for example the two-level circuits; obviously, the larger the design space, the better the optimization possibilities;

3. even with the various types of simplifying assumptions we just discussed, assignment problems remain difficult and they always require the solution of non-polynomial complexity subproblems. In many practical situations, this precludes the use of rigorous solutions, which orient us toward the use of heuristic techniques and approximate optimization.

Since the most important issue in modern FSM logic synthesis is the optimization of the combinational part (CC)[38], encoding techniques play an important role in this field. De Micheli depicted four relevant encoding problems frequently found in VLSI logic design [43]:

P1 - find an encoding of the inputs (or some inputs) of a CC that optimizes it;

P2 - find an encoding of the outputs (or some outputs) of a CC that optimizes it;

P3 - find an encoding of both the inputs and the outputs (or some inputs and some outputs) of a CC that optimizes it;

P4 - find an encoding of both the inputs and the outputs (or some inputs and some outputs) of a CC that optimizes it, such that the encoding of the inputs is the same as the encoding of the outputs (or the encoding of some inputs is the same as the encoding of some outputs).

Solving the general assignment problem for FSMs is equivalent to solving **P4**. On the other hand, solving the SA problem alone is also equivalent to solving **P4**. This happens because states are at the same time inputs and outputs of the combinational part of the FSM, due to the feedback present in sequential synchronous implementations of circuits. The next Example gives an idea of the complexity of solving **P4** exactly.

Example 5.1 (Exhaustive state assignment) Let $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$ be an FSM with $|S| = q$ states. Suppose also that we simplify the problem by using a fixed length encoding with the least number of bits ($\lceil \log_2(q) \rceil$) to represent states. Since we have at least q distinct codes to assign to the q states, the number of possible state assignments is at least $q!$. In order to find the best assignment, we should generate each of them and then minimize the resulting encoded machine. The best exact solution would be found after at least $q!$ minimizations, supposing we had an exact minimizer at our disposal. Since the minimization problem itself is a very complex one, we cannot envisage such an exact solution of the state assignment problem, except for very small problems of little practical interest.

∎

In Section 1.2.1.2, the various methods proposed to solve the SA problem have been partitioned into two classes we now restate:

1. the ***classical methods***: which start by selecting the number q of internal state code bits (usually the minimum possible value, $\lceil \log_2(q) \rceil$); these methods then compute the state assignment ξ to minimize the cost of the combinational circuit CC;
2. the ***symbolic methods***: which carry out an initial cost minimization in symbolic form, with the use, for instance, of a multiple-valued counterpart of a prime implicant based method, or symbolic factorization; these methods attempt then to mimic the symbolic optimization process on its binary coded counterpart, trying to select a code with a minimum possible length.

In conclusion, a trivial solution for the FSM assignment problem always exists, but it is not guaranteed to be an acceptable one. Exhaustive strategies to obtain solutions for the problem are not feasible. The search for an optimized solution of the assignment problem passes through the formulation of constraints the assignment must respect in order to reach or to get close to the optimum solution.

Let us now define state assignment and valid state assignments formally.

Definitions 5.5 (FSM valid state assignment) *Given an* FSM $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$, *a* ***state assignment*** *or* ***state encoding*** *of* $\mathcal{A}$ *is a function*

$$\Xi : S \longrightarrow \mathcal{P}(\mathcal{B}^n).$$

Function Ξ *is a* **valid** *state assignment for* $\mathcal{A}$ *if there exists an* FSM $\mathcal{A}' = \langle I, \Xi(S), O, \delta', \lambda' \rangle$ *which simulates* $\mathcal{A}$. *A sufficient condition for* $\mathcal{A}'$ *to simulate* $\mathcal{A}$ *is that*

$$\delta'(i, \Xi(s)) = \Xi(\delta(i, s)), \quad \text{and} \quad \lambda'(i, \Xi(s)) = \lambda(i, s).$$

If $\mathcal{A}'$ *simulates* $\mathcal{A}$, *the encoding preserves the input/output behavior of* $\mathcal{A}$.

This is the most important concept to be retained throughout the present work, since the main objective of this thesis is to propose a change in the way valid state assignments are computed, in order to enhance the quality of VLSI design implementations of FSMs. Most previous works limited the search space of the SA problem to functional injective encodings, since these are always valid, as shown by the following theorem.

Theorem 5.1 (Functional injective SAs) *Given an* FSM $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$, *let* ξ *be a functional injective encoding of the form* $\xi : S \longrightarrow \mathcal{B}^n$, *with* $n \geq \lceil \log_2(|S|) \rceil$. *Then,* ξ *is always a valid state encoding of* $\mathcal{A}$.

Proof. Remark that if ξ is functional and injective, it consists in nothing but a renaming of the states in S. Then, the proof is evident from the above definition of state assignment.

∎

We present now a formulation of the SA problem constraints that can be used to enhance the results of the encoding process. The presentation is separated in two parts: the input constraints, discussed in Section 5.2, and the output constraints, object of Section 5.3. The presentation of input and output constraints evolves within the context of combinational circuits only. Nonetheless, this happens with no loss of generality for SA issues, since we have chosen to deal with sequential synchronous implementations only. Section 5.4 discusses the techniques available to generate the input and output constraints, and contains a brief analysis of the complexity of these techniques.

5.2 Input Constraints

During the case study presentation in Chapter 2, we have seen that symbolic minimization-driven SA methods were developed based on the first condition of Humphrey, as well as on the derived Liu assignments properties for synchronous FSMs. The symbolic minimization methods rely upon a two-step procedure: an encoding independent minimization step, followed by a constrained encoding step. In fact, symbolic minimization is a general technique to reduce the number of cubes needed to represent a Boolean encoding of a general discrete function. The encoding independent step in these methods generates a set of constraints that, if respected during the second step, conducts to an optimal (as for the number of product terms) two-level implementation of a combinational circuit.

In Section 2.1.2, we intuitively described the FSM SA input constraints arising from the symbolic minimization process, according to the grouping of entries in the flow table. Here we discuss input constraints formally, based on tabular representations of discrete functions.

Definition 5.6 (Function specification implementation) *Given a function tabular specification*

$$\Gamma = \{(A_1, z_1), \ldots, (A_n, z_n)\},$$

with domain S and codomain L, and two functional injective assignments

$$\varphi : S \longrightarrow \mathcal{B}^m, \quad \psi : L \longrightarrow \mathcal{B}^r,$$

we say that one ***implementation*** *of Γ according to these assignments is a complete function*

$$g : \mathcal{B}^m \longrightarrow \mathcal{B}^r$$

that satisfies the function tabular specification

$$\Delta = \{(\varphi(A_1), \psi(z_1)), \ldots, (\varphi(A_n), \psi(z_n))\}.$$

Note that Δ do correspond to a function specification, i.e. we have $\psi(z_i) \neq \psi(z_j) \Longrightarrow \varphi(A_i) \cap \varphi(A_j) = \emptyset$ for all $i, j = 1, \ldots, n$. This comes as a consequence of the condition (3.1) in the function tabular specification definition, as well as from the injectivity of φ.

A function $\psi : L \longrightarrow \mathcal{B}^r$ can be seen as an r-tuple of functions $\psi_k : L \longrightarrow \mathcal{B}$, such that for all $z \in L$, $\psi(z) = (\psi_1(z), \ldots, \psi_r(z))$. With some notational abuse we write

$$\psi = (\psi_1, \ldots, \psi_r).$$

Likewise, a function $g : \mathcal{B}^m \longrightarrow \mathcal{B}^r$ can be considered as an r-tuple

$$g = (g_1, \ldots, g_r)$$

of functions $g_k : \mathcal{B}^m \longrightarrow \mathcal{B}$.

In view of this, to have g as an implementation of the function specification Γ it is necessary and sufficient that for all $k = 1, \ldots, r$ the function $g_k : \mathcal{B}^m \longrightarrow \mathcal{B}$ satisfy the specification

$$\Delta_k = \{(\varphi(A_1), \psi_k(z_1)), \ldots, (\varphi(A_n), \psi_k(z_n))\}.$$

Theorem 5.2 (Function specification bounds) *Let $\Gamma = \{(A_1, z_1), \ldots, (A_n, z_n)\}$ be a discrete function specification with domain S and codomain L. Then, there exists at least one functional injective assignment φ of S such that:*

1. *φ has a length of at most $\mid S \mid$;*
2. *the number of rows in a reduced tabular specification of a complete function f_e that is an implementation of Γ according to φ and any functional injective assignment ψ of L is at most n.*

Proof. Consider the two functional injective assignments of S and L, respectively

$$\varphi : S \longrightarrow \mathcal{B}^m, \;\; \psi : L \longrightarrow \mathcal{B}^r,$$

and suppose that φ is constructed to satisfy the following condition:

CONDITION $C(\Gamma, m)$: There is a set of switching cubes of the form $c_i : \mathcal{B}^m \longrightarrow \mathcal{B}$ such that $sat(c_i) = C_i$, and for all $i = 1, \ldots, n$, and for all $s \in S$ we have:

$$s \in A_i \implies \varphi(s) \in C_i; \quad (5.4)$$

$$s \notin A_i \implies \varphi(s) \notin C_i. \quad (5.5)$$

Designating $\varphi(A_i)$ the set of elements $\varphi(s)$ such that $s \in A_i$, the property (5.4), valid for all $s \in S$ is equivalent to:

$$\varphi(A_i) \subset C_i. \quad (5.6)$$

Note first that for $m =\mid S \mid$, there is always one functional injective assignment $\varphi : S \longrightarrow \mathcal{B}^m$ satisfying condition $C(\Gamma, m)$, e.g. any 1-hot encoding of S. This demonstrates that φ constructed as described above verifies the first statement of the enunciate.

Now, since $sat(c_i) = C_i$, for all $i = 1, \ldots, n$, we have that the cubes c_i are defined by

$$c_i(y) = \begin{cases} 1 & \text{if } y \in C_i; \\ 0 & \text{if } y \notin C_i. \end{cases}$$

It is easy to verify that the general switching function $g = (g_1, \ldots, g_r) : \mathcal{B}^m \longrightarrow \mathcal{B}^r$, the components g_k $(k = 1, \ldots, r)$ of which are defined by

$$g_k = \bigvee_{i=1}^{m} \psi_k(z_i).c_i \tag{5.7}$$

is an implementation of Γ according to the assignments φ and ψ. Stating it otherwise, for $k = 1, \ldots, r$, the functions g_k defined by (5.7) satisfy the specifications

$$\Delta_k = \{(\varphi(A_1), \psi_k(z_1)), \ldots, (\varphi(A_n), \psi_k(z_n))\},$$

according to Definition 5.6 above. Indeed, if $y \in \varphi(A_j)$ then $y \in C_j$ due to (5.6), thus $c_j(y) = 1$ and $c_i(y) = 0$ for every $i \neq j$, due to (5.5). Thus,

$$g_k(y) = \bigvee_{i=1}^{m} \psi_k(z_i).c_i(y) = \psi_k(z_j).$$

In this way, for all $y \in \mathcal{B}^m$ and for $j = 1, \ldots, n$, we have

$$y \in \varphi(A_j) \Longrightarrow g_k(y) = \psi_k(z_j).$$

This being true for $k = 1, \ldots, r$, function g is an implementation of Γ according to the assignments φ and ψ. Additionally, all functions g_k (5.7) are disjunctions of cubes taken from the set $\{c_1, \ldots, c_n\}$, which demonstrates the second statement of the enunciate.

∎

To understand the relevance of Theorem 5.2, remember that a function tabular specification is the representation of a partial function (equivalently, of a set of complete functions), and that symbolic minimization (cf. Definitions 3.17) generates a function specification with the least number of rows. From the Theorem, we conclude that symbolic minimization determines an upper bound to both, the number of rows of a reduced tabular specification of the implementation of a function (i.e. the number of product terms in a sum-of-products implementation of the function) and the length of the encoding needed to attain this number of rows. Theorem 5.2 is an interpretation of the results published by de Micheli et al in [43]. This Theorem states also the conditions that must be respected in order to attain the bounds. These correspond to the implications (5.4) and (5.5), that determine a bijection between the sets A_i in the original function specification and the sets C_i in the encoded function specification.

The following Example will help to clarify the symbolic minimization process, as well as the subsequent constrained encoding step.

Example 5.2 (Symbolic minimization and constrained encoding) Let $L = \{l, m\}$ and $S = \{s0, s1, s2, s3\}$ be sets, and $f : S \longrightarrow L$ be a discrete function. Assume that $c(\mathbf{x}) =$

```
{s0} {l}
{s1} {m}
{s2} {l}     {s0,s2} {l}
{s3} {m}     {s1,s3} {m}

   (a)           (b)
```

Figure 5.1: Original and minimized cube tables for example 5.2

$l \wedge x^{(\{s0,s2\})}$ and $d(\mathbf{x}) = m \wedge x^{(\{s1,s3\})}$ are cube functions. Let f be represented by the tabular specification of Figure 5.1(a).

Figure 5.1(b), on the other hand, represents an equivalent tabular specification obtained after symbolic two-level minimization. The row merging obtained by the minimization step is obvious from the Figure. This specification states that f is in fact the disjunction of the cubes $c(\mathbf{x})$ and $d(\mathbf{x})$.

The minimization was achieved using the ESPRESSO program, parameterized to perform exact multiple-valued minimization. The row cardinality of the minimized cube table, which is 2, is thus an upper bound for the minimized encoded specification row cardinality corresponding to any assignment of f, according to Theorem 5.2. Let us now consider how the symbolically minimized specification may guide the encoding step.

Figure 5.2 displays three possible functional injective assignments of the symbols in S. After substituting the codes of each assignment into the original cube table describing f, the encoded cube tables thus obtained were submitted to logic minimization (using again the program ESPRESSO), producing the cube tables shown to the right of each encoding.

encoding	table	encoding	table	encoding	table
α(s0)=0001		α(s0)=00		α(s0)=00	00 {l}
α(s1)=0010		α(s1)=01		α(s1)=01	01 {m}
α(s2)=0100	0-0- {l}	α(s2)=10	-0 {l}	α(s2)=11	11 {l}
α(s3)=1000	-0-0 {m}	α(s3)=11	-1 {m}	α(s3)=10	10 {m}
(a)		(b)		(c)	

Figure 5.2: Three assignments and corresponding minimized cube tables for example 5.2

Note that the first two assignments respect the upper bound predicted by symbolic minimization, while the third assignment does not. Note also that, since the assignments are injective, each code is associated with a row in the original specification.

The first assignment in Figure 5.2(a) corresponds to one positional cube scheme representation of S, under the enumeration order evident from the naming of its elements. Actually, every positional cube encoding is a trivial solution of the constrained encoding problem as demonstrated by Theorem 5.2. However, the code length for this scheme is often too long, leading to the search of smaller length encodings, which becomes then the cost function to be minimized during the encoding. Also, note that the associated cube table has a domain that is defined as $\mathcal{B}^4 - \{0000\}$, since for the all 0s input value there is a conflict in the reduced cube table, which would prevent it from being a cube table. Excluding the all 0s value from the

domain is needed in every 1-hot encoding.

The second and third assignments, on the other hand, implement minimum length codes. The encoding of Figure 5.2(b) respects both restrictions imposed by the minimized cube table, i.e. $\{\alpha(s0), \alpha(s2)\}$ and $\{\alpha(s1), \alpha(s3)\}$ do not intersect. The last assignment in Figure 5.2(c) does not respect any of these restrictions, and accordingly gives a representation that has more product terms than the predicted upper bound. Such a situation happens because this encoding does not allow that the row merging performed in the symbolic minimization step occurs in the logic minimization step.

■

The constraints generated by symbolic minimization do not consider the outputs encoding influence at all. All the bounds and results of the encodings mentioned above are valid for any encoding of the outputs, which is clear from the enunciate of Theorem 5.2. To point out this fact, the outputs remained unencoded in the above Example. We may now define what we mean by a set of *input constraints* on the domain of a discrete function.

Definition 5.7 (Set of input constraints for a discrete function) *Let $f : S \longrightarrow L$ be a partial discrete function. A* **set of input constraints** *of f is a set of cube satisfying sets $C = \{C_1, \ldots, C_m\}$ such that $C_i = sat(c_i)$, with c_i a cube function of the form $c_i : S \longrightarrow L$, for all $i = 1, \ldots, m$, and*

1. *$dom(f) \subset \bigcup_{i=1}^{m} C_i$;*
2. *for each C_i $(i = 1, \ldots, m)$, f is constant or unspecified in $C_i \cap dom(f)$.*

Given a function specification $\Gamma = \{(A_1, z_1), \ldots, (A_n, z_n)\}$ of a discrete function f, one set of input constraints for f is clearly the set $\{A_1, \ldots, A_n\}$. These constraints on the encoding of f can be described by a binary relation as follows.

Definitions 5.8 (Input relation) *Consider a discrete function $f : S \longrightarrow L$ and a specification Γ of f, such that Γ is represented as pairs of the form*

$$(S_i \subset S, L_j \subset L).$$

The first coordinate of a pair in Γ is the **input part** *of the row, while the second is its* **output part***. Define an* **input relation** *on S, also called* **face embedding relation** *on S, as a binary relation $\langle \mathcal{P}(S), S, \phi \rangle$ such that*

$$\phi = \{(S_i, s_k) \mid (S_i, L_j) \in \Gamma,\ s_k \in S,\ s_k \notin S_i\}.$$

Each pair in the graph ϕ of this relation is an **elementary input constraint***, also called* **elementary face embedding constraint** *on the input set S of function f. A pair of the form (S_i, S_k) with $S_k \subset S$, is called a* **full input constraint** *or* **full face embedding constraint** *iff $S_k = \{s_k \mid (S_i, s_k) \in \phi\}$ and $S_i \cup S_k = S$. Full input constraints may always be translated into an equivalent set of elementary input constraints with cardinality $|S_k|$.*

The input relation ϕ obtained from a (eventually reduced) function specification describes the elementary constraints to be respected by an encoding, so as to achieve the bounds predicted by Theorem 5.2.

Let $n = \mid \Gamma \mid$. The cardinality of the input relation graph ϕ depends on the number of rows n in the specification, as well as on the number of symbols on the set S. The cardinality of ϕ is then bounded by $\mathcal{O}(n.q)$, where q is the number of elements in S.

From the standpoint of the SA problem, each elementary input constraint (S_i, s_k) tells that states in S_i must be encoded such that in some bit position all of them present the same bit value, while state s_k presents the complement of this value in the corresponding position.

5.3 Output Constraints

In the previous Section, we discussed how conditions imposed on the encoding of the inputs of a discrete function may lead to an optimized binary implementation of this function. In the present Section, we are interested in analyzing the conditions to impose on the outputs of a discrete function in order to obtain optimized implementations as well. Historically, the use of input constraints came first, with the development of symbolic minimization methods to apply to the SA problem. In an enhancement of the work in [43], de Micheli recognized the existence of one kind of output constraints, the so-called *dominance constraints* [40], and used it to ameliorate the results of the SA method he proposed in [43], that considered input constraints only. After this, Devadas and Newton treated the output encoding problem independently of both input encoding and SA, and proposed a systematic approach to the generation of a more thorough set of output constraints [45]. In their work, besides the dominance constraints, two other kinds of constraints appear: *disjunctive constraints* and *disjunctive-conjunctive constraints*. Dominance and conjunctive constraints are special cases of the disjunctive-conjunctive constraints. We do not discuss disjunctive-conjunctive constraints in this work.

5.3.1 Dominance Constraints

We introduce the principle behind dominance constraints by using an example.

Example 5.3 (Dominance constraints) Let $I = \{i0, i1\}$, $J = \{j0, j1\}$ and $O = \{o0, o1\}$ be sets and f a discrete function $f : I \times J \longrightarrow O$, whose behavior is represented by the cube table in Figure 5.3(a).

```
Original Table      Output constraint     Encoding respecting
                      reduced Table        output constraint
{i0}{j0,j1} o0
{i1}{j1}    o0    {i0,i1}{j0,j1} o0           α(o0)=01
{i1}{j0}    o1    {i1}    {j0}   o1           α(o1)=11

      (a)                (b)                      (c)
```

Figure 5.3: Cube table optimization using dominance constraints

Consider the first two rows in Figure 5.3(a). Since the cubes corresponding to them assert the same output value, these rows could be merged by the logic minimization step, for some adequate encoding of the input set $I \times J$. However, symbolic minimization alone cannot merge the two rows, since this would imply that the resulting cube and the cube associated with the last row would not be disjoint anymore.

In hardware implementations of functions, we need to specify how to generate one of the binary values only, the other being the default value. Thus, if we guarantee that the encoding of the outputs is such that the code assigned to the output value $o1$ never presents a default value in a bit position where $o0$ has the complement of this value, the input part of the last row cube can be considered as a don't care for the output value $o0$. In this case, the first two rows can be merged. Figure 5.3(b) shows the equivalent cube table, assuming this constraint is respected, while Figure 5.3(c) displays an encoding of the outputs that do respect the constraints.

■

If dominance constraints are considered during symbolic minimization, the upper bound predicted by using only input constraints may thus be further reduced.

Definition 5.9 (Dominance between Boolean vectors) *Let* $\mathbf{x}$ *and* $\mathbf{y}$ *be two Boolean vectors* $\mathbf{x} = x_{n-1} \dots x_0$ *and* $\mathbf{y} = y_{n-1} \dots y_0$ *over* $\mathcal{B}^n$*. Then,* $\mathbf{x}$ ***dominates*** $\mathbf{y}$ *iff*

$$\forall i, \quad x_i = 0 \Longrightarrow y_i = 0.$$

This relation is denoted by $\mathbf{x} \succ \mathbf{y}$*.*

We may now introduce a dominance relation of a discrete function under an encoding.

Definitions 5.10 (Dominance relation) *Consider a discrete function* $f : S \longrightarrow L$ *and an encoding* $\alpha : L \longrightarrow \mathcal{B}^n$*, for some integer* n*. Define a* ***dominance*** *relation on* L *under the encoding* α *as a binary relation* $\langle L, L, \mu \rangle$ *such that*

$$\mu = \{(j, k) \mid \alpha(j) \succ \alpha(k)\}.$$

Each pair in the graph μ *of this relation is an* ***elementary dominance constraint*** *on the output set* L *under* α*.*

The cardinality of the dominance relation graph μ is bounded by $\mathcal{O}(|L|^2)$.

5.3.2 Disjunctive Constraints

We introduce the nature of disjunctive constraints through the presentation of an example as well, to ease its understanding.

Example 5.4 (Disjunctive constraints) Let $I = \{i0, i1\}$, $J = \{j0, j1\}$, $O = \{o0, o1, o2\}$ be sets and f a discrete function $f : I \times J \longrightarrow O$, whose behavior is represented by the cube table in Figure 5.4(a).

Original Table	Disj. constraint reduced Table	Encoding respecting Disj. constraint
{i0}{j0} o0		α(o0)=11
{i0}{j1} o1	{i0} {j0,j1} o1	α(o1)=01
{i1}{j0} o2	{i0,i1}{j0} o2	α(o2)=10
(a)	(b)	(c)

Figure 5.4: Cube table optimization using disjunctive constraints

The cardinality of this cube table cannot be reduced by symbolic minimization, since all weights of its cubes are distinct. No possible dominance relation that could be established would produce a reduction in this cardinality either. However, suppose that there is a functional injective encoding α of the output set O such that the code of $o0$ is the disjunction (inclusive or) of the codes of the elements $o1$ and $o2$, i.e. $\alpha(o0) = \alpha(o1) \vee \alpha(o2)$. If this restriction holds, every cube with weight $o0$ in the original cube table can be alternatively represented by the disjunction of two cubes with the same input part and weights $o1$ and $o2$, respectively. Proceeding to such a substitution in Figure 5.4(a), we see then that the symbolic minimization process may now merge cubes, obtaining the minimized table of Figure 5.4(b). Figure 5.4(c) presents an encoding that respects the restriction, and may accordingly lead to a cube table with the number of rows predicted by symbolic minimization.

∎

Again, disjunctive constraints may provide further enhancement to the results of symbolic minimization, if they can be adequately generated. We may then introduce a disjunctive relation of a discrete function under an encoding.

Definitions 5.11 (Disjunctive relation) *Consider a discrete function $f : S \longrightarrow L$ and an encoding $\alpha : L \longrightarrow \mathcal{B}^n$, for some integer n. Define a* **disjunctive** *relation on L under the encoding α as a binary relation $\langle L, \mathcal{P}(L), \chi\rangle$ such that*

$$\chi = \{(j, \{k, l\}) \mid \alpha(j) = \alpha(k) \vee \alpha(l)\}.$$

Each pair in the graph χ of this relation is an **elementary disjunctive constraint** *of the output set L under α.*

The cardinality of the disjunctive relation graph χ is bounded by $\mathcal{O}(|L|^3)$.

5.4 Complexity of Generating the SA Constraints

We mentioned above the existence of techniques to generate all constraints related to the SA problem in synchronous FSMs. In the present Section we briefly restate these methods and discuss their complexity.

5.4.1 Generation of the Input Constraints

We have pointed out, in Section 5.2, that the generation of the input constraints relies upon a symbolic minimization step, which can be performed by adequate multiple-level minimization

methods. These methods are available in logic level minimizers like the programs ESPRESSO [16] and MCBOOLE [30].

The symbolic minimization process can then be seen to have a complexity equivalent to two-level logic minimization. However, there is a major difference between the growth rate of the execution time for a function described in terms of binary variables, and the same parameter for a discrete function described with multiple-valued variables. The internal representation of multiple-valued variables is obtained with the positional cube scheme. In this case, the execution time depends *linearly* on the size of the sets defining the function. For switching functions, however, this dependence is only *logarithmic*, since functions are directly represented using a switching cube scheme. The use of specific techniques to deal with multiple-valued variables does alleviate the problems, but cannot change its more complex nature. An example of program using such techniques is ESPRESSO-MV [99], which we employ in this work.

5.4.2 Generation of the Output Constraints

Dominance constraints were identified by de Micheli in [40]. He suggested a method which consists in generating face embedding *and* dominance constraints simultaneously, using the following technique. While performing symbolic minimization, instead of constraining the final cube table to be disjoint, one iteratively builds a partial ordering for the outputs. The final result is a set of face embedding constraints and a set of dominance constraints. The final set of constraints is satisfiable by construction, since the only way to violate a dominance relation would be defining cyclic constraints. This is avoided by the choice of a partial order, instead of a more general relation between the outputs. The program CAPUCCINO [40] implements the method. The obtained results, although superior to ordinary, input constraints-based encoding, are not very good. One reason is that the iterative method relies on the treatment of one output at a time. However, the final quality of the result depends strongly on the order in which these outputs are considered. To date, no satisfactory technique to select the outputs is available.

Ciesielski et al proposed a dedicated method that directly obtains all dominance and disjunctive constraints that may potentially produce a reduction in the size of the final cube table [26]. This method is exact.

When studying the input relations, we have seen that obtaining these relations poses a problem roughly as complex as the one of a two-level multiple-valued minimization step. Although this implies that the generation of input constraints is already a non-polynomial time problem, there are efficient heuristic methods for the generation of a feasible set of constraints. On the other hand, the methods available for creating output constraints do generate a larger set of possibly useful relations, and not directly a feasible subset of them. This is indeed the case for the exact method suggested by Devadas and Newton in [45] and for the method suggested by Ciesielski et al in [26].

As a conclusion, the author knows of no efficient method to generate output constraints for the SA problem. An additional problem with output constraints is that they, unlike the input constraints, may conflict among themselves, as well as with the input constraints. This question is addressed in some detail in the next Chapter.

Chapter 6

Relationship among SM and SA Constraints

Although we have already defined various constraints types in previous Chapters, a formal statement of the *constraint* general concept has not yet been furnished. In fact, we will delay this statement until Chapter 11, where the problem we want to treat will be fully delimited. For now, we rely on the intuitive notion that a constraint is a restriction imposed on the solutions of a problem, a way to delimit the search in the solution space to potentially "good" choices. In particular, we are interested in this thesis in studying constraints associated to encoding problems, which we call *encoding constraints*.

The basic idea of a constraint formulation of a problem is to show the possibility of simultaneously considering a *feasible* subset of constraints describing the problem with sufficient thoroughness. After this study, we must devise a method to satisfy the feasible subset of constraints such that an optimal solution of the problem is obtained.

One of the original contributions of this work is the proposition of the simultaneous consideration of the SM and SA problems within the scope of an FSM state encoding method using a constraint formulation. The goal of this Chapter is to derive the relationships existing between the constraints in each of the two problems. In the two previous Chapters, we separately analyzed each of the constraints sets of SM and SA. The result of this analysis has been the modeling of each constraint kind by means of a binary relation, the pairs of which are the elementary constraints imposed on the solution of one of the two, SM or SA problems. We are now able to analyze the correlation between the problems with precision.

Definitions 6.1 (Encoding constraint set feasibility) *A set of encoding constraints is* **feasible** *if all elements in it can be satisfied simultaneously by some encoding. Two encoding constraints are* **feasible** *if they do not imply conflicting encoding requirements, i.e. iff they alone form a feasible set of constraints.*

Even if the SM constraints were not originally devised in the scope of encoding problems, they may be reinterpreted under the SA standpoint. Chapter 4 advanced an intuitive view on this interpretation, which we now restate. Let $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$ be an FSM:

1. if a pair of states (s, t), $s, t \in S$ of $\mathcal{A}$ belongs to the graph ι of the incompatibility relation

of $\mathcal{A}$, no valid state encoding for this machine may associate intersecting codes with s and t;

2. let $\mathcal{A}' = \langle I, S', O, \delta', \lambda' \rangle$ be a machine that simulates $\mathcal{A}$, obtained from a closed cover of compatibles κ of $\mathcal{A}$; if a compatible $C \in dom(\kappa)$ and a state $s \in S$ of machine $\mathcal{A}$ form a pair (C, s) of the graph γ describing the covering relation of $\mathcal{A}$, and if C is the compatible associated with the state $s' \in S'$ of machine $\mathcal{A}'$, to any valid code of s' corresponds a valid code of s which intersects the code of s';

3. if four states s, t, u, v of $\mathcal{A}$ form a pair $(\{s, t\}, \{u, v\})$ belonging to the graph σ of the closure relation of $\mathcal{A}$, assigning intersecting codes to s and t can be part of a valid state encoding iff u and v also receive intersecting codes.

The above three remarks need to be considered in the scope of an encoding method, so as to allow that SM constraints be accounted for in the encoding process. To do so, we need to extend the assignments traditionally generated as solutions of a constrained state assignment problem. This is one of the main practical targets of this Chapter.

The next Section provides an informal discussion on the need and the usefulness of extending currently used constrained encoding methods. Follows Section 6.2, where we review the state splitting concept, from the standpoint of equivalent machines obtainment. The results of this Section are used to prove some important propositions of Section 6.3, which is dedicated to justify the use of extended concepts of assignment. The following two Sections deal with the violation of SM constraints by input constraints and vice-versa, revealing an interesting result about the use of *injectivity constraints* in the scope of the SA problem. At last, Section 6.6 concerns the conflicting requirements across constraints, giving particular attention to the relationship between the dominance and disjunctive output constraints, as well as to the possible conflicts between output constraints and the other constraint kinds.

6.1 Extending Constrained Encoding Assignments

The FSM assignment problem and the definition of assigned automaton, both stated in Section 5.1, assumed assignments of the more general type characterized in Definition 5.1. However, all along Chapter 5 we referred to functional injective assignments only. These assignments imply that every code is a single Boolean vector (functional assignment), and that no two distinct elements of the assignment domain receive identical codes. This is in fact the assumption of most constrained encoding methods addressing the SA problem alone [116, 119, 43]. This choice is justified, for instance, by the observation that in the SA problem, any functional assignment that is a solution of constrained encoding has code length less than or at most equal to a non-functional assignment [43]. As the code length is one of the most important cost functions to minimize during the encoding step, using functional assignments helps the satisfaction of this requirement. On the other hand, using injective assignments guarantees that every state behavior in the original machine will correspond to a unique state behavior in the equivalent assigned machine and vice versa.

If state minimization is considered during the encoding of an FSM, the above statement about the code length of a functional assignment with regard to non-functional ones is no longer true. We may show this by means of a simple counterexample.

Example 6.1 (Non-minimum functional assignments) Let $\mathcal{A} = \langle I, S, O, \delta, \lambda\rangle$ be an FSM. Let the sets in this machine be $I = \{0, 1, 2\}$, $S = \{s, t, u, v, x\}$, $O = \mathcal{B}$, while δ and λ are specified by Table 6.1.

Table 6.1: Flow table for example 6.1

state \ input	0	1	2
s	s,1	t,0	v,0
t	t,1	u,-	v,0
u	s,1	s,1	v,0
v	x,0	v,1	t,1
x	s,0	t,0	t,1

The merge graph resulting from the compatibility analysis for $\mathcal{A}$ is shown in Figure 6.1. There is obviously only one minimum closed cover of compatibles for this machine, whose domain is

$$\{\{s, t\}, \{t, u\}, \{v\}, \{x\}\}.$$

Thus, we see that a reduction $\mathcal{A}' = \langle I, S', O, \delta', \lambda'\rangle$ of $\mathcal{A}$ may be built based on this minimum closed cover, resulting in a state set S' with cardinality equal to 4. Any solution assignment of the SA problem for machine $\mathcal{A}$ will have a code length which is at least $\lceil \log_2(|S|)\rceil = 3$, while the state set S' of machine $\mathcal{A}'$ may be assigned codes with length 2.

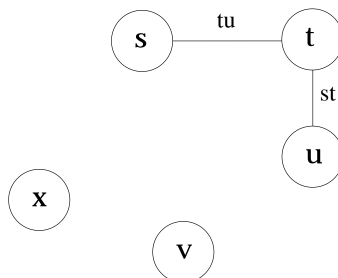


Figure 6.1: Merge graph for example 6.1

Since the reduction $\mathcal{A}'$ can be state encoded with a smaller encoding length than $\mathcal{A}$, we know that the behavior of $\mathcal{A}$ may be described with a smaller state encoding. The problem is to know if there are encodings of S leading *directly* to such a state encoding.

If the specification of $\mathcal{A}$ had only one compatible pair of states, say $\{s, t\}$[1], we intuitively see that encoding both s and t with a same code, i.e. using non-injective state encoding, would still allow the specified behavior of $\mathcal{A}$ to be obtained in the encoded machine.

However, in machine $\mathcal{A}$, the state t is present in two distinct compatibles, and thus cannot receive a single binary code. The solution to the encoding of t can only be obtained if three conditions hold:

1. the requirement of building functional encodings of the state set is relaxed;

[1]It should be clear that, in this case, no class set could exist

2. a code is assigned to t that contains at least two Boolean vectors;
3. we dispose of a method to adequately choosing which code to use when translating the original flow table into an equivalent encoded flow table.

We may advance that the cube encoding depicted in Table 6.2 is a valid encoding for machine $\mathcal{A}$.

Table 6.2: A valid non-functional, non-injective assignment for example 6.1

state	code
s	00
t	0-
u	01
v	10
x	11

■

The possibility of constructing valid non-injective, non-functional state encodings like the one in Table 6.2 to allow the simultaneous consideration of the SM and SA problems is demonstrated in Section 6.3. The next Section shows the state splitting techniques needed in those demonstrations.

6.2 State Splitting and Equivalent FSMs

State splitting is a technique counterpart of state reduction, in the sense that its objective is to "unreduce" a sequential machine. Given an FSM, applying state splitting to it generates an equivalent FSM with state set cardinality larger than the original machine. The concept has frequently been used for the decomposition of state machines [121, 63].

Definition 6.2 (Augmentation) *An* FSM $\mathcal{A}^* = \langle I, T, O, \delta^*, \lambda^* \rangle$ *is said to be an* ***augmentation*** *of an* FSM $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$ *iff* $\mathcal{A}^*$ *is simulated by* $\mathcal{A}$ *and* $|S| \leq |T|$.

In this Section, we shall propose a state splitting method to obtain, from a minimum closed cover of compatibles κ of an FSM $\mathcal{A}$, an augmentation $\mathcal{A}^*$ of $\mathcal{A}$, such that κ maps into κ^*, a minimum closed cover of $\mathcal{A}^*$ with the property of being a partition of the state set T of $\mathcal{A}^*$. The method developed herein will serve to demonstrate some important results in the next Section.

Method 6.1 (Partition–κ state splitting) Given an FSM $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$ and a closed cover of compatibles κ of $\mathcal{A}$, construct a new machine $\mathcal{A}^* = \langle I, T, O, \delta^*, \lambda^* \rangle$ as follows.

1. make T a set with cardinality $\sum_{s \in S} n_s$, where the integer values n_s are chosen such that $n_s = | \{k_l \in dom(\kappa) \,|\, s \in k_l\} |$. In this way, each state $s \in S$ is associated with n_s states in T, and n_s is the number of distinct compatibles of $dom(\kappa)$ containing s. Designate the states in T as $t_{s,l}$, with $s \in S$ and l being an index of a compatible k_l, such that $s \in k_l$;

2. construct λ^* such that for every input letter $i \in I$ and for each $t_{s,l} \in T$

$$\lambda^*(i, t_{s,l}) = \lambda(i, s);$$

3. construct δ^* as follows:

for every input letter $i \in I$, do:

for every $k_l \in dom(\kappa)$, do:

(a) compute the index b of *some* class of κ that contains the compatible implied by k_l under the input letter i, i.e. find a k_b such that $k_b \supseteq \delta(k_l, i)$;

(b) for every $s \in k_l$, make

$$\delta^*(i, t_{s,l}) = t_{\delta(i,s),b}.$$

The machine $\mathcal{A}^*$ is called a ***partition augmentation*** of $\mathcal{A}$.

∎

Below we will demonstrate that the partition–κ method generates an augmentation of the initial machine. Before doing so, let us illustrate the method through an example, to clarify its inner workings.

Table 6.3: Flow table for original FSM in example 6.2

state \ input	0	1
a	a,0	b,0
b	b,0	c,-
c	b,-	c,1
d	a,0	e,1
e	a,1	d,1

Example 6.2 (Partition–κ application) Let $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$ be a machine such that $I = O = \mathcal{B}$ and $S = \{a, b, c, d, e\}$. Table 6.3 defines the next state and output functions δ and λ.

Let κ be a minimum closed cover of compatibles of $\mathcal{A}$, such that $dom(\kappa) = \{k_1 = \{a, b\}, k_2 = \{b, c, d\}, k_3 = \{c, e\}\}$. The verification that this set corresponds to the domain of a cover verifying minimality and closedness is straightforward. Let us now apply the partition–κ method to this example, in order to generate a new machine, $\mathcal{A}^* = \langle I, T, O, \delta^*, \lambda^* \rangle$. The first step begins with the determination of the n_s values which are

$$n_a = 1, \;\; n_b = 2, \;\; n_c = 2 \;\; n_d = 1, \;\; n_e = 1.$$

Considering the compatibles in $dom(\kappa)$, we have $T = \{t_{a,1}, t_{b,1}, t_{b,2}, t_{c,2}, t_{d,2}, t_{c,3}, t_{e,3}, \}$. The construction of the output function λ^* is simple, the result appearing in Table 6.4. The generation of δ^* is a bit more complex. The flow table next state positions are generated one column at a time based on a compatible of $dom(\kappa)$ at a time. The implied compatibles are then used to conduct the filling of the flow table entries. The result of the application of step 3 appears also in Table 6.4.

Table 6.4: Flow table for the partition augmentation FSM in example 6.2

state \ input	0	1
$t_{a,1}$	$t_{a,1}$,0	$t_{b,2}$,0
$t_{b,1}$	$t_{b,1}$,0	$t_{c,2}$,-
$t_{b,2}$	$t_{b,1}$,0	$t_{c,3}$,-
$t_{c,2}$	$t_{b,1}$,-	$t_{c,3}$,1
$t_{d,2}$	$t_{a,1}$,0	$t_{e,3}$,1
$t_{c,3}$	$t_{b,1}$,-	$t_{c,2}$,1
$t_{e,3}$	$t_{a,1}$,1	$t_{d,2}$,1

It is easy to inspect this Table to see that the sets of states $\{t_{s,1} \,|\, s \in k_1\}$, $\{t_{s,2} \,|\, s \in k_2\}$ and $\{t_{s,3} \,|\, s \in k_3\}$ are compatibles, and that they form a closed cover of compatibles.

∎

Theorem 6.1 (Partition–κ) *Let $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$ be an* FSM *and $\mathcal{A}^* = \langle I, T, O, \delta^*, \lambda^* \rangle$ be another* FSM*, resulting from the application of the partition–κ state splitting method to $\mathcal{A}$, given a closed cover of compatibles κ of S. Then, $\mathcal{A}^*$ is an augmentation of $\mathcal{A}$.*

Proof. To be an augmentation of $\mathcal{A}$, the FSM $\mathcal{A}^*$ has to fulfill two conditions:

1. $|T| \geq |S|$;
2. $\mathcal{A}^*$ is simulated by $\mathcal{A}$.

According to step 1 of the partition–κ method, the cardinality of T is the sum, over all states in S, of the number of times a state appears in some element of κ. Since κ is a cover of S, each state belongs to at least one compatible in κ. Then, the first condition is fulfilled.

To show that $\mathcal{A}$ simulates $\mathcal{A}^*$, note that each state $s \in S$ is split into a set of states $\{t_{s,l}\}$ by the partition–κ method. If this set is a compatible of $\mathcal{A}^*$, and if the set of compatibles represented by all such sets is the domain of a closed cover of $\mathcal{A}^*$, the obtainment of the FSM $\mathcal{A}$ from $\mathcal{A}^*$ is evident by state minimization, showing that $\mathcal{A}$ simulates $\mathcal{A}^*$.

Consider first the output function λ^*. Given a state $s \in S$, all output values $\lambda^*(i, t_{s,l})$ are identical, for each $i \in I$ and every l, by step 2 of the method. Thus, no length 1 input sequence can distinguish among states $t_{s,l}$. Assume that a length 2 input sequence is applied to $\mathcal{A}^*$, and consider the next state function δ^*. Given s and i, every next state of a state $t_{s,l}$, under the input letter i, is in another set of the same form, $\{t_{m,l}\}$, for some m. This occurs because in step 3 of the method, δ^* is computed by the expression $\delta^*(i, t_{s,l}) = t_{\delta(i,s),b}$. Thus, no distinct output may occur with a length 2 input sequence. By induction on the sequence length, we prove that, given a state s, every set of states of the form $\{t_{s,l} \mid t_{s,l} \in T\}$ is a compatible. The set of all such compatibles is the domain of a cover of T. This set is closed, due to the fact that the set of next states of a set with the form $\{t_{s,l}\}$, i.e. any implied compatible of $\{t_{s,l}\}$, is a set of the same form.

∎

Corollary 6.1 (Partition–κ augmentation) *Every* FSM *has at least one augmentation that is a partition augmentation. To each closed cover of compatibles of a machine corresponds a partition augmentation.*

Corollary 6.2 (Partition–κ closed cover) *Every partition augmentation $\mathcal{A}^*$ of an* FSM *$\mathcal{A}$ has a closed cover κ^* such that $|\kappa^*| = |\kappa|$, where κ is the closed cover of $\mathcal{A}$ used to construct $\mathcal{A}^*$, and such that κ^* is a partition of the state set of $\mathcal{A}^*$.*

The importance of Theorem 6.1 and its corollaries is that, given a closed cover of compatibles, they show that the partition-κ method allows to define an FSM that simulates the original one, that has a closed cover of compatibles with same cardinality as the original closed cover, and where all classes are disjoint. We may then draw a parallel between assignments to the original machine and to the one obtained after applying the partition-κ method, which simulates the original one. This is the objective of the next Section.

6.3 Justifying the Use of Extended Assignments

Example 6.1 in Section 6.1 showed that the use of functional injective assignments may prevent minimum solutions of the SM/SA problem. The same Example suggested the use of less restrictive assignments to deal with the problem. Intuitively, the functional characteristic of an assignment is desirable to reduce the code length in SA constrained encoding problems, but it prevents the participation of states in more than one compatibility class in an encoded version of the original FSM. The injective characteristic avoids even the existence of compatibility classes in the solution. We demonstrate in this Section that less restrained encodings may allow the overcoming of these limitations, and yet generate valid state encodings.

Theorem 6.2 (Disjoint classes closed cover encoding) *Let $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$ be an* FSM *and κ be a closed cover of compatibles of $\mathcal{A}$ such that its classes are disjoint. Construct an encoding e of S as follows: for each state s_k on a class k of κ, associate $e(s_k) = \mathbf{x}$ with $\mathbf{x}$ being a Boolean vector of the form $\mathbf{x} = x_{n-1} \ldots x_1 x_0$ and $n \geq \lceil \log_2 |\kappa| \rceil$. Also, make the encoding such that*

$$\forall k, l \in \kappa, \;\; k \neq l \Rightarrow e(s_k) \neq e(s_l).$$

Then, e is a valid state encoding of $\mathcal{A}$.

Proof. The encoding e is a functional encoding, but not necessarily injective. To see this, it suffices to verify that κ is supposed to be a partition of S, and that e will be injective only in the special case where the classes of κ are singletons.

Since κ is a closed cover of compatibles, there is a machine $\mathcal{A}'$ that simulates $\mathcal{A}$, and which, by state reduction, has a state corresponding to each compatible $k \in \kappa$. Thus, e is associated with a functional injective state assignment e' of $\mathcal{A}'$. Since by Theorem 5.1 any functional injective assignment is a valid encoding of an FSM state set, e is a valid state encoding of $\mathcal{A}'$. Since $\mathcal{A}'$ simulates $\mathcal{A}$, e is a valid state encoding of $\mathcal{A}$.

∎

This Theorem ensures that functional non-injective encodings can be used to take advantage of the existence of compatibility classes in a machine, provided that these classes do not overlap. It can be applied to any closed cover of compatibles formed by disjoint compatibles, and particularly to any cover that is additionally a minimum cover, if any exists. Thus, assignments like e in Theorem 6.2 can simultaneously achieve state minimization. The next Theorem generalizes this statement.

Theorem 6.3 (Closed cover encoding) *Let $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$ be an* FSM *and κ be a closed cover of compatibles of $\mathcal{A}$. Let us construct an encoding e of S as follows:*

1. *build a functional injective encoding $\epsilon : \kappa \longrightarrow \mathcal{B}^n$, with $n \geq \lceil \log_2 |\kappa| \rceil$.*
2. *build the encoding*

$$e : S \longrightarrow \mathcal{P}(\mathcal{B}^n), \quad \text{such that } \forall s \in S, \; e(s) = \{\epsilon(k) \,|\, k \in \kappa, s \in k\}.$$

Then, e is a valid state encoding of $\mathcal{A}$.

Proof. Encoding e need neither be functional nor injective. If some state s is in two distinct classes of κ, e fails to be functional. If any compatible is not a singleton, e fails to be injective.

Consider the partition augmentation $\mathcal{A}^* = \langle I, T, O, \delta^*, \lambda^* \rangle$ of $\mathcal{A}$ based on κ, as well as an encoding of T as follows:

$$e^* : T \longrightarrow \mathcal{B}^n,$$

such that, for every $s \in S$ and every $k_l \in \kappa$

$$e^*(t_{s,l}) = \epsilon(k_l).$$

This encoding is clearly one of the assignments that can be obtained by the state splitting method, by distributing the multiple Boolean vectors associated by e with a state that is an element of a set of compatibles in a one-to-one fashion to the states in T. In particular, this encoding preserves the compatibility classes structure captured by the encoding e^*.

Let κ^* be the closed cover of compatibles of $\mathcal{A}^*$ corresponding to κ. From the definition of e and e^*, the elements of a compatible $k^* \in \kappa^*$ are all assigned the same single code, and every compatible k^* of κ^* receives a distinct code. By Corollary 6.2, κ^* is a partition. Then, we may apply the results of Theorem 6.2 to $\mathcal{A}^*$, and conclude that e^* is a valid state encoding of $\mathcal{A}^*$. Since $\mathcal{A}^*$ is an augmentation of $\mathcal{A}$ (by Theorem 6.1), and since the encoding e generates e^*, e is a valid state encoding of $\mathcal{A}$.

∎

Let us illustrate the method proposed by the Theorem 6.3 through an example.

Example 6.3 (Non-functional, non-injective assignment) We refer here to the 5-state FSM described in Table 6.1 during the presentation of Example 6.1 in page 85. There we extracted the following 4-element closed cover of compatibles from the machine description:

$$\kappa = \{\{s, t\}, \{t, u\}, \{v\}, \{x\}\}.$$

Table 6.5: A functional, injective assignment for closed cover κ of example 6.3

state	code
{s,t}	00
{t,u}	01
{v}	10
{x}	11

Let us build an encoding for κ, $\epsilon : S \longrightarrow \mathcal{B}^2$ according to step 1 of the theorem. One possibility is presented in Table 6.5.

Note that this assignment is functional, injective, and has minimum length. After that, we build the final assignment for S, the state set of the machine, according to the step 2 of the Theorem. The result appears in Table 6.6.

Table 6.6: A non-functional, non-injective assignment for the state set S of example 6.3

state	code
s	00
t	0-
u	01
v	10
x	11

This is a valid assignment of S according to Theorem 6.3, and the encoding is neither functional nor injective.

Thus, from Theorem 6.3 we conclude that the use of non-functional, non-injective state assignments allows the consideration of state minimization in its full extent.

6.4 Violation of SM Constraints by Input Constraints

In this Section and the next one, we will investigate the relationship between the SM constraints and the input constraints. From the study of the relationship between these two kinds of constraints we may draw important information concerning their feasibility, i.e. the possibility or not of finding valid solutions of a constrained encoding problem by considering a given set of constraints. The compatibility among constraints may be analyzed according to various points of view. We are interested here in verifying the conditions under which some constraints do not admit that other constraints be considered as well. Determining this relationship is fundamental to generate a feasible set of constraints prior to looking for a solution of the constrained encoding problem.

Before presenting the theorems, let us remember from Chapter 5 that a SA full input constraint $(\{s_i\}, \{s_k\})$ is satisfied iff after encoding, the set of state codes for $\{s_i\}$ is contained in the satisfying set C_i of a cube and C_i does not intersect any code of some state in $\{s_k\}$. This fact is used in the proof of the theorems.

Theorem 6.4 (Incompatibility constraints non-violation) *Given a finite state machine $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$, if two states $s, t \in S$ are incompatible, any valid state encoding that respects the whole set of full input constraints generated by symbolic minimization of a cube table describing $\mathcal{A}$ assigns disjoint codes to s and t.*

Proof. The result of the symbolic minimization process is a cube table that also represents $\mathcal{A}$. Thus, we know that every specified entry of a flow table describing $\mathcal{A}$ will be considered in some row of the minimized cube table. To guarantee disjoint codes to s and t by respecting a set of full input constraints, it suffices that one of these constraints contains only one of s, t. Since $(s, t) \in \iota$, the incompatibility relation of $\mathcal{A}$, we know from the state compatibility definition that

$$\begin{array}{rl} \exists i \in I \text{ such that either} & \delta(i,s), \delta(i,t) \text{ are specified and} \\ & \delta(i,s) \neq \delta(i,t) \\ \textit{or} & \lambda(i,s), \lambda(i,t) \text{ are specified and} \\ & \lambda(i,s) \neq \lambda(i,t). \end{array}$$

In either case, there is an impossible symbolic grouping of s and t in the $i-th$ column of the flow table. Thus, among the full input constraints that consider this column, there is at least one where s and t do not appear simultaneously.

■

The intuitive interpretation of the above proof is quite easy. The proof of the Theorem arises from the observation that there is only two flow table column configurations that make s and t incompatible: either s and t assert distinct output values in a given column, or s and t assert distinct, incompatible states in some column. Knowing that symbolic minimization groups states only if they assert the same next state and/or output value, the proof is immediate.

The importance of theorem 6.4 is clear. It shows that symbolic minimization results cannot violate either full or elementary incompatibility constraints derived from the structure of an FSM. Hence, any encoding method that respects the whole set of input constraints derived from symbolic minimization is guaranteed to respect all constraints in ι. Yet, some SA constrained methods based on input constraints satisfaction explicitly include additional constraints to avoid that distinct states be assigned identical codes [103, 26]. We call these *injectivity constraints*[2], since they ensure that the state assignment ξ is one-to-one or injective. These constraints have the same form as face embedding constraints, i.e. $(\{s_i\}, s_k)$, but the first element of the pair representing it is always a singleton.

A valid subset of covering constraints is always respected by any SA method generating valid state encodings for FSMs, since every state is assigned a code. Thus, covering constraints may be disregarded during the use of a simultaneous strategy for solving the SM and SA problems.

Now, suppose that there is a set of face embedding constraints generated by symbolic minimization, and leading to intersecting codes for two distinct states. After the presentation of Theorem 6.4 and the observation about covering constraints in the last paragraph, the only SM constraints that can be violated by this set as a whole are the compatibility constraints and the closure constraints. Hence, we may already precise that if we consider machines without

[2] In [103], the authors call it *uniqueness constraints*, while in [26] the concept is modeled using the so-called *fundamental dichotomies*.

closure constraints, the addition of injectivity constraints can have only one consequence, which is to avoid valid state minimization to occur automatically during the encoding process. Then, we must look for a better technique to consider the injectivity constraints in our case, if this is needed at all.

Any reasoning concerning the injectivity constraints must involve the analysis of the relationship between input and closure constraints, which we consider in the next theorem.

Theorem 6.5 (Closure constraints violation) *Given an* FSM $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$, *if two states* $s, t \in S$ *are conditionally compatible, any encoding that respects the whole set of full input constraints generated by symbolic minimization of a cube table of* $\mathcal{A}$, *assigns disjoint codes to* s *and* t.

Proof. To guarantee disjoint codes to s and t by respecting a set of input constraints, it suffices that one of the constraints contains only one of these states. Since (s, t) is a conditionally compatible pair, $\exists u, v \in S, u \neq v$ such that $(\{s, t\}, \{u, v\}) \in \sigma$, with σ being the closure relation of $\mathcal{A}$. In this case, symbolic minimization guarantees that

$$\exists i \in I \text{ such that either } \quad \delta(i, s) = u, \delta(i, t) = v$$
$$\textit{or} \quad \delta(i, s) = v, \delta(i, t) = u.$$

Thus, no symbolic grouping of s and t is possible in the *i–th* column of the flow table. Obviously, among the face embedding constraints associated with this column there must be at least one constraint where s and t do not appear simultaneously.

■

Theorem 6.5 shows that the input constraints resulting from symbolic minimization cannot violate any elementary closure constraint derived from the structure of an FSM either. The extension of theorems 6.4 and 6.5 to non-elementary forms of incompatibility and closure constraints is immediate and shall not be discussed here.

The conclusion we arrive at by the combination of the results of theorems 6.4 and 6.5 is that the injectivity constraints are useless, as long as attention is restrained to methods that respect *all* input constraints generated by disjoint minimization. We may even state that their use can only worsen the quality of a state assignment, since they prevent valid state minimization to occur. Another conclusion we may draw from these theorems is that we may get some state minimization out of an encoding based only on input constraints satisfaction. In this case, however, the merging derives from a particular type of relation, which is the total compatibility between states. In this sense, the minimization we obtain is in fact a weaker form than that obtainable by considering the full set of SM constraints.

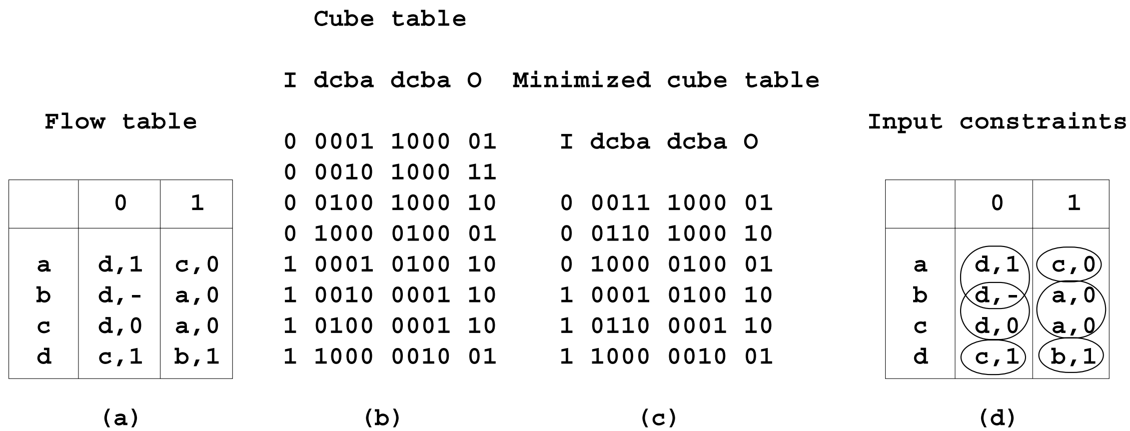
6.5 Violation of Input Constraints by SM Constraints

We demonstrated in the last Section that a set of input constraints produced by disjoint minimization violates neither incompatibility nor closure constraints. A question we want to answer now is whether a similar condition holds in the opposite direction. In other words, we want to verify if the SM constraints may violate some input constraint, thus destroying the certitude of keeping the bounds derived from the latter. We will see that this is indeed the case. After

showing how this is possible, we delimit the conditions under which these violations may occur, or conversely, describe how to avoid that the violations jeopardize the quality of state encoding techniques.

We show that SM constraints may lead to violation of an initial set of input constraints through the presentation of a small, yet significant example.

Example 6.4 (Input constraints violation) Let $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$ be a finite state machine where $I = O = \mathcal{B}$, $S = \{a, b, c, d\}$, and where δ and λ are given by the flow table in Figure 6.2(a).



Flow table

	0	1
a	d,1	c,0
b	d,-	a,0
c	d,0	a,0
d	c,1	b,1

(a)

Cube table

I	dcba	dcba	O
0	0001	1000	01
0	0010	1000	11
0	0100	1000	10
0	1000	0100	01
1	0001	0100	10
1	0010	0001	10
1	0100	0001	10
1	1000	0010	01

(b)

Minimized cube table

I	dcba	dcba	O
0	0011	1000	01
0	0110	1000	10
0	1000	0100	01
1	0001	0100	10
1	0110	0001	10
1	1000	0010	01

(c)

Input constraints

	0	1
a	d,1	c,0
b	d,-	a,0
c	d,0	a,0
d	c,1	b,1

(d)

Figure 6.2: Flow table, cube table, minimized cube table and full input constraints for FSM $\mathcal{A}$

After transforming the flow table into a mixed binary and positional cube table (Figure 6.2(b)), we submitted the machine description to the program ESPRESSO-MV [99] for symbolic minimization. The enumeration order used for the positional cube scheme encoding of the state set in the second and third groups of columns in Figure 6.2(b) is the alphabetic order, from right to left in each column. The result of the symbolic minimization appears in Figure 6.2(c), where the second column contains a representation of the face embedding constraints. Each 1 in a Boolean vector in this column implies that the associated state is in the first coordinate set of a pair $(\{s_i\}, \{s_k\})$ representing the full input constraint.

The full input constraints

$$\{(\{a,b\},\{c,d\}), (\{b,c\},\{a,d\}), (\{d\},\{a,b,c\}), (\{a\},\{b,c,d\})\},$$

are depicted again in Figure 6.2(d) as groups of flow table entries. Turning the attention to the SM problem, we observe that there is only a single pair of compatible states in machine $\mathcal{A}$, namely b and c. A closed cover of compatibles is immediately extracted from the machine, namely $\{C_1 = \{a\}, C_2 = \{b, c\}, C_3 = \{d\}\}$. We may now build a minimization of $\mathcal{A}$ based on this closed cover. This machine is $\mathcal{A}' = \langle I, S', O, \delta', \lambda' \rangle$, where $S' = \{a, bc, d\}$, and δ' and λ' are given by the flow table in Figure 6.3(a).

The symbolic minimization procedure applied to machine $\mathcal{A}$ is repeated for $\mathcal{A}'$, to generate its face embedding constraints. Figure 6.3 illustrates the process. Machines $\mathcal{A}$ and $\mathcal{A}'$ have generated six and five full input constraints, respectively. We see that there is a direct relationship between the two sets of constraints, conditioned only by the merging of states.

(a) Flow table

	0	1
a	d,1	bc,0
bc	d,0	a,0
d	bc,1	bc,1

(b) Cube table

I	dbca	dbca	O
0	00 1	10 0	01
0	01 0	10 0	10
0	10 0	01 0	01
1	00 1	01 0	10
1	01 0	00 1	10
1	10 0	01 0	01

(c) Minimized cube table

I	dbca	dbca	
0	00 1	10 0	01
0	01 0	10 0	10
-	10 0	01 0	01
1	00 1	01 0	10
1	01 0	00 1	10

(d) Input constraints

	0	1
a	d,1	bc,0
bc	d,0	a,0
d	bc,1	bc,1

Figure 6.3: Flow table, cube table, minimized cube table and full input constraints for FSM $\mathcal{A}'$

Intuitively, we note that all but two constraints have kept the same 'amount of information' about the original machine, changing only the form of presenting this information. The two constraints whose information contents changed illustrate the two phenomena that state merging may cause, namely constraint splitting and constraint merging. As an example of constraint splitting, let us observe the face embedding constraint in the first line of Figure 6.2(c). The merging of states b and c has eliminated the output don't care in one of the flow table entries, preventing one of the original constraints to continue to exist. On the other hand, the same state merging may allow constraints to merge, as is the case with the third and sixth lines of Figure 6.2(c), which become the third line of Figure 6.3(c).

■

Both constraint merging and constraint splitting cause changes in the number and form of the input constraints. Constraint merging constitutes a desirable condition, since it reduces the bounds on the final size of the implementation. Constraint splitting, however, may cause problems if their presence forces an increase in the cardinality of the initial set of face embedding constraints. Once state merging necessarily reduces the size of the flow table, we could reason that an increase in the number of input constraints after state minimization is an unlikely situation. Nevertheless, the next example shows that this reasoning does not account for the general situation.

Example 6.5 (Increase in input constraints) Let $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$ be an FSM where $I = \mathcal{B}^2$, $S = \{a, b, c\}$, $O = \mathcal{B}$, and where δ and λ are given by the flow table in Figure 6.4(a).

We apply the same steps from example 6.4, transforming the flow table in a mixed binary positional cube table (Figure 6.4(b)), submitting the machine description to ESPRESSO–MV for disjoint minimization (Figure 6.4(c)), and depicting the face constraints in Figure 6.4(d).

Again, there is only a single pair of compatible states in machine $\mathcal{A}$, namely (a, b). A closed cover of compatibles is immediately extracted from the machine, namely $\{C_1 = \{a, b\}, C_2 = \{c\}\}$. Based on this closed cover, we build a minimization of $\mathcal{A}$. The new FSM is $\mathcal{A}' = \langle I, S', O, \delta', \lambda' \rangle$ where $S' = \{ab, c\}$, and δ' and λ' are given by the flow table in Figure 6.5(a).

Machine $\mathcal{A}'$ undergoes the same symbolic minimization procedure as machine $\mathcal{A}$, in order to generate its face embedding constraints. Figure 6.5 illustrates the process. Machines $\mathcal{A}$ and $\mathcal{A}'$ have thus generated five and six input constraints, respectively, showing that state reduction

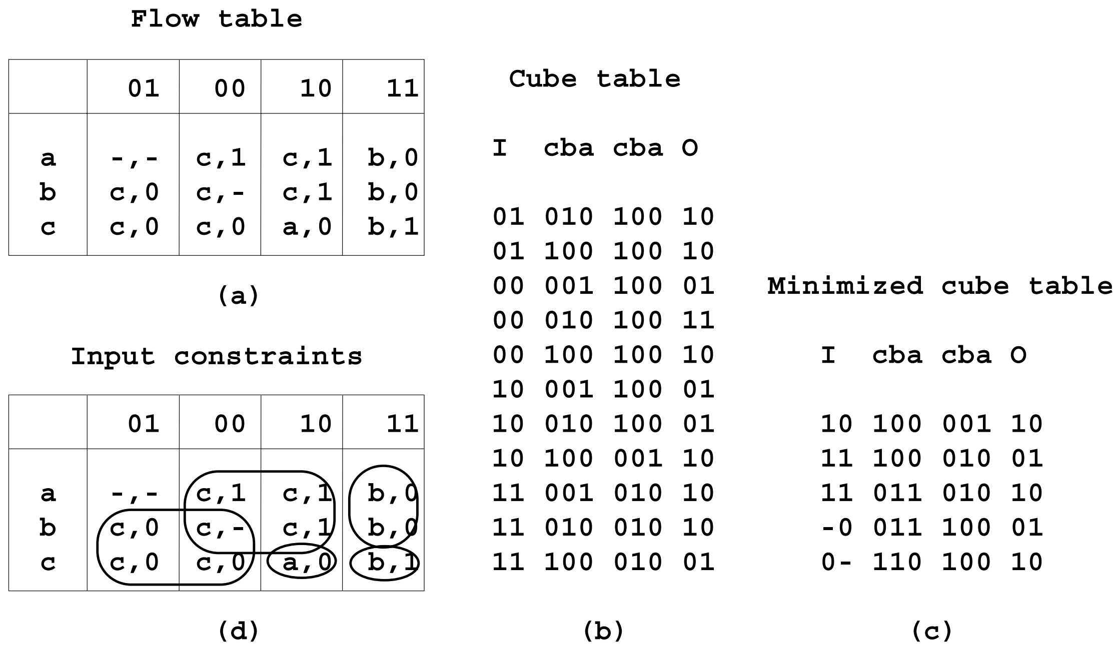


Figure 6.4: Flow table, cube table, minimized cube table and full input constraints for FSM $\mathcal{A}$

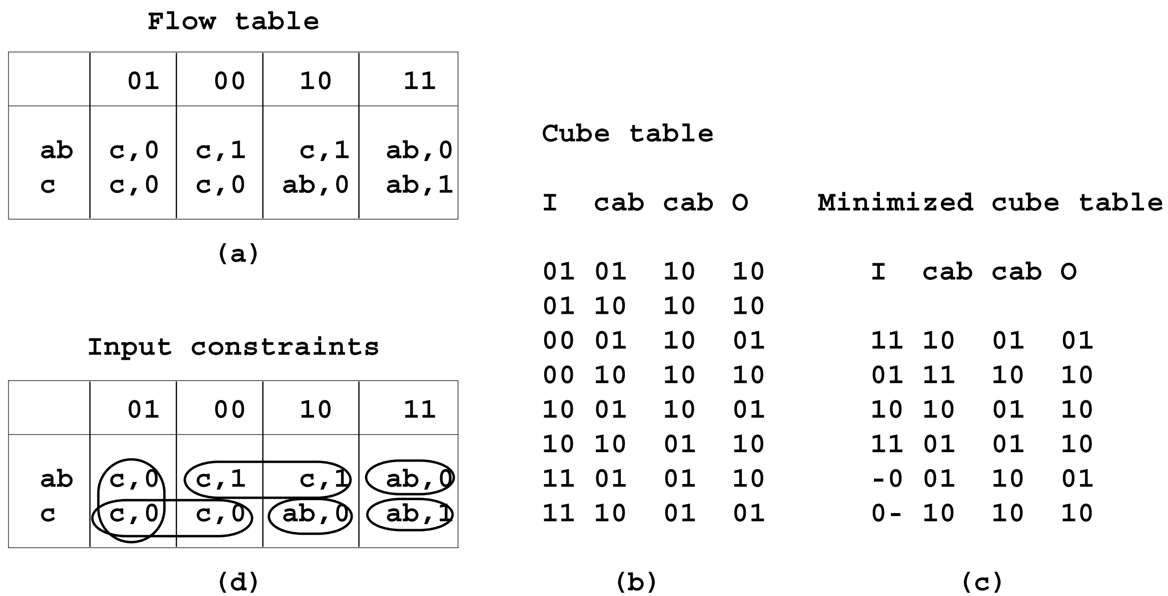


Figure 6.5: Flow table, cube table, minimized cube table and full input constraints for FSM $\mathcal{A}'$

can indeed increase the number of input constraints.

∎

The result obtained in Example 6.5 is very important. It tells that performing state minimization *before* state assignment may increase the number of input constraints derived from a minimum cube table obtained with symbolic minimization. The method we propose in Chapter 12 allows considering of the SM constraints without incurring in such violations.

6.6 Conflicts within and with Output Constraints

Symbolic minimization techniques generate a feasible set of full input constraints for an FSM represented as a cube table. The SM constraints can all be obtained at once from the compatibility table method of Paull and Unger [91]. In this Section, we depict some relationships between the output constraints and the other SM and SA constraints. De Micheli proposed a method that automatically builds up a feasible set of input and dominance constraints in [40] (see Section 5.4.2, for more details), but the method, although efficient, does not provide good solutions. The alternatives for output constraints generation depicted in Chapter 5 produce constraints that may conflict with the input and SM constraints. Indeed, all these methods generate output constraints separately from the other constraints. Since none of the available output constraints generation methods are adequate for our purposes, we treat the conflicts related to output constraints differently from the conflicts between input constraints and SM constraints. While the results of the previous Section regarded how a feasible set of full input constraints interact with individual SM constraints, the discussion here focuses on the conflicts between individual output constraints and individual input and SM constraints.

Another distinction between output constraints and the others is that the former ones are sensitive to the particular values of the encoding, and not only to the fact that these values are distinct or not. Any encoding Ξ of a set S respecting a feasible set of constraints F, and containing only input or SM constraints, can be exchanged by an encoding generating exactly the complement of each code generated by Ξ, for each element of S. This new encoding automatically respects all constraints in F, which is not necessarily true if output constraints are part of F.

6.7 Conflicts among Output Constraints

It should be clear that output constraints may conflict among themselves. This never happens with input constraints, for instance, where there is always an encoding satisfying a set of input constraints, for a code length that is large enough.

Example 6.6 (Dominance constraints) Given an FSM $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$, an encoding Ξ of S, and two states $s, t \in S$, suppose that $(s,t), (t,s) \in \mu$, i.e. the dominance relation on S under Ξ. Then, if s and t are incompatible states, we see that the dominance relations cannot both be satisfied by any valid encoding of S, since they imply that s and t must receive identical codes under Ξ. If the states are compatible, on the other hand, this must be made explicit by adding to the same set of constraints a pair (s,t) from θ, i.e. the compatibility relation of $\mathcal{A}$.

Globally, this case implies also that any closed cover of compatibles considered in the encoding must have states s and t either together, or both absent in every compatible. Otherwise they risk receiving distinct codes, which would violate at least one of the dominance constraints. Also, in order to satisfy the two dominance constraints at once, no input constraint separating s and t may be satisfied. In general, admitting $(s,t),(t,s) \in \mu$ is not a good choice, because of all the additional restrictions it implies.

Example 6.7 (Disjunctive constraints) Given an FSM $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$, an encoding Ξ of S and three states $s,t,u \in S$, suppose that $(s,\{t,u\}),(t,\{s,u\}) \in \chi$, i.e. the disjunctive relation on S under Ξ. Then, if s and t are incompatible states, we see that the disjunctive relations cannot both be satisfied by any valid encoding of S, since they imply that s and t must receive identical codes under Ξ. All other observations made in the previous example are valid *mutatis mutandis* for disjunctive constraints. Again, admitting $(s,\{t,u\}),(t,\{s,u\}) \in \chi$ is not generally a good choice, because of all the additional restrictions it implies.

The distinct output constraints may also conflict among them.

Example 6.8 (Dominance $\times$ disjunctive) Given an FSM $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$, an encoding Ξ of S, and three states $s,t,u \in S$, suppose that $(t,s) \in \mu$ and $(s,\{t,u\}) \in \chi$. This implies that s and t must receive identical codes under Ξ, just as in the Example 6.7. This situation should accordingly be avoided. On the other hand, suppose that $(s,t) \in \mu$ and $(s,\{t,u\}) \in \chi$. The dominance constraint is then redundant, since respecting the disjunctive constraint implies respecting it automatically.

Other less obvious incompatibilities among constraints may occur. Suppose a full input constraint $(\{s_i\},\{s_k\}) \in \phi$, an input relation with $s,t \in \{s_i\}$, $u \notin \{s_i\}$, and a disjunctive constraint $(u,\{s,t\}) \in \chi$. These two constraints can never be satisfied simultaneously, since any code assigned to u satisfying the disjunctive relation would cause the face generated by $\{s_i\}$ not to evaluate to 0 for the code of u, which violates the input constraint.

Given that not all constraints are obtained by a single unified method, the production of a feasible set of constraints is a fundamental step prior to the constraints satisfaction phase of the solution of the SM and SA problems using a simultaneous strategy. Additionally, a special care has to be taken while considering the output constraints, due to the conflicts they may insert in the constraint set.

Chapter 7

Conclusions on Constraint Nature and Generation

The objective of Part II has been to introduce and discuss the constraints describing thoroughly both the SM and SA problems, as well as the techniques available to generate these constraints. None of the techniques showed here is new. What is original in the present work is the constraint interpretation proposed, which consists in decomposing constraints into its constituent parts and modeling the sets of elementary constraints thus obtained as binary relations. Also, the contents of Chapter 6 are fundamentally new, and provide the theoretical basis for the rest of this thesis. At least one work has already suggested the use of non-functional, non-injective encodings to account for state minimization and state assignment issues at once [6], but no theoretical justification was available for this choice in the referred work. Finally, to the author knowledge, the partition-κ method of state splitting into disjoint compatible classes is an original contribution as well.

From the presentation of constraint generation techniques in Chapters 4 and 5, we may conclude that:

1. the elementary SM constraints generation relies on polynomial time/space complexity algorithms, and it is thus considered an easy and cost-effective task;
2. the complex part of the SM problem, on the other hand, is the generation of a closed cover of compatibles from the elementary constraints, where a non-polynomial time complexity covering-closure problem has to be solved;
3. the SA constraints generation is intrinsically a hard task, whose time complexity is non-polynomial in the worst case;
4. in practice, the SA input constraints can be efficiently generated by multiple-valued minimizers that, allied to the upper bounds they provide on the two-level cube table row cardinality of a solution, gives an adequate starting point for the research of optimal encodings;
5. no acceptable input/output constraints generation technique is available, and even the ones available do not offer a degree of efficiency adequate to our requirements. The search for such efficient techniques is considered out of the scope of this work, but the

developments we propose in the next Parts take output constraints into account, in hope that efficient methods for their generation will be available in the future.

Theorem 6.3 showed that assigning codes that are sets of Boolean vectors to states in an FSM allows the simultaneous consideration of SM and SA constraints in the scope of a state encoding method. However, such an encoding is certainly incomplete from the point of view of a hardware implementation, where only binary information is allowed to be treated. Thus, such an encoding cannot be but an intermediate step before layout generation, where a complete encoding is required. The solution of the problem posed by such incomplete encodings relies on the combinational logic minimization step subsequent to encoding. We know [89] that incompletely specified Boolean codes are treated since long by two-level, as well as multiple-level minimizers, and that these tools are quite efficient nowadays. In fact, using codes that are sets of Boolean vectors, and not only single Boolean vectors, to encode states increases the degrees of freedom during logic minimization. Minimization tools can, in this case, pick the Boolean vectors more adequate to solve the combinational problem. This is advantageous, since the cost functions employed by these lower abstraction level tools model more realistically the final implementation.

Example 6.5 demonstrated an important result, which is that executing state minimization before state assignment may provoke an increase in the upper bound predicted by symbolic minimization. We demonstrate, in Chapter 9, that it is possible to respect the original bound and still consider state minimization. Later, in Chapter 12, we propose a method that respects the demonstrated result.

In the specific case of this work, the encodings we are going to generate will consider an optimal two-level implementation of an FSM. We have already discussed efficient two-level representations of discrete functions in Chapter 3, all of them derived from the use of cube functions. Accordingly, we will limit attention from now on to cube encodings. Cube encodings, as stated in Definition 5.1, use codes that are satisfying sets of some switching cube function. In this way, we limit attention to codes that can be represented by cubes. This choice grants us with the possibility of relying on very effective combinational two-level minimization programs to perform the final assignments from the cube encoding we generate. Examples of such programs are MCBOOLE [30] and ESPRESSO[16].

Part III

A Unified Framework for SM and SA Constraints

Chapter 8

Pseudo-Dichotomies

The mathematical concept of *partition* has been associated to the synthesis of sequential machines for a long time. In Chapter 1, we have mentioned several works using partitions to model sequential machines concepts. The *structure theory* of Hartmanis and Stearns, for instance, relies mostly on the manipulation of partitions to encode, decompose and minimize the number of states in sequential machines [63]. Tracey redefined the partition concept in [111], to adapt it to his need of an algebraic structure to model the state assignment problem for asynchronous machines. In order to avoid the conflict with the well-established mathematical definition, Unger rebaptized the structure suggested by Tracey with the name *partial state dichotomy* [113]. The abbreviated form *dichotomy* is the term most generally accepted today to describe this concept. In another work, Ciesielski and Davio suggested a terminology, which is more consistent with the meaning of the word 'dichotomy' in the English language, and which clearly expresses the relationship between dichotomies and partitions [25]. They call Unger's partial state dichotomies *pseudo-dichotomies*, and in the special case where a pseudo-dichotomy is equivalent to a partition, it is called a *dichotomy*. This is the terminology we assume herein.

Several publications reported the use of dichotomies to model the state assignment problem in both asynchronous [113, 60, 106] and synchronous [119, 26, 103, 106] FSMs.

In this Chapter, we give a definition of the pseudo-dichotomy concept that is showed to be more comprehensive than that in previous approaches. This extended concept is applied in this work to the resolution of the SM and SA problems only. However, it was devised to allow the application of the notion to the solution of more general problems, such as the *Boolean constrained encoding problem*, to be stated in Part IV.

In the next Section, we discuss the extended pseudo-dichotomy definition. In Section 8.2, we provide a first insight on the relationship between pseudo-dichotomies and the assignment of binary codes to symbols, by means of an example.

8.1 The Pseudo-Dichotomy Definition

The definition of *pseudo-dichotomies* presented below has a close connection with the concept of partition, as defined in Section 3.1. A pseudo-dichotomy is a concept useful to model constraints in Boolean encoding problems. In general, these constraints consist on indications to separate or not to separate codes of a set of symbols. Two-block partitions are adequate to

model such a behavior. A two-block partition may be interpreted as an indication to make bits on one block distinct from the bits in the other block (thus separating the codes of symbols in distinct blocks). However, the partition definition implies that all elements of the symbol set be contained in some block. This can be relaxed by taking partitions of subsets of the symbol set. On the other hand, the rules to use such partition-like structures may also vary from a problem to another. An encoding constraint may tell that symbol codes have to be separated in at least one bit position of the encoding, like the input constraints; they may require that codes never be separated, like in the case of compatible states in an FSM, and so on.

We propose here the pseudo-dichotomy as a two-part structure: one two-block partition-like structure, to model the symbol separation characteristic of the constraint, and a general switching function to tell how to satisfy the requirements of specific constraints with a given symbol separation characteristic. Follows the formal definition and a brief discussion of the reasons to propose such a general concept. Section 8.2 will present an example to illustrate one application of pseudo-dichotomies.

Definitions 8.1 (Pseudo-dichotomy) *Let $S = \{s_0, \ldots, s_{n-1}\}$ be a set, the elements of which are called **symbols**, and $\mathcal{B} = \{0, 1\}$. A **pseudo-dichotomy** (PD) of S is an algebraic structure $\partial = \langle p, t\rangle$ where p is the graph of a binary relation $\langle \mathcal{B}, S, p\rangle$, with*

$$p : \mathcal{B} \longrightarrow S \quad \text{such that } p(0) \cap p(1) = \emptyset,$$

and t is a switching function

$$t : \mathcal{B}^n \longrightarrow \mathcal{B}.$$

*Function t is called the **satisfaction function** of ∂. The sets $p(0)$ and $p(1)$ are called the **0–side** and the **1–side** of ∂, respectively.*

*A **dichotomy** of S is a PD $\partial = \langle p, t\rangle$ such that*

$$p(0) \cup p(1) = S.$$

*This binary relation with graph p is, in this case, a partition. A **seed pseudo-dichotomy** (SPD) is a PD $\partial = \langle p, t\rangle$ where either*

$$|\ p(0)\ | = 1 \quad or \quad |\ p(1)\ | = 1.$$

*Given a Boolean vector $\mathbf{x} = x_{n-1} \ldots x_0$, a PD ∂ is **satisfied** by $\mathbf{x}$ iff $t(x_{n-1}, \ldots, x_0) = 1$. A **flexible pseudo-dichotomy** is a PD where*

$$\forall x_{n-1} \ldots x_0 \in \mathcal{B}^n,\ t(x_{n-1}, \ldots, x_0) = t(\overline{x_{n-1}}, \ldots, \overline{x_0}).$$

*A **fixed pseudo-dichotomy** (FPD) is a PD $\partial = \langle p, t\rangle$ where the satisfaction function t is the cube function whose three-valued cube switching representation is as follows:*

1. *if $s_i \in p(0)$, position x_i of the cube representation is 0;*

2. if $s_i \in p(1)$, position x_i is 1;

3. otherwise, position x_i is -.

Two PDs *$\partial_1 = \langle p_1, t_1 \rangle$, $\partial_2 = \langle p_2, t_2 \rangle$ of S are* ***compatible*** *iff there is at least one Boolean vector $\mathbf{x} = x_{n-1} \ldots x_0$ such that $t_1(\mathbf{x}) = t_2(\mathbf{x}) = 1$. A set of* PDs *is* ***compatible*** *if every two* PDs *in it are compatible. The* PD *∂_1* ***covers*** *∂_2 iff*

$$\forall \mathbf{x} \in \mathcal{B}^n, \;\; t_2(\mathbf{x}) = 1 \Rightarrow t_1(\mathbf{x}) = 1.$$

A set of PDs *Δ* ***covers*** *a* PD *∂ iff it contains a* PD *that covers ∂.*

We represent PDs using the value vector notation presented in Chapter 3, which we employ to characterize binary relation graphs, instead of discrete functions only. Given a PD $\partial = \langle p, t \rangle$ on the set of symbols S, ∂ may be described by the value vector $[p(0)\ p(1)]$, which contains the images of the elements 0 and 1 by the binary relation whose graph is p.

Observe that in a dichotomy $\partial = \langle p, t \rangle$, p is the graph of a partition. On the other hand, the binary relation graph p of a PD $\partial = \langle p, t \rangle$ is not necessarily the graph of a partition of S, but instead the graph of a partition of a subset of S.

In most applications, useful PDs have non-empty 0–sides and 1–sides. However, the definition does not impose this condition. In at least one application found in the literature [106], the use of PDs with one of the sides empty can be advantageous. This application arises in a problem called *constrained via minimization* [1], an optimization problem found in multilayer routing of integrated circuits and printed circuit boards.

8.1.1 Generality of the Pseudo-Dichotomy Definition

Consider an assignment of a set of symbols with sets of Boolean vectors. The constraints this mapping must respect are relationships among the symbols, as pointed out in Part II. In this way, they may be expressed as conditions to be respected among the symbols in some subset of columns of the encoding. PDs were defined to model a single one of these columns.

In view of this elementary character of PDs, let us now examine the satisfaction function t of a generic PD $\partial = \langle p, t \rangle$ of a set S. Its interpretation is as follows. The domain of a satisfaction function t is the set of all binary assignments of length 1 to symbols in S. Function t evaluates to 1 for every binary assignment that satisfies the PD, otherwise it evaluates to 0. In this way, PDs with a same binary relation p can be satisfied in different ways, if their satisfaction functions t are distinct. This fact allows that different kinds of constraints be accounted for more easily.

We note that the satisfaction function is not present in previous definitions of the PD concept. Actually, several of the first published applications dealt with just one kind of constraint at a time [111, 119]. As the need to manipulate other constraint kinds arose, the proposal of *ad hoc* frameworks took place to deal with the "anomalous" behavior of the new constraints [60, 26, 103]. For example, Ciesielski et al [26] and Saldanha et al [103] independently proposed the fixed PD extension to allow the modeling of the SA output constraints.

Our proposal breaks a PD into two parts: the encoding part, expressed by a partition-like binary relation with graph p, and the satisfaction part, characterized by the switching function t. This approach has the merit of being general. For instance, it allows the use FPDs easily, since they are a more general case, while flexible PDs are a special case of the concept. More examples of the enhanced malleability of our PD definition will appear in Chapter 9, during the discussion of constraints representation with PDs.

The consideration of future constraint kinds using the above PD definition is straightforward. It suffices to define the conditions under which such a new constraint is satisfied, generating a new type of function t for all PDs of this kind.

Finally, we must point out that a distinction exists between PD satisfaction and constraint satisfaction. Constraints are the components into which we decompose an encoding problem, while PDs are the elementary components into which we decompose the constraints. Constraint satisfaction implies, in general, the simultaneous satisfaction of a set of PDs. In Chapter 9, this distinction will become clear as we discuss constraint representation and satisfaction using PDs. In Chapter 11, we will find a new general formulation of the encoding problem based on the use of constraints. There, the terms constraint and constraint satisfaction are formally defined, without connection with any specific encoding problem. Indeed, Chapter 11 will generalize the constraint-based formulations advanced for the SM and SA problems in Part II.

8.2 Pseudo-Dichotomies and Encoding

In this section we provide a first illustration of the close relationship existing between PDs and encoding, by means of an example.

Fixed dichotomies may represent minterms, and FPDs may represent cubes. As an example, the solution of the SA problem is assumed here to be a cube encoding, whose domain is a two-dimensional matrix, where rows correspond to state codes in the form of three-valued cube representations, and columns correspond to encoding columns. One distinction between our approach and all others constrained encoding methods we could find proposed in the literature [26, 103, 105, 111, 113, 119] is that the cube matrix we generate contains unspecified (don't care) entries. In particular, all such constrained encoding methods obtain the solution matrix by generating a set of fixed dichotomies, i.e. minterms. Our method generalizes the approach by creating a set of FPDs. Each FPD in our solution is mapped into a column of the solution encoding as follows. States whose symbols appear on the 0–side (resp. 1–side) are assigned a 0 (resp. 1) value in the column associated to this FPD. States that do not appear in any side of the FPD in question receive a don't care - in the corresponding column. A justification for this procedure, as well as a systematic method to convert cubes into PDs and vice versa, is provided in Chapter 12, during the presentation of the encoding method we propose.

Example 8.1 (Pseudo-dichotomies and encoding) To exemplify the mapping of final PDs into an encoding, we have used the prototype ASSTUCE program, developed in the scope of the present work, to generate a solution state assignment for machine LION9, one of the MCNC FSM benchmark machines [118]. Table 8.1 reproduces the flow table for this machine.

Table 8.1: Flow table for machine LION9

state \ input	00	01	10	11
0	0,0	-,-	1,0	-,-
1	0,0	-,-	1,0	2,0
2	-,-	3,0	1,0	2,0
3	4,1	3,1	-,-	2,1
4	4,1	3,1	5,1	-,-
5	4,1	-,-	5,1	6,1
6	-,-	7,1	5,1	6,1
7	8,1	7,1	-,-	6,1
8	8,1	7,1	-,-	-,-

The final FPDs generated by the program, represented by their value vectors, are:

[{0,1,2,5,6,7,8},{3}], [{0,1,2},{3,4,5,6,7,8}], [{1,2,3,4,5},{7,8}],
[{1,2,3,4},{6,7,8}], [{1,2,3},{5}].

These value vectors represent the partition-like binary relations p of the pseudo-dichotomy definition. Note indeed that in each of the value vectors above no symbol appears in both sides of the vector. As for the t switching functions, they are determined here as follows, for every pseudo-dichotomy. Consider each of the state symbols as Boolean variables. Then, the functions t are simply switching cube functions with as many literals as there are symbols in the corresponding value vector. Symbols on the 0-side are associated with complemented literals and symbols on the 1-side to non-complemented literals. For instance, for the first value vector we have $t = \overline{0}\,\overline{1}\,\overline{2}\,\overline{5}\,\overline{6}\,\overline{7}\,\overline{8}\,3$ (satisfied by two distinct Boolean vectors from $\mathcal{B}^9$), and for the last one, $t = \overline{1}\,\overline{2}\,\overline{3}\,5$ (satisfied by thirty-two distinct Boolean vectors from $\mathcal{B}^9$).

```
State encoding     PLA after minimization

state   code          inputs    outputs
  0     00---         0---1--   001100
  1     00001         01---0-   110000
  2     00001         11----0   010100
  3     11001         ---0---   000010
  4     -100-         -0-1---   010001
  5     010-0         -1-1---   000011
  6     01-1-         -1---1-   011100
  7     0111-
  8     0111-

         (a)                  (b)
```

Figure 8.1: State encoding and final PLA for machine LION9

The mapping describing the state assignment appears in Figure 8.1(a), under the form of a truth table. Each output column corresponds to one of the FPDs depicted above, and are associated with the satisfying set of some function t represented as a three-valued vector.

Figure 8.1(b), on the other hand, shows the personality matrix of the PLA that implements the combinational part of LION9. This PLA was obtained after encoding states in the original machine description, extracting the resulting combinational part and submitting it to logic minimization using the program ESPRESSO, tasks that can all be accomplished directly by our prototype implementation, as it will be discussed in Chapter 14.

The personality matrix is described in Figure 8.1(b) by a binary cube table a format defined in Section 3.3.1. This format is the output generated by ESPRESSO, except for the header lines, which have been omitted. In fact, this is the default output format, called the **fd** ESPRESSO format, as explained in Appendix A. Following the format conventions, the first bits to the left in the input part correspond to primary input values (in our Example, the first two bits), and the last to the present state value (the last five bits of the input part in Figure 8.1(b)). Accordingly, the first bits to the left in the output part stand for next state values, and the rightmost bits correspond to primary output values.

∎

Chapter 9

Pseudo-Dichotomies and Constraint Representation

Constraints describing both the SM and SA problems under the point of view of FSM encoding were studied in Part II, together with the relationship among them. An algebraic structure capable of modeling all these constraint kinds has been defined in Chapter 8, the pseudo-dichotomies. The current Chapter is dedicated to show how each of the relevant constraint kinds highlighted in Part II can be mapped into a set of PDs. The objective of this mapping is to reduce the separately formulated constraints to a unique representation that can be efficiently manipulated. This representation is what we call the ***unified pseudo-dichotomy framework***. The framework we propose occupies a central place in this thesis, since its development allowed the application of constrained encoding techniques to consider the two problems simultaneously.

In the rest of this Chapter, we consider given an FSM $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$, as well as a set of binary *encoding* relations defining all elementary constraints to be considered in a solution of the SM and SA problems. References to the encoding Ξ, as stated in Chapter 7, imply the use of cube encodings. Additionally, we assume that such cubes are noted using the three-valued representation (Definition 3.22).

In Section 9.1 we will map each elementary constraint kind belonging to some relevant relation into a set of PDs carrying the same encoding data. After relating elementary constraints to PDs, we propose the unified PD framework. This proposition comprises the consideration of how a feasible subset of elementary constraints is generated from the possibly conflicting binary relations describing the SM and SA problems. By defining the unified framework below, we will attain the integration of the SM and SA problems. As a matter of fact, the simultaneous solution of both problems can be achieved by solving the problem posed by the PD framework. This problem, which we call the *two-level* SM/SA *problem*, is stated formally in Chapter 11.

9.1 Representing Constraints with Pseudo-dichotomies

Consider all constraint kinds found during the search for a solution to the SM and SA problems. These classes of constraints are as follows:

1. SM constraints:

   (a) compatibility constraints and incompatibility constraints;
   (b) closure constraints;
   (c) covering constraints;

2. SA constraints:
   (a) input constraints (or face embedding constraints);
   (b) output constraints:
      i. dominance constraints;
      ii. disjunctive constraints;

The constraints above are sufficient to completely describe the SM and SA problems. We now discuss their representation under elementary forms, using the PD concept. Each elementary constraint kind is modeled by depicting the characteristics of the set of PDs needed to represent it, i.e. telling how to build the graph p of the binary relation and the satisfaction function t of each PD $\partial = \langle p, t \rangle$ in the set.

9.1.1 Representing SM Constraints

The modeling of the SM constraints is another of the original contributions of this thesis, since to the authors knowledge, no application has ever devised their representation using PDs for synchronous implementations of FSMs, even though this mapping is straightforward, as we will see below.

First, compatibility and incompatibility constraints are complementary, and a choice can be made to use one or the other, since using both is redundant. We have chosen to use compatibility constraints, because in present machines the number of compatible states is typically much smaller than the number of incompatible ones [118], with few exceptions. More important than that, the compatibility constraints are needed anyway during the construction of the PD framework, as we will see in Section 9.2.

Given a pair $(a, b) \in \theta$, the state compatibility between a and b is modeled by one PD $\partial = \langle p, t \rangle$, such that the graph p is represented by the value vector $[\{a\}, \{b\}]$. As for the satisfaction function t, we know from the results of Chapter 6 that compatible states must be encoded with intersecting codes, i.e. $\Xi(a) \cap \Xi(b) \neq \emptyset$. This corresponds to preventing that in some encoding column we have either a 1 for the code of a and a 0 for the code of b, or vice versa. For example, $\Xi(a) = 010$, $\Xi(b) = 0-0$ is part of an encoding that satisfies the compatibility constraint above, while $\Xi(a) = 011$, $\Xi(b) = 0-0$ is not. Note that the function t must ensure that the pseudo-dichotomy in this case is a flexible SPD.

Solving the SA problem implies assigning a code to each state. In this way, the covering constraints lose their relevance in a simultaneous strategy for solving the SM and SA problems. They are accordingly ignored herein.

Closure constraints are similar in nature to the compatibility constraints. Consider a pair $(\{a, b\}, \{c, d\}) \in \sigma$. This elementary closure constraint is modeled by two flexible SPDs $[\{a\}, \{b\}]$ and $[\{c\}, \{d\}]$ whose satisfaction functions are defined in the same way as for the elementary compatibility constraint.

If a closed cover of compatibles is available, and it is expressed by a compatibility relation and a closure relation, the closure relation is obviously redundant. Given a closed cover of compatibles, decomposing its compatibles into elementary compatibility constraints is a trivial task, and provides all necessary information about possible state merging situations. However, our approach is to avoid the costly step of finding an optimal closed cover of compatibles. To cope with this goal, the consideration of the closure constraints may be fundamental, as will be discussed in Section 9.2.

9.1.2 Representing SA Constraints

The first SA constraints derived from synchronous FSMs descriptions to be modeled with PDs were the input constraints [119]. Yang and Ciesielski showed in [119] that reinterpreting an input constraint as a set of flexible SPDs is a natural way of allowing constrained encoding techniques to be applied during the solution of the state assignment problem.

The input constraints generated by symbolic minimization are in the form of full input constraints [43]. Let $S = \{0,1,2,3,4,5,6,7,8\}$ be the state set of some finite state machine, and let the pair $(\{0,1,2\},\{3,4,5,6,7,8\})$ be one such full input constraint extracted from a symbolically minimized cube table of some FSM. To model this constraint with PDs, we may choose $\partial = \langle p,t\rangle$ such that $p = [\{3,4,5,6,7,8\},\{0,1,2\}]$ and t evaluating to 1 for column encodings separating the codes of every two states in opposite sides of the PD. Notice that t is the disjunction of the two minterms 111000000 and 000111111. This PD is satisfied iff one of the two column encodings 111000000^T or 000111111^T appears in Ξ. It is clear that it takes one whole column of Ξ to satisfy this PD alone. It is sufficient, but not necessary to satisfy the full input constraint [119]. To alleviate the restrictions imposed on Ξ, we may instead use the corresponding elementary input constraints in ϕ. In our example, the full input constraint would then produce the following set of SPDs:

$$[\{3\},\{0,1,2\}] \quad [\{4\},\{0,1,2\}] \quad [\{5\},\{0,1,2\}]$$
$$[\{6\},\{0,1,2\}] \quad [\{7\},\{0,1,2\}] \quad [\{8\},\{0,1,2\}].$$

These SPDs may be satisfied separately, along several columns of the encoding. The satisfaction function t is defined in the same way it was defined for the full input constraint, and is the disjunction of a set of cubes which can be satisfied with more code possibilities than the original satisfaction function. The construction of t is exemplified below.

Example 9.1 (The satisfaction function of input constraints) A full input constraint can be represented by a PD having a satisfaction function equivalent to the disjunction of two minterms. For an elementary input constraint, the satisfaction function becomes the disjunction of two cubes. The elementary input constraint $(\{0,1,2\},\{7\})$, for example, corresponds to a PD where $p = [\{7\},\{0,1,2\}]$, and where the satisfaction function t is the disjunction of two cubes, which in the three-valued switching cube representation are $111----0-$ and $000----1-$. The cube having $x_i = 1$ if $s_i \in p(1)$ is called the ***direct cube of*** t, while the other is the ***reverse cube of*** t.

■

Ciesielski et al [26] and Saldanha et al [103] independently proposed similar approaches to

model output constraints with SPDs. They extended the basic flexible SPD model presented in [119] to allow the use of both fixed and flexible SPDs.

Output constraints, just as the fixed SPDs, are not "symmetric" with regard to the complement of Boolean vectors. Given two states a and b, a dominance relation $a \succ b$ tells that the code assigned to a must dominate the code assigned to b. Stated otherwise, in no column of Ξ a can receive a value 0 if b receives a value 1. This can be translated into one single SPD $\partial = \langle p, t \rangle$, where $p = [\{a\}, \{b\}]$, with the satisfaction function ensuring the dominance of $\Xi(a)$ over $\Xi(b)$ in every valid encoding column. In [26, 103], the same constraints are modeled by two fixed SPDs, one like we just explained, and another to ensure that the codes assigned to the states a and b are distinct. In our case, this SPD would have $p = [\{b\}, \{a\}]$. However, we have showed through Theorems 6.4 and 6.5 that injective constraints are redundant in case all input constraints are respected, and worse, they may prevent the obtainment of compatible codes. Thus, we do not even need to generate them.

Given three states a, b and c, a disjunctive relation $a = b \vee c$ tells that the code assigned to a must be the disjunction, or logic OR of the codes assigned to b and c. As we did for the dominance constraint, consider a bit by bit interpretation for this constraint type. In every column where Ξ assigns 0 to the code of a, neither b nor c can be assigned 1, and in every column where Ξ assigns 1 to the code of a, b and c cannot simultaneously be 0. Disjunctive constraints can then be expressed by three distinct SPDs $[\{a\}, \{b\}]$, $[\{a\}, \{c\}]$ and $[\{b, c\}, \{a\}]$. The satisfaction function t must guarantee that the bit by bit interpretation is retained for every valid encoding. Again, the same observation made about the injective constraints for dominance relations may be applied here, *mutatis mutandis*.

9.2 The Pseudo-Dichotomy Framework

In Chapter 5 we stated the FSM assignment problem, of which SA is a part. The results of Chapter 6 allowed us to extend the SA problem to consider the SM problem as well. The last Section showed how to separately translate each relevant SM and SA constraint kind in terms of PDs. In order to solve simultaneously the SM and SA problems, two tasks are left to describe:

1. the selection of a compatible set of PDs to be satisfied, which defines precisely the problem to be solved;
2. the development of a PD satisfaction method to be applied to this compatible set.

The first task description is accomplished in this Section, while the second is the object of Chapter 12.

We propose the organization of PDs into a framework capable of representing all conditions to be attained by the encoding. Some initial remarks are useful to explain the structure of this framework. To satisfy one input constraint, one column of Ξ with the correct configuration suffices. The other constraints, however, need to be verified across all columns of the encoding simultaneously.

Definition 9.1 (PD framework) *Consider an algebraic structure $\mathcal{F} = \langle F_l, F_g \rangle$, where F_l, F_g are sets of PDs of a set S of symbols. An encoding Ξ of S satisfies $\mathcal{F}$ iff each element in F_l*

is satisfied by at least one column of Ξ *and each element in* F_g *is satisfied by every column of* Ξ. *If an encoding that satisfies* $\mathcal{F}$ *exists,* $\mathcal{F}$ *is called a* PD *framework of* S*, and* F_l *and* F_g *are called the* **local part** *and the* **global part** *of* $\mathcal{F}$*, respectively.*

From the definition of PD framework we see that the local part expresses conditions that need to occur in some column of an encoding Ξ of S, while the global part collects conditions that need to be verified by every column of Ξ. The definition is not dependent upon the specific problem we are trying to solve, being applicable to a wide range of problems, as will be discussed in Section 11.3. Assuming that we do not consider PDs where the satisfaction function $t(\mathbf{x}) = 0$ for every Boolean vector $\mathbf{x}$, a special case of algebraic structure $\mathcal{F} = \langle F_l, F_g \rangle$, where $F_g = \emptyset$, is *always* a PD framework, because an encoding can always be found that satisfies the local part alone. The reason for this statement is that PDs in the local part need be satisfied by one column of the encoding only. Thus, we can add columns to the encoding until all PDs are satisfied.

If it were guaranteed that no conflict could arise among the distinct constraints, the destination of the PDs produced by the translation process of last Section into the framework would be immediate to determine. The PDs derived from the input constraints would go to the local part, while all other constraints would be inserted in the global part. However, this is not the general case, as can be deduced from the study of the relationships among constraint kinds in Chapter 6. Thus, we need first a method to generate a feasible subset of constraints before filling the local and global parts of the framework with the associated PDs. Let us begin with the local part.

9.2.1 Building the Local Part

The input constraints are fundamental, since they supply bounds on the product term cardinality of the two-level hardware implementation solution of the SM and SA problems. However, Theorem 6.5 demonstrated that the input constraints generated by symbolic minimization may violate some of the closure constraints, preventing the occurrence of valid state merging situations in the solution of the problem. If there is a way to keep the bounds of the solution, while avoiding that compatible states be encoded disjointly, an ideal local part would then be obtained. This is indeed the case if we use the following method.

Method 9.1 (Input constraints relaxation) Let $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$ be an FSM, with θ being the compatibility relation graph of $\mathcal{A}$, and ϕ the graph of an input relation on S. Then,

1 **for each** pair $(\{s_i\}, s_k) \in \phi$ **do**
2   **if** there is $(s_l, s_k) \in \theta$ **or** $(s_k, s_l) \in \theta$, such that $s_l \in \{s_i\}$
3   **then**, **eliminate** s_l from $\{s_i\}$ in the pair $(\{s_i\}, s_k) \in \phi$;
4   **if** the resulting set $\{s_i\} = \emptyset$
5   **then**, **eliminate** the pair $(\{s_i\}, s_k)$ from ϕ;

Theorem 9.1 (Input constraints relaxation) *Given an* FSM $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$*, the compatibility relation graph* θ *of* $\mathcal{A}$*, and an input relation graph* ϕ *on* S*, suppose that* ϕ *is the result of the decomposition of the full face embedding constraints arising from symbolic minimization.*

Apply the input constraints relaxation method to ϕ, obtaining a set of relaxed input constraints ϕ'. Then, any state encoding Ξ' of $\mathcal{A}$ that respects all relaxed input constraints in ϕ' and all compatibility constraints in θ, is a valid state assignment of $\mathcal{A}$.

Proof. From Theorems 6.4 and 6.5, we know that respecting ϕ leads to an encoding Ξ that assigns disjoint codes to incompatible states, as well as to conditionally compatible states. We also know, from the same Theorems, that Ξ is a valid encoding. The input constraints relaxation method eliminates all requirements to disjointly encode compatible states in ϕ, and only these. The elimination of all such requirements is guaranteed by the use of the binary relation graph θ in the method, which contains all information about compatible states. As for the *only* part, suppose this is not true. Then, there must exist a pair $(\{s_i\}, s_k) \in \phi$ such that a state $s_j \in \{s_i\}$, incompatible with s_k, is eliminated in some step of the application of the method. This would happen iff either $(s_j, s_k) \in \theta$ or $(s_k, s_j) \in \theta$, which is impossible by the construction of the compatibility relation graph θ.

Thus, only incompatible states are required to be separated by ϕ', and assignment Ξ' may assign disjoint codes only to them, because it respects all compatibility constraints in θ. The encoding Ξ' thus assigns intersecting codes to every compatible in $\mathcal{A}$. Also, the intersection of codes of a set of states where at least one is not compatible with all others is the empty set. This corresponds to an injective encoding of the set of maximal compatibles of $\mathcal{A}$, a closed cover of compatibles, which by Theorem 6.3 is a valid state encoding of $\mathcal{A}$.

∎

Corollary 9.1 (Bounds Preservation) *The input constraints relaxation method does not increase the upper bound on the row cardinality of the encoded cube table predicted by symbolic minimization.*

Proof. Bounds preservation is a direct consequence of the fact that requirements are suppressed by the relaxation method, but no requirement is added to assign disjoint codes to states that were present in the original face embedding constraints of ϕ. Input constraints may then be changed or suppressed, but never created by the input constraint relaxation method.

∎

The input constraints relaxation method corresponds to considering a closed cover of compatibles that is simple to determine, making a complex search for an optimal closed cover of compatibles dispensable. Notice that not even the maximal compatibles generation needs to take place. Also, since all compatible pairs are retained, the considered cover is automatically closed, and the closure constraints may be ignored as well.

We could argue that better results might be obtained if the determination of an optimal subset of closure constraints occurs during constraints generation. The following reasons may be advanced against this statement. First, this would imply the search for an optimal closed cover of compatibles, to furnish to the state assignment step. The immediate consequence of such a choice is to greatly increase the complexity of the constraints generation step. Second, this choice would be barely distinct of a serial strategy, where state minimization precedes pure state assignment, since the final grouping of states would be predetermined before logic minimization. In this thesis, on the other hand, we advocate the use of constraints derived from SM to *increase* the degree of freedom of the subsequent logic minimization step. This

idea relies upon the fact that this step, being at a lower level of abstraction, works with more realistic cost functions than state minimization, and may thus choose the grouping of states more adequately. The benefits of considering simultaneously state minimization and assignment derive thus, although only partly, from our simple choice of a trivially available closed cover of compatibles.

A solution of intermediate complexity, not considered here, could be envisaged, which is to devise heuristic techniques that start with all compatible pairs, and then try to eliminate "a priori bad pairs" before applying the input constraints relaxation method, without violating neither the "closedness" nor the "coverness" of the initial set of state pairs. In this case, consideration of the compatibility relation graph θ and of the closure relation graph σ becomes fundamental, because no compatible pair can be eliminated if some pair retained in the closed cover implies it, otherwise the cover loses its closedness property. The generation of the local part can be understood in practice with the help of an example.

Example 9.2 (Input constraints relaxation) Consider machine BEECOUNT, the FSM used as case study in Chapter 2. There is a full input constraint in this FSM with the form $(\{0,1,3\},\{2,4,5,6\})$, according to Figure 2.2. Decomposing this constraint into the elementary face embedding constraints of relation graph ϕ for this FSM, we obtain:

$$(\{0,1,3\},2),(\{0,1,3\},4),(\{0,1,3\},5),(\{0,1,3\},6).$$

From the compatibility analysis for this machine, we know that the compatibility relation graph θ is the set

$$\{(0,1),(1,0),(0,2),(2,0),(3,4),(4,3),(5,6),(6,5)\}.$$

The application of the input constraints relaxation method to the subset of elementary constraints above gives us the relaxed set:

$$(\{1,3\},2),(\{0,1\},4),(\{0,1,3\},5),(\{0,1,3\},6).$$

■

As for the practical aspects of the pseudo-dichotomy framework, we may devise two distinct ways to build the local part F_l. After the application of the input constraints method and the translation of the relaxed constraints into SPDs, we may directly consider every relaxed constraint as part of F_l. Alternatively, we may include a previous step that eliminates the SPDs covered by some other SPD. In the case of input constraints, the general definition of covering among PDs may be specialized.

Definition 9.2 (Elementary input constraints SPDs covering) *Consider two SPDs $\partial_1 = \langle p_1, t_1\rangle$ and $\partial_2 = \langle p_2, t_2\rangle$ obtained from the translation of a set of elementary input constraints. Then, ∂_1 covers ∂_2 if either*

$$p_2(0) \subseteq p_1(0) \wedge p_2(1) \subseteq p_1(1) \quad \textit{or} \quad p_2(0) \subseteq p_1(1) \wedge p_2(1) \subseteq p_1(0)$$

This last technique allows the subsequent constraint satisfaction step to perform better and faster, but it can be very expensive, since the covering step is NP-complete.

9.2.2 Building the Global Part

We assume here that the initial set of output constraints is feasible internally, i.e. that they can all be satisfied at once by some encoding, and also that they do not violate any input constraint generated by symbolic minimization. The global part F_g is filled by inserting first the PDs corresponding to the compatibility and closure constraints (if any is needed) describing the same closed cover of compatibles used to fill the local part. Then, the output constraints are inserted one at a time, if they neither become redundant with regard to, nor create conflict with the SM constraints, according to the discussions in Section 6.7.

The SPDs generated from the SM constraints were considered in both local and global parts of the framework. We could ask why there is a need to keep them in the global part, since they have already been used to relax the SPDs in the local part. The reason is that, after constructing the framework, the procedure of constraint satisfaction tries to find an optimal way of merging compatible SPDs, in order to obtain the smallest set of PDs covering all SPDs in both parts of the framework. Even if after relaxation no SPD in the local part conflicts with any SPD in the global part, the merging of SPDs to form the PDs of the solution may produce violations of the global part SPDs. This may be prevented by verifying each merging of SPDs in the local part against the SPDs in the global part.

Chapter 10

Conclusions on the Unified Framework

The objective of Part III was to provide a unified representation for the SM and SA problems constraints studied in Part II. We have observed that pseudo-dichotomies provide an adequate algebraic structure to represent every relevant constraint in each of the problems.

Chapter 8 proposed a new definition of the pseudo-dichotomy concept, which is more general than previous approaches. PDs constitute a useful way of modeling a single column of an encoding. The basic innovation of the definition presented here is to allow that dichotomies be represented by a pair $\langle p, t \rangle$, where p is a two-block partition of a subset of a set S, while t, the satisfaction function, is a switching function defined over the set of Boolean vectors with cardinality equal to $|S|$. In fact, the concept is more general than we need. For our purposes of addressing two-level hardware implementations, we accordingly limit attention to PDs where the satisfaction function is a cube over $\mathcal{B}^{|S|}$. The separation of the partition p from the satisfaction function t allows that several kinds of constraints be represented uniformly (for instance, the SM and the SA constraints), with the behavior of the satisfaction function identifying the kind of constraint under consideration. The extended concept allows the definition of pseudo-dichotomies where the satisfaction function is not directly related to the partition p, although this should normally be of little use in modeling encoding columns.

The identification of two markedly distinct kinds of behavior for the constraints led to the proposition of a bipartite framework where PDs representing constraints are inserted. Although the whole development of the framework is directed to allow the simultaneous solution of the SM and SA problems, its proposition was motivated by the insufficiencies of previously suggested frameworks to model the SM constraints [103, 26, 106]. In this way, our PD framework generalizes previous efforts, and can be applied as well to solve problems other than the one addressed in this thesis. We provide examples of such problems in Part IV.

Although the proposed framework is general enough to be applied to other problems, as will be discussed in Chapter 11, its construction is not. The mapping of constraints into a compatible set of PDs, derived in Chapter 9 is specific to the SM and SA problem. In general, before using the framework, a problem dependent analysis of the involved constraints has to be effectuated to determine how the constraints interact, after which a feasible set of them can be generated and translated into a compatible set of PDs.

In the scope of this thesis, the analysis of the constraints permitted to ignore the covering constraints, since states are all encoded, anyway. The closure constraints could be ignored as

well, due to the choice of considering the whole set of elementary compatibility constraints in the framework.

Based on this set, we proposed an original method to relax the input constraints. This relaxation remains the fundamental step in the construction of the framework, because it allows that the subsequent PD satisfaction step assign intersecting codes to compatible states, without compromising neither the validity of the encoding nor the bounds derived from symbolic minimization. Theorem 9.1 demonstrated the correctness of the relaxation method.

Notice that we defined the relaxation method grounded on the use of the whole compatibility relation graph θ. However, Theorem 9.1 may be stated more generally, in terms of any subset of θ that represents a closed cover of compatibles. Such a generalization gives support to the intermediate complexity solutions suggested in Section 9.2.1. The importance of these solutions relies on the fact that they may lead to reductions on the encoding length with regard to the solutions obtained with the current approach. This happens because encoding every compatible state with intersecting codes leads always to the most sparse codes, which adds degrees of freedom to the logic minimization step, but may also add surface to the final two-level implementation, which is an undesirable effect.

Another important benefit of the relaxation method is the preservation of the bounds predicted by symbolic minimization, as demonstrated by Corollary 9.1. Recall that this was not the case when using the serial strategy to solve the SM and SA problems. In fact, Examples 6.4 and 6.5 showed that state minimization done before state assignment may *increase* the bounds predicted by symbolic minimization.

The output constraints treatment proposed here suffers from the lack of an adequate generation method for them, as realized in Chapter 6. Our proposition for these issues is then limited to some basic assumptions, together with the *ad hoc* analysis of their relationship in the same Chapter. A more comprehensive set of heuristic techniques to deal with output constraints is available in [26].

The final achievement of this part is the unification of the SM and SA problems within the PD framework. In fact, the PD framework defines a problem by its construction. The next Part of this thesis addresses the solution of this problem, after describing it formally.

Part IV

Encoding by Constraints Satisfaction

Chapter 11

The Boolean Constrained Encoding Problem

The Boolean constrained encoding problem has a fairly general statement, which can be matched with the requirements of numerous problems in VLSI design. Some general techniques have been proposed to solve it [78, 106]. In this Chapter, we suggest a new statement of the Boolean constrained encoding problem that is a generalization of previous attempts, e.g. of those in [26, 105, 78, 103]. The goal of such a generalization is to include the problem defined by the PD framework presented in Chapter 9 as one of the problems that can be matched with the Boolean constrained encoding problem. In this way, the generalization of previous techniques for solving the latter can be applied to the solution of our problem.

In Section 11.1 the Boolean constrained encoding problem is defined. Then, Section 11.2 states formally the problem which is the object of this thesis, the two-level SM/SA problem, and show how its formulation can be mapped into an instance of the Boolean constrained encoding problem. To this last Section follows a brief discussion of some works dealing with restricted versions of the Boolean constrained encoding problem, in Section 11.3. In Chapter 12, we provide a method to solve the two-level SM/SA problem as a version of the Boolean constrained encoding problem.

11.1 Boolean Constrained Encoding - Statement

Problem Statement 11.1 (Boolean constrained encoding) *Consider two finite sets*

$$S = \{s_0, \ldots, s_{n-1}\} \text{ and } C = \{f_0, \ldots, f_{m-1}\},$$

where the elements $f_i \in C$ are switching functions

$$f_i : \mathcal{B}^n \longrightarrow \mathcal{B}.$$

Associate with each f_i a positive real number $c(f_i)$ and a discrete function g_i

$$g_i : (\mathcal{B}^k)^n \longrightarrow \mathcal{B}.$$

We call the elements of S **symbols***, and the elements of C* **encoding constraints** *on the symbols of S. The number $c(f_i)$ is the* **gain** *of f_i, while g_i is the* **encoding constraint satisfaction function** *of f_i. A constraint f_i is* **satisfied** *by a set $E \subseteq (\mathcal{B}^k)^n$ iff there is an element $e \in E$ such that $g_i(e) = 1$. The satisfaction of a constraint f_i by E is indicated here, with a little notational abuse by*

$$g_i(E) = 1.$$

The Boolean constrained encoding problem can then be stated as follows:

Find a function

$$h : S \longrightarrow \mathcal{P}(\mathcal{B}^k),$$

where $\mathcal{P}(\mathcal{B}^k)$ is the powerset of $\mathcal{B}^k$, and such that

1. *k is minimized;*
2. *the gain $c(h)$ is maximized, where $c(h)$ is defined as*

$$c(h) = \sum_{i=0}^{m-1} c(f_i).g_i(\times_{j=0}^{n-1}(h(s_j)))$$

Function h is denominated an **encoding function** *or simply an* **encoding***, while the integer value k is called the* **encoding length***.*

This enunciate is intended as a general statement for several specific problems. The internal structure of the problem can be best understood by mapping them to a practical instance of the problem, which is done in the following Example.

Example 11.1 (FSM state assignment as input assignment) Consider the state assignment problem proposed in Definitions 5.5, and suppose we approximate the solution of this problem as the input assignment problem, i.e. by using the face embedding constraints generated by symbolic minimization only, and not considering output constraints nor any SM constraints. This is in fact the approach used by the programs DIET [119] and KISS [42], as well as one of the approaches allowed by the program NOVA [116]. Let us express this problem as an instance of the Boolean constrained encoding problem.

The set of symbols S is clearly the set of states of the FSM. The set of encoding constraints C, on the other hand must be computed from the face embedding constraints obtained by symbolic minimization. Consider, for instance, the machine of Example 6.4. The machine is an FSM $\mathcal{A} = \langle I, T, O, \delta, \lambda \rangle$ where $I = O = \mathcal{B}$, $T = \{a, b, c, d\}$, and where δ and λ are given by the flow table in Figure 11.1. In the same Figure, we show the grouping of entries obtained by symbolic minimization. To each grouping corresponds a full face embedding constraint, according to definition of these constraints. Remember that these constraints are designed by the present states associated with the entry group. We have thus six constraints, two of them repeated twice. In our case, the four distinct full face embedding constraints are:

$$(\{a, b\}, \{c, d\}), (\{a\}, \{b, c, d\}), (\{b, c\}, \{a, d\}), (\{d\}, \{a, b, c\}).$$

Flow table and face embedding constraints

	0	1
a	d,1	c,0
b	d,-	a,0
c	d,0	a,0
d	c,1	b,1

Figure 11.1: Flow table and input constraints for FSM $\mathcal{A}$

The last two constraints appear twice. Remember the interpretation of these constraints: two symbols in opposite sides of these constraints must have codes differing in at least one column of the final encoding. To compute the encoding constraints as switching functions, we translate these into pseudo-dichotomies $\partial\langle p, t\rangle$, as explained in Example 8.1. In our case, representing the ps with the value vector notation, and using the state symbols as Boolean variables to describe the functions t as a disjunction of cubes, the pseudo-dichotomies would be, respectively:

$$\begin{aligned}
\partial_1 &= \langle p_1 = [\{c,d\}\{a,b\}], t_1 = ab\overline{c}\overline{d} \vee \overline{a}\overline{b}cd\rangle; \\
\partial_2 &= \langle p_2 = [\{b,c,d\}\{a\}], t_2 = a\overline{b}\overline{c}\overline{d} \vee \overline{a}bcd\rangle; \\
\partial_3 &= \langle p_3 = [\{a,d\}\{b,c\}], t_3 = a\overline{b}\overline{c}d \vee \overline{a}bc\overline{d}\rangle; \\
\partial_4 &= \langle p_4 = [\{a,b,c\}\{d\}], t_4 = abc\overline{d} \vee \overline{a}\overline{b}\overline{c}d\rangle.
\end{aligned}$$

The structure of the functions t derive from the nature of the face embedding constraints. Indeed, suppose that states are encoded in the order $abcd$. To separate states c, d from states a, b (according to the first constraint), we need to have a column in the encoding that is either $(0011)^T$ or $(1100)^T$. The functions t express that if they evaluate to 1, the constraint is respected. Now, the set of encoding constraints C is simply the set of all t functions. Note that there is an important step that has been skipped in the generation of encoding constraints, namely the decomposition of the full input constraints into a set of elementary constraints. To achieve an optimum state assignment, this step should be taken. However, since this would not change the nature of the constraints, only their number and composition, we may disregard it here.

Now, we tackle the modeling of the f_i gain $c(f_i)$ and the satisfaction function g_i. To compute the gains we observe that some constraints are repeated in the cube table obtained by symbolic minimization. We thus assign to each encoding constraint f_i a value $c(f_i)$ that is equal to the number of times the face embedding associated with f_i appears in the symbolically minimized cube table. This choice is justified by the fact that each entry set in the symbolically minimized cube table is associated with one row in the final minimized two-level implementation, according to Theorem 5.2. Then, satisfying a constraint guarantees that all groupings associated with this constraint can be performed in the final implementation. If not all constraints are finally satisfied, we had better choose to satisfy constraints that are repeated several times, since this will hopefully lead to a greater percentage of possible groupings of all those predicted by symbolic minimization.

The constraint satisfaction function g_i, in this specific example, has an expression which

is identical to the corresponding function f_i, but defined over a larger domain. This is a consequence of the fact that face embedding constraints are satisfied by a single encoding column of the result. However, this does not account for the general case. In some problems, a constraint is satisfied if the corresponding function f_i is respected in *every* column of the constraint, like the compatibility constraints or the output constraints. There are even encoding constraints where f_i need to be satisfied in a specific number of columns [44]. That is why the domain of the satisfaction functions g_i are all possible encodings of length k. Given this mapping, the state assignment problem can be stated just as the general Boolean encoding problem, where we need to look for an optimum encoding h of the state set T, such that the length is the minimum possible and the gain c(h) is maximum.

∎

Given the above example on how to map a given encoding problem to the general statement of the Boolean constrained encoding problem, we may give general interpretations for the elements of the formulation.

First, note that the multiplier inside the summation in the expression for $c(h)$, i.e. the expression $g_i(\times_{j=0}^{n-1}(h(s_j)))$, evaluates to 1 iff the constraint f_i is satisfied by the encoding obtained under the usual interpretation given above. Otherwise, the multiplier evaluates to 0.

Let us interpret the meaning of the sets S and C above. S is a set of symbols to be encoded according to the constraints in C. The encoding constraints f_i, on the other hand, map a Boolean vector with the same cardinality as the set of symbols into a binary digit. The most frequent interpretation for this function is that it tells whether a bit configuration participates in the satisfaction of the encoding constraint. To each f_i the problem statement associates g_i, a function that characterizes the encoding constraint. It is through g_i that the behavior of the distinct encoding constraint kinds can be accounted for. The encoding constraint satisfaction function g_i tells, for every possible functional encoding of k bits, if the encoding constraint f_i is satisfied by it.

The encoding function h is the solution of the problem. It associates an arbitrary set of binary k-tuples with each symbol, unlike previous propositions [78, 106], which associate a single k-tuple with each symbol.

Another observation about the above stated problem is that its solution is not unique in the general case. In most instances of the Boolean constrained encoding problem, the optimum solution is found only when considering a trade-off between the goals of minimizing k and maximizing the gain $c(h)$.

All proposals the author could find in the available literature choose to solve restricted versions of the Boolean constrained encoding problem. Two of these proposals are of major importance [116], and we define them below.

Definitions 11.1 (Complete and partial constrained encoding) *In the scope of Boolean constrained encoding problems, choose to satisfy all encoding constraints unconditionally, thus maximizing $c(h)$. At the same time, look for an encoding that minimizes k. This restricted version is called* ***complete (Boolean) constrained encoding****.*

Another restricted version of the Boolean constrained encoding problem is obtained as follows: establish a value for k (often the minimum possible, but not necessarily), looking then

for an encoding with length k that maximizes $c(h)$. This problem is called ***partial (Boolean) constrained encoding****.*

Encoding constraints were modeled in Problem Statement 11.1 as switching functions of n variables. This choice is general enough to represent most kinds of constraints found in encoding problems belonging to logic and low levels of abstraction in VLSI descriptions. Pseudo-dichotomies $\partial = \langle p, t \rangle$, as defined in Section 8.1, are as general as the definition of encoding constraints, due to the form of the satisfaction function t, which is identical to the form of an encoding constraint. However, the function g associated with an encoding constraint may account for the satisfaction of the constraint across the whole set of columns of the encoding, which was not possible with PDs. That is one of the reasons why we needed to provide, in Chapter 9, a framework where to consider the effect of constraints that influence the composition of more than one column of the encoding. We are now able to state the SM/SA problem defined by the pseudo-dichotomy unified framework of Chapter 9.

11.2 The Two-level SM/SA Problem Statement

Problem Statement 11.2 (Two-level SM/SA problem) *Given a finite state machine $\mathcal{A} = \langle I, S, O, \delta, \lambda \rangle$, assume that the SM and SA constraints of $\mathcal{A}$ are obtained by the application of adequate techniques, and that these constraints are translated into the binary relations stated in Chapters 4 and 5. Assume also that a PD framework is constructed from these binary relations according to the techniques described in Section 9.2. Then, the* ***two-level SM/SA problem*** *can be stated as:*

> *Find a minimum two-level implementation of $\mathcal{A}$ corresponding to a set of FPDs Π computed from the PD framework. Since this problem is computationally very hard, consider two approximations of its solution:*
>
> *The first is called* ***complete two-level SM/SA problem****, and it states that Π is such that:*
>
> 1. *every PD in the local part of the framework is covered by at least one FPD in Π;*
> 2. *every FPD in Π covers every PD in the global part of the framework.*
>
> *The second is called* ***partial two-level SM/SA problem****, and it states that Π is such that:*
>
> 1. *it has the minimum code length, no incompatible states in S have intersecting codes, and the number of PDs in the local part of the framework that are covered by at least one FPD in Π is maximized;*
> 2. *every FPD in Π covers every PD in the global part of the framework.*

Given Π, the three-valued representation of each cube corresponding to the satisfaction function of an FPD in Π is a column of an optimal two-level encoding of $\mathcal{A}$ considering state minimization and sate assignment simultaneously. The set of all such columns is the image $\Xi(S)$ of the optimal assignment.

The SM/SA problem have been converted in this way, into an equivalent pseudo-dichotomy covering problem. As was expected, this is still an NP-hard problem, since it is at least as complex as an ordinary minimum cover problem [54].

11.2.1 Boolean Constrained Encoding and the PD Framework

The mapping of an instance of the Boolean constrained encoding problem into an instance of the SM/SA problem can be obtained in a straightforward manner using the statement of the latter in terms of PDs, as done above.

To the set of symbols S of the general problem, associate the set of states of the FSM whose SM/SA problem is to be solved. To the set of constraints C of the general problem, associate the set of satisfaction functions t_i of all PDs inside a PD framework constructed for the FSM from the binary relations describing its SM and SA constraints. The construction of functions t_i for the face embedding constraints has already been illustrated in Example 11.1. Now suppose that two states s_1, s_2 are compatible in an FSM. This corresponds to the existence of a compatibility constraint (s_1, s_2). A pseudo-dichotomy associated to this constraint, according to the conventions of representing p by its value vector, and using the state symbols as Boolean variables, is $\partial_c = \langle p_c = [s_1\ s_2], t_c = \overline{s_1 \oplus s_2}\rangle$, where $\overline{s_1 \oplus s_2}$ stands for the 'Boolean equivalence' function, meaning that the codes for s_1 and s_2 must not be disjoint. For a dominance constraint (s_1, s_2), the pseudo-dichotomy would be $\partial_d = \langle p_d = [s_1\ s_2], t_d = s_2 \rightarrow s_1\rangle$, where $\rightarrow$ stands for the 'Boolean implication' function, meaning that where s_2 is encoded with a 1, s_1 must also be encoded with a 1. Finally, for a disjunctive constraint $(s_1, \{s_2, s_3\})$, we would have three pseudo-dichotomies $\partial_{j1} = \langle p_{j1} = [s_1\ s_2], t_{j1} = s_2 \rightarrow s_1\rangle$, $\partial_{j2} = \langle p_{j2} = [s_1\ s_3], t_{j2} = s_3 \rightarrow s_1\rangle$, $\partial_{j3} = \langle p_{j3} = [s_2 s_3\ s_1], t_{j3} = s_1 \rightarrow (s_2 \wedge s3)\rangle$, with $\wedge$ standing for the usual 'Boolean conjunction' function. The disjunctive constraint can of course be modeled as a single encoding constraint function f_i that is the conjunction of t_{j1}, t_{j2} and t_{j3}.

The behavior of a function g_i to associate with each satisfaction function t_i (which stands from now on for an encoding constraint) depends on which part of the framework a PD ∂_i is located in, with $\partial_i = \langle p_i, t_i\rangle$. If ∂_i is in the local part, g_i is made such that one single column of some functional encoding Ξ satisfying ∂_i causes $g_i(\Xi(S)) = 1$, while ∂_i in the global part implies that we make $g_i(\Xi(S)) = 1$ iff ∂_i is satisfied by every column of the functional encoding.

As for the encoding function h, it is mapped to the state assignment Ξ mentioned before.

The weights $c(f_i)$ of encoding constraints f_i are present in our approach as well, but their characteristics will be discussed while presenting the implementation of our constrained encoding method in Chapter 14.

Given this mapping, which can be verified against the definitions of the general problem as well as of the special case we treat, the SM/SA problem can be treated as an ordinary Boolean constrained encoding problem.

11.3 Solutions to Constrained Encoding Problems

In Chapter 1, we have found mentions to the programs DIET [119] and NOVA [116], that use complete and partial constrained encoding problem formulations, respectively, to solve the SA

problem. The method we present in Chapter 12 can approximate the solution of any one of these simplified problems, but not the general approach, which is more complex to solve automatically. Indeed, the author could find no proposal of a method to exactly solve the general problem in the available literature. Partial constrained encoding, often gives better results. This happens because it generally correlates better with the minimum area cost function, most often the main concern in VLSI design.

The solution of the SA problem in synchronous FSMs is one of the most important applications of Boolean constrained encoding, but not the only one. Boolean constrained encoding has already been used to provide solutions to problems such as:

1. input assignment in combinational circuits [42];
2. output assignment in combinational circuits [45];
3. input and output assignment in combinational circuits [26];
4. race-free state assignment for asynchronous sequential machines [111];
5. delay-free state assignment for asynchronous sequential machines [114];
6. two-way network partitioning [106];
7. two-layer constrained via minimization (in routing algorithms) [106];
8. state assignment guaranteeing a fully testable implementation of an FSM [44].

Given this broadness of applicability some works suggest general solutions for the Boolean constrained encoding problem. Lin and Newton [78] for instance, proposed a generalized approach to solve the Boolean constrained encoding problem, which they call *constrained cubical embedding problem*. Their framework relies upon a user-tunable general solver, implemented using a probabilistic hill climbing algorithm [98]. The application specific constraints and cost functions are not part of the solver, and may be supplied by the user to extend the functionalities of the tool. User intervention for tuning is reinforced by the existence of an interactive graphical interface. The user may in this case be an application designer that produces a tool adequate to solve a given encoding problem. To tune the tool, however, such a user need a deep knowledge of the problem he is trying to solve, so that the general solver can be adequately parameterized.

Shi and Brzozowski [106] proposed another general approach to solve Boolean constrained encoding, and applied it successfully to a number of apparently unrelated practical problems in logic and low level VLSI design. They suggested an effective greedy approach to constraint satisfaction based on PDs, capable of solving problems using either complete or partial constrained encoding.

Along the present thesis, we have seen how SM can be considered during SA, and how a constraint formulation may guide the search for an optimal solution to both problems. We may generalize this statement saying that constrained encoding can benefit from the consideration of compatibility classes inside the problem to be solved. This is the main weakness of the works we just cited. The approaches of Lin and Newton, and Shi and Brzozowski solve a restricted version of the Boolean constrained encoding problem, where the encoding function h is injective and its

image is the subset of $\mathcal{P}(\mathcal{B}^k)$ containing singletons only[1]. As a consequence, the encoding these approaches generate cannot cope with compatibility classes and the associated compatibility constraints, even if both can be captured from the initial problem specification.

The method we describe in the next Chapter does not solve the Boolean constrained encoding in its full extent either. However, it is less restrictive than any other method mentioned in this Chapter. The restriction we impose is due to the search for a two-level solution of the SM and SA problems. We limit the image set of function h to those elements of $\mathcal{P}(\mathcal{B}^k)$ that can be associated with cubes. Eventually, we intend to extend the present work to deal with multiple level solutions of the problem. For instance, the use of Boolean relations [20] as codes for symbols can be considered instead of cubes, which extends the applicability of the method. Although no present constraint satisfaction proposal could be adapted to solve our version of the SM and SA problems, the analysis of the various methods showed that the ideas advanced by the work of Shi and Brzozowski in [106] are useful in our generalization efforts. Our constraint satisfaction method resembles somewhat theirs, in the sense that it is also a greedy approach using the *column encoding* technique to be explained in the next Chapter.

[1]Stating it in practical terms, these approaches assign completely specified functional codes (singletons of $\mathcal{P}(\mathcal{B}^k)$) to the symbols in S, and no two symbols are assigned a same code (injective encoding).

Chapter 12

The ASSTUCE Encoding Method

In Chapter 9, we proposed a PD framework to integrate the SM and SA problems. The obtainment of the framework resulted from the constraint formulation each of the problems received in Chapters 4 and 5 respectively. The PD framework in fact poses a new problem, the two-level SM/SA problem, which has been formally stated in the previous Chapter.

The next Section discusses the principles behind the method we propose, while Section 12.2 gives a detailed overview of the ASSTUCE method.

12.1 Solving the SM/SA Problem

We call the method we developed the ASSTUCE[1] ***method***. It is in fact a generalization of an existing greedy heuristic technique proposed by Shi and Brzozowski in [105]. The original method could not be used, since it is limited to functional encodings, the cube encodings we need being thus impossible to obtain. The main differences between our method and the one proposed in [105] are summarized in the next Chapter.

The idea of the method is to generate one column of the encoding at a time (identified below by the vector ξ), so that the generated columns satisfy each a maximum number of PDs in the local part of the unified framework, and do not violate any PD in the global part. After each column generation step, all satisfied PDs in the local part are eliminated, and column generation proceeds. There are two possible stop conditions, depending on what constrained encoding approach is chosen, complete or partial. If complete constrained encoding is chosen, the method execution stops only when the local part is empty. Otherwise, execution stops when either the local part is empty, or when every two incompatible states are assigned disjoint codes.

Column generation proceeds also in a step by step fashion. At each step, the method computes a single bit to 'move', so as to increase the number of PDs in the local part of the framework that are satisfied by the current contents of the column under generation. A move is a change of an initial value in the column under generation. Just one change is allowed in each position of the column during column generation. Thus, the method evolves linearly, without

[1]The denomination comes from ASSign + redUCE, and it is intended as a homophone of the French word "astuce", meaning trick or astuteness.

backtracking. The initial column configuration is usually a column vector of don't cares, since this is the most flexible configuration, due to the absence of backtracking in the algorithm. For the same reason, any move is either a change to a 1 bit value or to a 0 bit value, never to a don't care value. After a change, a bit position is locked until the present column generation ends.

At each bit move step, sparse matrices are used to compute how far we are from satisfying each constraint in the local part, as well as to calculate how many constraints would be satisfied with a change to 1 or to 0 of each of the bit positions that are not locked. Then, the change leading to the greatest number of satisfied PDs is taken, but only if it does not violate any PD in the global part. This procedure is repeated until either all bit positions are locked or until no valid moves are left, due to violations of the global part in all unlocked positions.

The final encoding is the collection of generated bit columns.

The ASSTUCE method will be presented in three phases. First, we provide an overview of the method through the presentation of an example. Next, we introduce the main data structures used to speed up the execution time of a programmed version of the ASSTUCE method. During this phase we provide bounds for the time complexity implied by the operations in the method. Finally, we introduce some heuristic techniques that can be used to accelerate processing.

In Chapter 14, we discuss an implementation of the method, and compare it with other approaches to solve separately the SM and SA problems.

12.2 ASSTUCE Method Overview

Consider a set of symbols $S = \{0, 1, 2, 3, 4\}$ and $\mathcal{F} = \langle F_l, F_g \rangle$, where F_l and F_g are sets of PDs. Assume that

$$
\begin{aligned}
F_l \;=\; \{ & \langle p_0 = [\emptyset, \{1\}], t_0 = \overline{1} \vee 1 \rangle, \\
& \langle p_1 = [\{1\}, \{3\}], t_1 = \overline{1 \oplus 3} \rangle, \\
& \langle p_2 = [\{4\}, \{1, 2, 3\}], t_2 = 1\,2\,3\,\overline{4} \vee \overline{1}\,\overline{2}\,\overline{3}\,4 \rangle, \\
& \langle p_3 = [\{3, 4\}, \{0, 2\}], t_3 = 0\,2\,\overline{3}\,\overline{4} \vee \overline{0}\,\overline{2}\,3\,4 \rangle \}.
\end{aligned}
$$

Clearly, the satisfaction functions t_i of the PDs in F_l were defined as if they were derived from elementary input constraints. Assume also that $F_g = \emptyset$. Since F_g is empty, $\mathcal{F}$ is guaranteed to be a PD framework. There are two particularities in this framework that we need to explain. First, there is a PD in F_l that has an empty 0-side. Such a PD is always trivial in the context of our problem, but we have introduced it to facilitate the method's presentation. PDs with one empty side are called ***unary* PD*s***, and have been used in other constrained encoding problems [106]. Second, the global part of the framework is also empty. This was done to simplify the discussion. While treating the example, we shall refer to the global part without taking into account the fact that it is trivially satisfied.

In the following discussion, assume that the global part of the framework is stored somewhere, and that we may consult it to verify if a given PD belongs to it. In Section 12.3 we will discuss data structures to represent F_g. The PDs in F_l, on the other hand, will be represented using a matrix formulation. The PD ***matrix*** P contains one row for each element of S, and one column for each PD in F_l. The components p_{ij} are taken from the three-valued set $\{0, 1, -\}$, and

each column p_j corresponds to a three-valued switching cube representation of the direct cube of the PD satisfaction function associated to the column. Stated more simply, each position i of a column j of P contains a 1 if the element s_i is in the 1-side of a PD ∂_j, contains a 0 if s_i is in the 0-side of ∂_j, and contains $-$ otherwise. The $-$ value is called by the usual denomination of ***don't care***. The matrix P for our example is displayed below, together with the *encoding vector* ξ, to be introduced next.

$$
\begin{array}{c}
\\ 0 \\ 1 \\ 2 \\ 3 \\ 4
\end{array}
\quad
\overset{P}{
\begin{array}{cccc}
p_0 & p_1 & p_2 & p_3
\end{array}
}
\quad
\begin{pmatrix}
- & - & - & 1 \\
1 & 0 & 1 & - \\
- & - & 1 & 1 \\
- & 1 & 1 & 0 \\
- & - & 0 & 0
\end{pmatrix}
\quad
\xi = \begin{pmatrix}
- \\ - \\ - \\ - \\ -
\end{pmatrix}
$$

Starting from the above matrix and vector, we depict one typical iteration of the algorithm.

12.2.1 The First Iteration

The ***encoding vector*** ξ is a column vector with length equal to the cardinality of S. It represents a single column of the final encoding Ξ. The ASSTUCE method accepts as input the initial configuration of this vector. If none is given, it generates the *all don't cares* vector above. This is in fact the least restraining configuration that may be specified, since our approach is to produce one column encoding at a time, through the execution of a series of "moves" on the ξ vector. Each ***move*** is a change of the value of a component of ξ. The allowed changes depend on the initial value of ξ, but at any given moment, no change can make the vector contain more don't cares than in any previous step. Stated otherwise, no change from 1 or 0 to $-$ is allowed. This justifies the statement that the all don't cares version of ξ is the less restraining initial configuration, and it is a consequence of the choice of a greedy constructive algorithm that generates column encodings without backtracking.

The First Move - From matrix P and vector ξ we produce an ***evaluation matrix*** E, which contains information about how far we are from satisfying the PDs in P using vector ξ. E is a matrix of pairs with the same row and column cardinalities as P, and the coordinate values of each pair are taken from the same three-valued set used to construct P and ξ. The rules to construct the components e_{ij} of E, given p_{ij} of P and ξ_i of ξ, appear in Table 12.1

Table 12.1: Rules for building the evaluation matrix E

$\xi_i \backslash p_{ij}$	0	1	-
0	(1,-)	(-,1)	(-,-)
1	(-,0)	(0,-)	(-,-)
-	(1,0)	(0,1)	(-,-)

Remember that a PD in F_l is satisfied if the direct or the reverse cubes of its satisfaction function evaluate to 1 (cf. Section 9.2.1). In what follows, ***to satisfy a cube*** will mean to make

it to evaluate to 1. In a pair e_{ij}, the first coordinate refers to the reverse cube $\overline{c_j}$ associated to the satisfaction function of the PD represented in column j, while the second coordinate refers to the direct cube c_j. A value 0 in any coordinate of a pair e_{ij} means that a change of the current value of the component ξ_i to 0 will make the encoding vector ξ closer to satisfy the corresponding (reverse or direct) cube. A value 1 means that a change of the current value of the component ξ_i to 1 will make the encoding vector ξ closer to satisfy the corresponding (reverse or direct) cube, and a value $-$ in any coordinate tells that it is not possible to get closer to satisfy the cube, because it is already satisfied in position i by the current value ξ_i.

Now, the number of moves needed to satisfy each PD can be computed. Since there are two ways to satisfy each PD, there are two such numbers, and they are gathered into a vector of pairs of integers, which we call the ***distance vector*** ν. The components ν_j of ν are computed by simply counting the number of non-don't care values in a given column, and this for each coordinate. The number of presently unsatisfied PDs u is the number of components of ν where no coordinate is 0. Matrix E, vector ν and u, which are generated by considering P and ξ above are:

$$\overset{P^{(1)}}{\begin{pmatrix} - & - & - & 1 \\ 1 & 0 & 1 & - \\ - & - & 1 & 1 \\ - & 1 & 1 & 0 \\ - & - & 0 & 0 \end{pmatrix}} \quad \overset{\xi^{(1)}}{\begin{pmatrix} - \\ - \\ - \\ - \\ - \end{pmatrix}} \quad \overset{E^{(1)}}{\begin{pmatrix} (-,-) & (-,-) & (-,-) & (0,1) \\ (0,1) & (1,0) & (0,1) & (-,-) \\ (-,-) & (-,-) & (0,1) & (0,1) \\ (-,-) & (0,1) & (0,1) & (1,0) \\ (-,-) & (-,-) & (1,0) & (1,0) \end{pmatrix}}$$

$$u^{(1)} = 4 \qquad \overset{\nu^{(1)}}{\begin{pmatrix} (1,1) & (2,2) & (4,4) & (4,4) \end{pmatrix}}.$$

Since $u \neq 0$, we have to choose a component of vector ξ to change, in order to get the as close as possible to a column encoding that maximizes the number of satisfied PDs. To do so, we define a ***direction matrix*** D, which carries information about in which direction a single change of a component in vector ξ can satisfy a PD from F_l. D is a pair matrix, just like E, but with a distinct interpretation for its values, and a distinct generation method. The first and second coordinates of a pair d_{ij} in D correspond to changes to 0 and to 1, respectively. The values of the d_{ij} pairs are thus interpreted as follows:

$$\begin{cases} (1,1), & \text{if a change of } \xi_i \text{ to either 1 or 0 satisfies one of } \overline{c_j},\ c_j; \\ (0,-), & \text{if a change of } \xi_i \text{ to 0 unsatisfies one of } \overline{c_j},\ c_j; \\ (-,0), & \text{if a change of } \xi_i \text{ to 1 unsatisfies one of } \overline{c_j},\ c_j; \\ (1,-), & \text{if a change of } \xi_i \text{ to 0 satisfies one of } \overline{c_j},\ c_j; \\ (-,1), & \text{if a change of } \xi_i \text{ to 1 satisfies one of } \overline{c_j},\ c_j; \\ (-,-), & \text{otherwise.} \end{cases}$$

The configurations not shown either never happen or represent trivial cases. The first configuration arises only in a trivial case (from which the first PD in P is an example) and in a special case. In fact, the trivial case PD of our example is irrelevant in practice [106]. In the list above, *unsatisfies* means *turns a satisfied cube into an unsatisfied one*.

Matrix D can be obtained from an inspection of the distance vector ν and the evaluation matrix E. There are various possible situations, but only five non-trivial ones. If we have

$\nu_j = (0, \geq 2)$ (resp. $\nu_j = (\geq 2, 0)$), the cube $\overline{c_j}$ (resp. c_j) is already satisfied, and any change of a component ξ_i such that $p_{ij} \neq -$ can only increase u, the number of unsatisfied PDs. If, on the other hand, we have $\nu_j = (1, \geq 2)$ (resp. $\nu_j = (\geq 2, 1)$), there is a component ξ_i that, if changed, will satisfy the cube $\overline{c_j}$ (resp. c_j). Finally, there is the special case where $\nu_j = (1, 1)$ and the direct and reverse cube associated to column j have only two non-don't cares in their three-valued representation. Then, there are exactly two possible single changes such that one can lead to the satisfaction of $\overline{c_j}$ and the other to the satisfaction of c_j. Obviously, $\nu_j = (\geq 2, \geq 2)$ implies that no single change of one component of ξ can satisfy any direct or reverse cube.

To accumulate the values of matrix D and determine the best move, we use a ***gain vector*** ω. Vector ω is a column vector of pairs of integers. Its contents are obtained as a componentwise sum of pairs over all columns of matrix D, for each D row. The sum is done as follows:

1. if a coordinate of a pair d_{ij} is 1, add 1 to the same coordinate of ω_i;
2. if a coordinate of a pair d_{ij} is 0, add -1 to the same coordinate of ω_i;
3. if a coordinate of a pair d_{ij} is -, add 0 to the same coordinate of ω_i;

The direction matrix D and the gain vector ω resulting for our example are:

$$\overset{D^{(1)}}{\begin{pmatrix} (-,-) & (-,-) & (-,-) & (-,-) \\ (1,1) & (-,-) & (-,-) & (-,-) \\ (-,-) & (-,-) & (-,-) & (-,-) \\ (-,-) & (-,-) & (-,-) & (-,-) \\ (-,-) & (-,-) & (-,-) & (-,-) \end{pmatrix}} \quad \overset{\omega^{(1)}}{\begin{pmatrix} (0,0) \\ (1,1) \\ (0,0) \\ (0,0) \\ (0,0) \end{pmatrix}} \quad \text{select } \xi_1 \rightarrow 0$$

The first coordinate of a component ω_i of ω gives the gain of changing ξ_i to 0, while the second coordinate gives the gain of a change to 1. The best choice is the one with the greatest positive gain. In our example, there are two such best choices. Either we select a change of ξ_1 to 0 or to 1. Suppose that one selection is made. At this point we must verify if the change does not violate any of the global constraints. Suppose that a violation occurs. Then, the change is discarded and a second choice is tried, and so on. If no feasible change exists, none is performed, and the encoding column generated up to now is a column of the final encoding. Our example has an empty global part, and thus such conflicts will not occur here. The first choice is then arbitrarily taken, changing ξ. This is what we call the ***first move***, indicated by the exponent (1) on the denomination of the structures.

The Second Move - After the first move, the component ξ_1 is ***locked*** (indicated by the symbol $\times$ in the corresponding position of the gain vector ω), and cannot change anymore

during this column generation. The computations for the next move are summarized below.

$$\xi^{(2)} = \begin{pmatrix} - \\ 0 \\ - \\ - \\ - \end{pmatrix} \quad E^{(2)} = \begin{pmatrix} (-,-) & (-,-) & (-,-) & (0,1) \\ (-,1) & (1,-) & (-,1) & (-,-) \\ (-,-) & (-,-) & (0,1) & (0,1) \\ (-,-) & (0,1) & (0,1) & (1,0) \\ (-,-) & (-,-) & (1,0) & (1,0) \end{pmatrix}$$

$$\nu^{(2)} = \begin{pmatrix} (0,1) & (2,1) & (3,4) & (4,4) \end{pmatrix} \qquad u^{(2)} = 3$$

$$D^{(2)} = \begin{pmatrix} (-,-) & (-,-) & (-,-) & (-,-) \\ (-,1) & (-,-) & (-,-) & (-,-) \\ (-,-) & (-,-) & (-,-) & (-,-) \\ (-,-) & (-,1) & (-,-) & (-,-) \\ (-,-) & (-,-) & (-,-) & (-,-) \end{pmatrix} \quad \omega^{(2)} = \begin{pmatrix} (0,0) \\ \times \\ (0,0) \\ (0,1) \\ (0,0) \end{pmatrix} \quad \text{select } \xi_3 \to 1$$

The End of the Iteration - This process goes on until no more moves are possible. The impossibility of making moves arises either:

1. when all positions of ξ are locked, or
2. when all possible moves violate a PD in the global part F_g, or
3. when only negative gains exist in ω for every non-locked position in ξ that does not violate a PD in the global part F_g, indicating that any allowed change will decrease the number of satisfied PDs.

One heuristic choice that can be used is to make moves whenever one is possible, until all positions of ξ are locked, retaining the configuration of ξ that satisfied most constraints, and not necessarily the last one.

Suppose that after the second move, we make a third move, by selecting $\xi_2 \to 0$ and a fourth move selecting $\xi_4 \to 1$. There is still room for another move in the example, which is:

$$\xi^{(5)} = \begin{pmatrix} - \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} \quad E^{(5)} = \begin{pmatrix} (-,-) & (-,-) & (-,-) & (0,1) \\ (-,1) & (1,-) & (-,1) & (-,-) \\ (-,-) & (-,-) & (-,1) & (-,1) \\ (-,-) & (0,-) & (0,-) & (-,0) \\ (-,-) & (-,-) & (-,0) & (-,0) \end{pmatrix}$$

$$\nu^{(5)} = \begin{pmatrix} (0,1) & (2,0) & (1,4) & (1,4) \end{pmatrix} \qquad u^{(5)} = 2$$

$$D^{(5)} = \begin{pmatrix} (-,-) & (-,-) & (-,-) & (1,-) \\ (-,1) & (-,0) & (-,-) & (-,-) \\ (-,-) & (-,-) & (-,-) & (-,-) \\ (-,-) & (0,-) & (1,-) & (-,-) \\ (-,-) & (-,-) & (-,-) & (-,-) \end{pmatrix} \quad \omega^{(5)} = \begin{pmatrix} (1,0) \\ \times \\ \times \\ \times \\ \times \end{pmatrix} \quad \begin{matrix} \text{select } \xi_0 \to 0 \\ \text{and stop.} \end{matrix}$$

12.2.2 The Subsequent Iterations

After the fifth move, all positions in ξ are locked, a first column encoding has been generated, namely 00011^T, which satisfies three of the four PDs describing the problem. A new iteration is needed, since there are unsatisfied PDs left. All satisfied PDs are eliminated from matrix P, transforming it into a smaller PD matrix, which is the input of the new iteration, together with the specified vector ξ. Note that the vector ξ need not be the same at each step. This is useful if, for example, some symbol codes are to be preestablished. Assuming the starting ξ vector to be the same as before, the start of the new iteration appears below.

$$
\begin{array}{cccc}
\xi^{(1)} & & & \\
\begin{pmatrix} - \\ - \\ - \\ - \\ - \end{pmatrix} & E^{(1)} & = & \begin{pmatrix} (-,-) \\ (0,1) \\ (0,1) \\ (0,1) \\ (1,0) \end{pmatrix} \\
& \nu^{(1)} & = & \begin{pmatrix} (4,4) \end{pmatrix} \quad u^{(1)} = 1
\end{array}
$$

The associated direction matrix D would be an all don't cares column vector, and the weight vector would have only pairs of the type $(0,0)$. A danger arises in the situation where the maximum gains are 0, given an all don't cares initial encoding vector. Since no direction was given about how to choose one of the **to 0** or **to 1** moves when they have identical weights, the method may pick one arbitrarily. In the example, we have to make at least three moves before obtaining a matrix D that contains anything except don't cares. Suppose that the method picks moves arbitrarily, but deterministically, and that the moves $\xi_2 \rightarrow 1$ and $\xi_3 \rightarrow 0$ are the first and second ones. Then, there will be no further way of satisfying the PD. The encoding column would be added to the encoding and the iteration would repeat the same procedure ad infinitum, without never finding a satisfying vector for the PD. To avoid this, every time that the maximum gains in the ω vector are zero, we choose to satisfy always the direct cubes. This ensures that the column generation always stops. For our example, we would thus obtain a vector ξ like -1110^T satisfying the last PD, and the method would have come to the end.

The final encoding respecting all constraints would be obtained from the concatenation of the two encoding columns, which gives

symbol	*code*
0	0−
1	01
2	01
3	11
4	10.

Note that the encoding is neither functional, nor injective, what we wanted to be able to do, and that it satisfies the framework.

12.2.3 ASSTUCE Method Discussion

The above method is heuristic, with no backtracking. The main advantage of its application is the execution speed. Indeed, our practical implementation, to be discussed in Part V, iterates on encoding generation, not only on column encoding generation, without great expense in computation time, even for big examples. However, some points should be pointed out about the method:

1. several choices have to be made during the execution, like: which best move to take, in which direction, and what to do when only null gains are computed, which happens frequently in the initial steps of the iteration;

2. only locally optimal solutions can be obtained, even if iteration over a first solution is possible and applied;

3. direct matrix computation may be substituted by sparse matrix techniques to accelerate execution, as we discuss below.

The computational cost of the column encoding generation problem is influenced mainly by four tasks:

1. **Task 1:** computation of the evaluation matrix E and of the distance vector ν;

2. **Task 2:** computation of the direction matrix D and of the gain vector ω;

3. **Task 3:** selection of the component in ω with the maximum gain;

4. **Task 4:** verification of the feasibility of performing a move on the selected component.

Suppose that n is the cardinality of the set of symbols S to encode, and that m is the cardinality of the local part F_l of the framework. Suppose also that the number of operations to verify if a move does not violate a PD in the global part F_g of the framework is constant. Suppose finally that we use the natural choices, namely one-dimensional vectors for ν, ω and two-dimensional arrays for E, D. A column encoding generation consists in an iteration repeated at most n times where:

1. E, ν, D, ω are built with complexity bounded by $\mathcal{O}(n.m)$;

2. a best component to change is selected. Suppose that selection of the best component takes a constant number of operations $\mathcal{O}(1)$; we will show later that this is possible. Each move has to be verified against at most $(n-1)$ positions in ξ, giving a complexity bounded by $\mathcal{O}(n)$;

3. a move is made and the associated component of ξ is locked, using a constant number of operations $(\mathcal{O}(1))$.

The final bound on the complexity of column encoding generation is then $\mathcal{O}(n^2.m)$.

The next Sections show that the use of special data structures may reduce this complexity, so that it is a function of the number c of non-don't care components in the initial PD matrix P, instead of a function of $m.n$. Note that c is at most $m.n$, the total number of entries in matrix P, and that in practice it is quite smaller, specially for large examples.

12.3 The ASSTUCE Method Data Structures

The next two Sections describe the data structures used to accelerate the computation in the ASSTUCE method.

12.3.1 Data Structure for the Global Part

In the previous Section, we have not presented any structure for the global part F_g of the framework. This part is constructed only once, before the execution of the method, and the only operation performed on it during execution is to consult it to see if some move in ξ violates any PD inside F_g. In order to perform consult operations efficiently, consider the PDs inside F_g. Any such PD corresponds to an elementary compatibility, dominance or disjunctive constraint, with either two symbols (in the case of compatibility and dominance constraints) or three symbols (in the case of disjunctive constraints) inside it. Internally, symbols are obviously converted to integers to enhance ease of manipulation, according to some natural enumeration order defined on S. We may then use a three-level hash array based on dynamic perfect hashing techniques [46] to store all PDs of the global part. Dynamic perfect hashing guarantees constant access time to its contents. Since the number of access levels is bounded, the worst case complexity of a consult operation is still $\mathcal{O}(1)$. Thus the above supposition holds, and the bounds above can indeed be attained. Task 4 has thus a complexity of $\mathcal{O}(n)$ for a single move, and $\mathcal{O}(n^2)$ for a column generation.

12.3.2 Data Structures for the Local Part

We present below a set of data structures and a discussion on how to accelerate Tasks 1, 3 and 2 listed in the previous Section, in this order.

Task 1 - The main structure of the local part is the evaluation matrix E, since the PD matrix P is used only once to build the first version of E. After that, the contents of E may be incrementally updated, from the changes in the contents of the ξ vector. There are only two types of operations done in E. One operation is to visit all non-don't care components of a row to change one or two of its coordinates. The other is to visit all non-don't care components of a column to verify its contribution to the associated component of the distance vector ν. For this purpose, a data structure similar to the one suggested in [105] is used, which is bipartite. The contents of E are duplicated and put into two sparse arrays: one efficient for row operations, and the other efficient for column operations. These data structures are described below and exemplified in Figure 12.1, with the contents they would display during the computation of the first move in the previous example.

- **Symbol array** Y - A one-dimensional array, where each entry corresponds to a symbol s_i of S, and which contains a pointer to a linked list of PDs relevant to s_i. Each element of the list contains an index j of a PD that contains s_i in either its 1-side or its 0-side, and the corresponding pair e_{ij} of E. A symbol array example is depicted in Figure 12.1(a);
- PD **array** T - A one-dimensional array, where each entry corresponds to a PD p_j of F_l, and which contains two components: a pointer to a linked list of symbols in either side of

p_j and a pair of integers representing the associated component of the distance vector ν. Each element of the linked list contains an index i of a symbol in either the 1-side or the 0-side of p_j, and the corresponding pair e_{ij} of E. A PD array example appears in Figure 12.1(b);

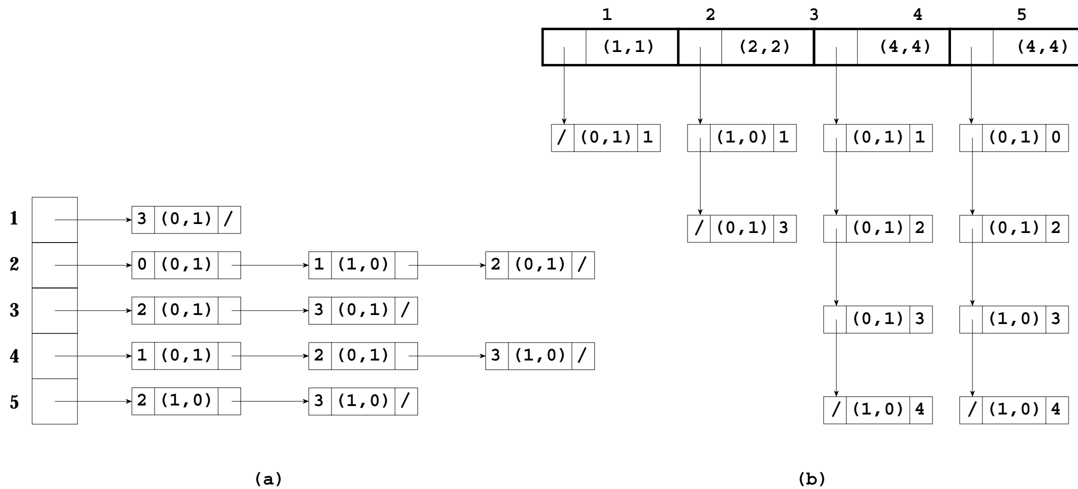


Figure 12.1: Symbol array and PD array during first move computation

We build the initial Y and T arrays from the set of PDs in F_l and from ξ. P need not be constructed, since all of its information are already inside F_l. For each element p_{ij} in some PD p_j Table 12.1 is consulted once to obtain the pair to fill both Y and T entries, and the constructed entry is inserted in the beginning of list i of Y, and in the beginning of list j of T. At the same time we use the pair obtained from the table look-up operation to determine values to add to the coordinates of the corresponding component of the distance vector ν. All operations take constant time, which gives complexity $\mathcal{O}(c)$ (recall that c is the number of non-don't care entries in the initial PD matrix P.) for the simultaneous construction of Y, T and ν.

To build the subsequent versions of Y, T and ν, total rebuilding is unnecessary, since only one row change may occur for each move. The new contents of a row of Y and of the distance vector ν after a move may be determined by using a look-up technique defined in Table 12.2. This Table gives the new values of e_{ij} of E as a function of the move taken and of the previous value of e_{ij}. Here, E stands for the matrix represented by the bipartite structure formed by Y and T. In the same Table, we indicate the action to update the related vector ν entry coordinates. The first coordinate of a pair in ν is denoted by x and the second by y. The signs $+$ and $-$ to the right of x and y indicate increment and decrement operations, respectively.

Table 12.2: Rules to build E and ν incrementally after a move

Move\\e_{ij}	(1,0)	(0,1)	(-,0)	(0,-)	(-,1)	(1,-)
- $\rightarrow$ 1	(-,0),$x-$	(0,-),$y-$				
- $\rightarrow$ 0	(1,-),$y-$	(-,1),$x-$				
1 $\rightarrow$ 0			(1,-),$x+$,$y-$	(-,1),$x-$,$y+$		
0 $\rightarrow$ 1					(0,-),$x+$,$y-$	(-,0),$x-$,$y+$

Suppose that the number of PDs in F_l containing a symbol s_i in either 0-side or 1-side is h_i. Since there is at most n moves during a column generation, the maximum number of update operations over Y, T and ν along an encoding column generation step is $\sum_{i=0}^{n-1} h_i = c$. Then, the total cost of Task 1 is bounded by $\mathcal{O}(c)$.

Task 3 - We will show that ω, just like E (represented by Y and T) and ν, can be computed incrementally. There are five operations implied in the search of a best move candidate component of ω:

1. to compute a gain coordinate of a component of ω, given its index;
2. to insert a component into ω;
3. to update a component of ω and sort it according to the best component criterion;
4. to find the next component of ω with a maximum gain given its index, if any exists;
5. to delete a component of ω;

An adequate data structure for these operations is used in bucket sort techniques [27], and it is called ***bucket list***. Figure 12.2 displays this data structure. The **Bucket** B is a list of pointers to doubly linked lists with $2.k+1$ pointers, where $k = max\{h_i, 0 \leq i \leq n\}$, where h_i is the number of non-don't care entries in row i of the PD matrix P at each column generation step. Stated in words, k is the cardinality of the largest subset of PDs in F_l that contains symbol s_i in either its 0-side or its 1-side. This gives the range of gains that can be achieved in ω at any moment during encoding column generation. The linked lists contain only unlocked components of ω.

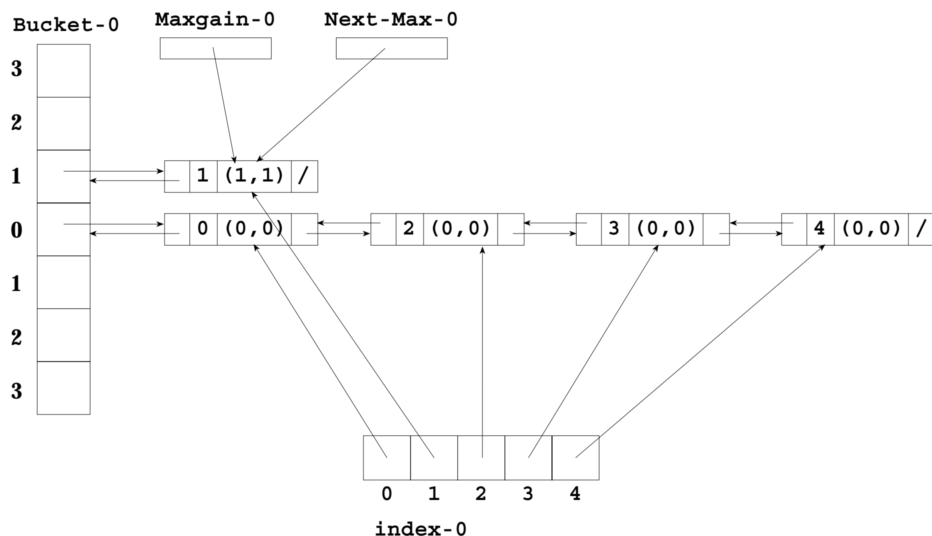


Figure 12.2: Bucket list B_0 before the first move

Each list comprises elements with the same coordinate gain. Since each gain is a pair, we employ two bucket lists: one sorted by the gains of a change to 0 (B_0), and the other sorted by the gains of a change to 1 (B_1). Figure 12.2 shows the state of B_0 only. In this particular

state, namely before the first move, B_1 contains exactly the same information. Each entry in a linked list has the index i of the symbol s_i associated to a gain pair ω_i and the contents of the pair itself. There is an **index** array to allow direct access by index to the elements of B, and two auxiliary pointers, namely **Maxgain** and **Next-Max** to keep track of the best move component at any time.

Every time some component ξ_i is locked after a move, the entries corresponding to ω_i in B_0 and B_1 (one in each of these) are eliminated, and the auxiliary pointers are updated, if needed. The existence of two such pointers is due to the possibility that the best move is not taken because it would violate a PD in the global part. To avoid discussing the bookkeeping details between the two bucket lists, assume that at a given moment several best choices are in just one of them. Then, if the absolute best gain is not possible to be taken, we use **Next-Max** to navigate the bucket list, looking for the next best choice until a feasible one is found, without deleting unfeasible choices. One could ask why not deleting these choices. The reason is that, in the next move, such best choices may be allowed to be taken, because the just performed move eliminated the violation implied in the previous partial encoding. That is why the pointer **Maxgain** keeps track of the best of these choices. After each move, and after deleting the entry associated to the move taken, **Next-Max** is reset to the same value as **Maxgain**.

As for the complexity of the operations, computing a gain coordinate of a given component of ω takes a constant number of operations, using direct access through **index**; inserting a component in ω is also done in $\mathcal{O}(1)$, since insertion can be done anywhere in the list (the order of elements in the lists is immaterial), and because the eventual update of the auxiliary pointers is done in constant time; the last two operations, i.e. finding the next best component and deleting a component of ω, have the same worst-case complexity $\mathcal{O}(n)$, since they may require a search based on either the **index** or the **bucket**. In practice, this happens only in very big problems, where many empty **bucket** elements may arise. For instance, if the best component is not the last element in some list, these operations will take always constant time.

Task 2 - The objective of this Task is to compute the direction matrix D, as well as the gain vector ω. First, given the bucket list structure, we do not need to build the direction matrix at all. We may simply accumulate the values directly in the respective positions of the bucket lists B_0 and B_1. First, we will explain how to construct B_0 and B_1, based on the contents of Table 12.3. Afterwards, we see how to incrementally update both bucket lists.

Table 12.3: Rules to build B_0 and B_1 from E and ν

$e_{ij}\backslash\nu_j$	$(1,\geq 2)$	$(\geq 2,1)$	$(0,\geq 2)$	$(\geq 2,0)$	$(1,1)$
(0,1)	(1,0)	(0,1)			
(1,0)	(0,1)	(1,0)			
(1,-)	(0,1)	(0,0)		(0,-1)	(0,1)
(-,1)	(0,0)	(0,1)	(0,-1)		(0,1)
(0,-)	(1,0)	(0,0)		(-1,0)	(1,0)
(-,0)	(0,0)	(1,0)	(-1,0)		(1,0)

Given an initial ξ, we have seen how to simultaneously obtain from the PDs in F_l, a representation of the evaluation matrix E and of the distance vector ν, using data structures Y and

T. Given the interpretation of the matrix E components presented in Section 12.2.1, and the rules for constructing ω based on it, we may use Table 12.3 for the incremental construction of the bucket lists B_0 and B_1, from Y and T.

Each entry in Table 12.3 corresponds to a pair to sum coordinatewise with the contents of a pair in ω, represented in both B_0 and B_1. Given this look-up table, the initial states of B_0 and B_1 are obtained by the following procedure, where a component $\nu_j = (x_j, y_j)$ is an element of ν.

```
1    build bucket lists B0 and B1, and initialize them empty;
2    for each list l_i in the symbol array Y
3      do create a pair of integers b, and initialize it to (0,0);
4           for each element e_ij in l_i
5               do if (x_j < 2  or  y_j < 2)
6                      add the element in entry (e_ij, ν_j) of Table 12.3 to b;
7      insert (i, b) into B0 and B1, according to b value;
```

The addition of pairs in line 6 of the procedure is done coordinatewise. The execution of the above procedure is $\mathcal{O}(c)$, since there is exactly c elements in Y, and since all operations take constant time. The complexity of building the initial bucket lists is thus determined.

As for the updating of B_0 and B_1, notice that after a move in row i of ξ, not all columns of D change. In fact, E is changed only in row i by this move. In general a row of E has much less than the maximum m elements, since we use elementary constraints, which correspond to sparse SPDs. Thus, the number of columns that may change in D, which is exactly the number of elements h_i in row i of E, is normally small. We may thus accelerate the average complexity of updating B_0 and B_1 (which represent ω) by subtracting the old values of these columns and adding the new values. The worst case complexity of the procedure, considered along the whole column generation, is again $\mathcal{O}(c)$. Then, the complexity of Task 2 is bounded by $\mathcal{O}(c)$.

The total worst-case complexity of the ASSTUCE method is then $\mathcal{O}(n^2 + c)$.

12.4 ASSTUCE Heuristic Improvements and Extensions

Some heuristic techniques may enhance the efficiency of the method exposed in the preceding Sections. Also, the addition of some more information on the basic data structures may allow more problems to be tackled with this method.

12.4.1 Improvements

If during a column encoding generation the algorithm does not stop when only negative gains are available, it may escape from some local minima in the solution space. An additional copy of a data structure is needed to keep the column that performed best up to the present moment, i.e. the one that obtains the smallest possible value for u, which is the number of unsatisfied PDs. The worst case complexity of the algorithm does not change if we use this heuristic, but processing will surely take additional time.

After generating a complete encoding, iterate using the last column generated in the previous step as the first initial column of the problem. The reasoning behind this optimization is that the last PDs that have been satisfied are intrinsically "hard" to satisfy. Then, satisfying them first may reduce the needs of the subsequent steps. This leads to the remark that all but the last column generated in one encoding step may be redundant, since columns generated afterwards may satisfy PDs already satisfied. A post-verification can then be made to avoid redundant columns.

Since the technique is fast, the encoding generation may be repeated several times with various initial configurations for the initial vector ξ. This normally leads to a small number of iterations until a local minimum is reached.

The use of balanced bit assignments may enhance significantly the method. Suppose a Boolean vector $\mathbf{x}$. The best way to separate the greatest number of elements in $\mathbf{x}$ is to have the number of zeroes and ones in $\mathbf{x}$ closest to half the number of bits in the vector. In this way, if we choose the moves so as to consider the balancing of the final codes, we may minimize the number of columns to satisfy all PDs.

12.4.2 Extensions

During the generation of the constraints for an FSM, we may identify that some constraints are more important than others, and that if not all constraints are satisfied, it is better to satisfy first the more important ones. This can be done by assigning weights to the constraints, and changing the algorithm accordingly. The only important change is in the dimensioning of the bucket lists, since the use of non-unit gains implies a change in its number of buckets. However, this is trivial to consider.

One example of weighted constraints may be easily identified. After symbolic minimization, we obtain a cube table from where the input constraints are extracted, as discussed for example, in Chapter 2. Since some of the constraints may appear several times in the cube table, we need to consider only the distinct ones, which we do in the encoding phase. However, suppose that a partial SM/SA problem must be solved. Then, some constraints may remain unsatisfied by the encoding. Since the cube table cardinality is the upper bound of the final cube table, if we satisfy preferentially the constraints that appear several times in the table, we may stay close to the bound, even if violating it. Thus, the number of times an input constraint appears in the symbolically minimized cube table is a good measure for the weight of this constraint.

In some problems, maybe some symbol codes are known to be good in advance. Then the problem may be specified with codes fixed for some states. This can be accommodated into the method by imposing values on the successive configurations of the initial encoding vector ξ, and using the possibility of locking elements from the beginning.

Chapter 13

Conclusions on Constraint Satisfaction

In the present Part, we proposed a new statement for the Boolean constrained encoding problem, to which several problems can be reduced. Our proposition generalizes previous approaches, so as to consider encodings that need neither be functional nor injective. Even the previous formulation of the problem was already general enough to be useful to a significant set of distinct applications, which can be realized from the applications list of Section 11.3.

The main objective of generalizing the statement of this problem was to make it comprise the two-level SM/SA problem defined in Section 11.2. After showing informally that the two-level SM/SA problem can be solved as a special case of Boolean constrained encoding problem, we proposed a method to approximate the solution of the former that is a generalization of one approach to the latter.

All previous methods to treat encoding problems using PDs [26, 103, 111, 113, 119] rely upon techniques to solve PD satisfaction that are similar to those used in logic minimization, consisting of two phases: the generation of "prime PDs", and the solution of the associated covering problem. However, some difficulties arise with these techniques. First, both phases comprise the solution of NP-complete problems, and second, prime-covering is adapted to solve the complete two-level SM/SA problem only, because all codes are generated at the same time, during the covering phase. These difficulties do not arise with the greedy column encoding method proposed in [106], which makes it more adapted to the needs of this work.

The approach of Shi and Brzozowski [106] has the merit of being efficient, while obtaining effective results. Also, their method was shown to be applicable to several constrained encoding problems. However, this approach cannot be used to tackle the two-level SM/SA problem, since it generates functional encodings only, although not necessarily injective.

The intent behind the ASSTUCE method proposition is to verify the feasibility of the theoretical findings of this thesis, and to allow experimentation of heuristic techniques to enhance the practical results obtained with it. This has been achieved through the enhanced possibility of parameterizing the basic greedy approach. Indeed, the method allows, for instance, that either complete or partial constrained encoding approximations be employed, that the generation of balanced codes (codes with approximately the same number of 0s and 1s) be favored, etc.

The generalization of the method proposed in [106], although not trivial, could ensure the same worst case complexity as the original method. In [106], the authors computed the worst case complexity as $\mathcal{O}(c)$, while in Chapter 12 we obtained a bound $\mathcal{O}(n^2 + c)$ for the ASSTUCE

method. The difference is immediately explained, if we consider that the work of Shi and Brzozowski addresses no constraint imposing global conditions on the encoding, such as the output or the SM constraints. They suggest techniques to treat the output constraints, but do not account for them during the complexity analysis. Recall that the term n^2 in the complexity bound comes directly from the need of verifying column encodings against the PDs in the global part of the framework. Now, consider the simple case where the output constraints are ignored and the problem FSM has no compatible pair of states. The verification step against the global part can accordingly be dropped, because the global part is empty, and the complexity reduces to $\mathcal{O}(c)$, as in the original method.

The main change that has been done in the original method is to allow that the encoding vector ξ accept don't care values, which implied extensive changes in the assumptions of Shi and Brzozowski. The ASSTUCE method can be used to tackle any of the problems listed in Section 11.3, with one exception, discussed in Section 15.2. The verification of this fact can be obtained by realizing that Shi and Brzozowski already demonstrated this possibility, and that our method generalizes theirs. In addition, we extended the complexity analysis in [106] to consider the output constraints influence.

Chapter 14 will present the characteristics of the program implementing the ASSTUCE method, together with a comparison with other programs addressing the SM and SA problems.

Part V

Implementation, Results and Final Remarks

Chapter 14

Implementation and Benchmark Results

This Chapter discusses the implementation of the ASSTUCE method in the form of a computer program, and the benchmark results obtained by comparing it with other approaches to the solution of the SM and SA problems.

14.1 The ASSTUCE Implementation

The ASSTUCE program comprises five modules, which are responsible for the following tasks:

1. SM constraints generation;
2. input constraints generation;
3. input constraints relaxation and framework construction;
4. ASSTUCE method application;
5. encoded FSM combinational part minimization.

The execution flow of the program is illustrated in Figure 14.1. In this Figure, rectangular boxes stand for data repositories, rounded boxes stand for executable modules and the arrows indicates the data flow. Processing transforms an FSM behavioral description into an encoded FSM description.

The first two modules are straightforward implementations of well-known constraint generation techniques, namely the compatibility table of Paull and Unger [91], and symbolic minimization with the help of ESPRESSO [99, 43], respectively. The third module performs the following steps:

1. breaks the constraints into their elementary forms, i.e. pairs of binary relations, cf. Chapters 4 and 5;
2. performs the relaxation of the input constraints based on the SM constraints, according to Method 9.1;

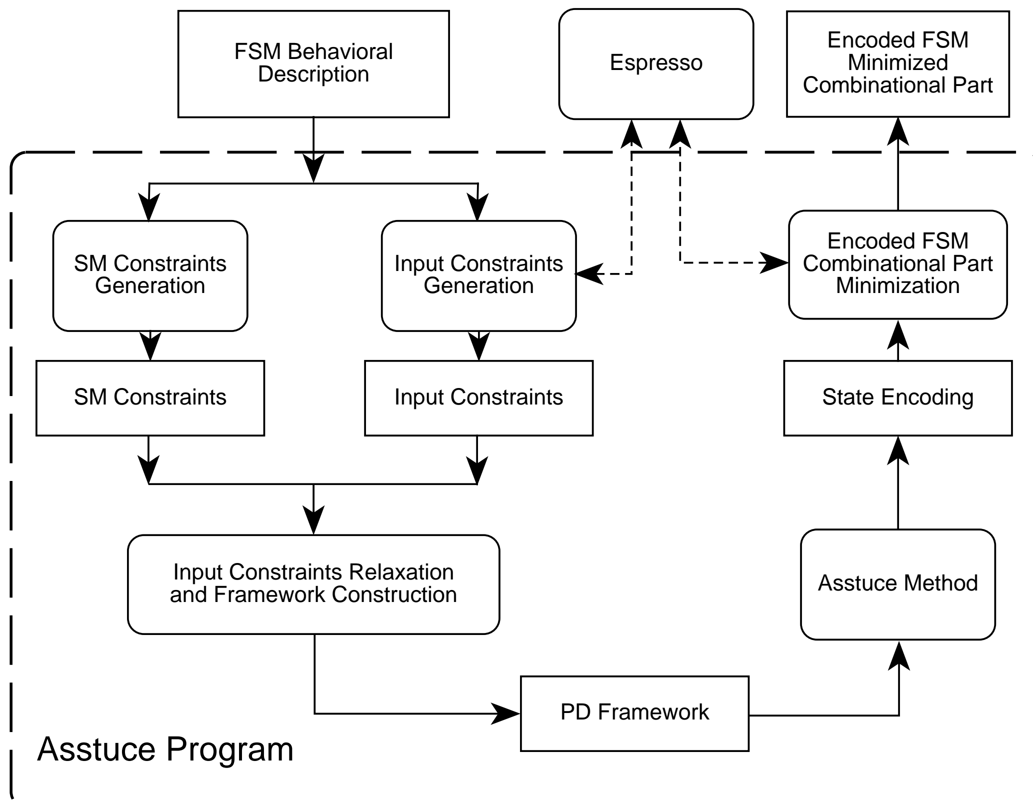


Figure 14.1: Execution flow for the ASSTUCE program

3. generates the local and global parts of the PD framework from the relaxed input constraints and from the SM constraints, according to the discussion of Section 9.2.

All processing in the first three modules is accomplished without major complex tasks being performed, except if covered PDs in the global part of the framework are required (by the user) to be discarded during execution of the third module. The fourth task consists in the application of the ASSTUCE method to generate the state encoding, and the last task uses this encoding to generate an encoded FSM, minimizes the combinational part of this FSM and outputs the result of the minimization.

Note the use of the ESPRESSO program by ASSTUCE in Figure 14.1. The dotted arrows in this Figure indicate the calling of ESPRESSO in two situations: to perform symbolic minimization of the original FSM next state and output functions, and to minimize the combinational part of the encoded FSM.

We describe the program ASSTUCE based on a pseudo-code that uses indentation as an indication of the program block structures [27]. Routines are represented by SMALL CAPITAL characters, while pseudo-code keywords appear in **boldface**. In the pseudo-code, $\leftarrow$ indicates the assignment operation, + denotes concatenation of data structures and # stands for the computation of the cardinality of a data structure. Assigning the empty set symbol $\emptyset$ to a data structure means initializing it to contain conceptually no data. Parameters followed by **:ref** in the header of a routine are passed by reference. Otherwise, they are passed by value. Routines in the pseudo-code are called ***functions*** if they return a value, otherwise they are called ***procedures***.

Of the five tasks executed in sequence by the ASSTUCE program, only one deserves detailed attention, the application of the ASSTUCE method, because it comprises the costliest step in terms of global execution time, which is the column encoding performed by the function GENERATE_COLUMN. This function is called by a function named SATISFY_ONCE, which controls the generation of a complete valid state encoding. The main routine of the ASSTUCE method implementation, which is not shown here, iterates calling SATISFY_ONCE until a local minimum for the encoding length is achieved. The iteration relies upon heuristic ordering techniques. One of these techniques has already been presented in Section 12.4, and consists in starting PD satisfaction using the last column encoding found during the previous call to SATISFY_ONCE. Another technique is to alternate the order in which elements with the same best gain value in the bucket lists B_0 and B_1 are considered. In practice, the number of iterations needed to reach a local minimum is small.

Consider the pseudo-code for functions SATISFY_ONCE and GENERATE_COLUMN. The variables of the pseudo-code are represented using the same terminology employed to describe the data structures of the method in Chapter 12, with minor changes to enhance readability. In this way, Ξ represents the encoding under construction. It is a two-dimensional array to which columns are added during the execution of SATISFY_ONCE. The symbol ξ represents the encoding vector, while $\mathcal{F}$ stands for the PD framework. The variable YT corresponds to the symbol and PD arrays seen as a single entity, u keeps the current number of unsatisfied PDs, and ω refers to the bucket lists B_0 and B_1, which are used to represent the gain vector.

```
SATISFY_ONCE(ξ,F:ref)
1     Ξ ← ∅;
2     YT ← INITIALIZE_YT(ξ,F);
3     YT_start ← YT;
4     u ← COMPUTE_u(YT);
5     while u > 0
6       do (col,sat) ← GENERATE_COLUMN(ξ,F,YT);
7          Ξ ← Ξ + col;
8          YT ← YT_start;
9          YT ← ELIMINATE_SAT(YT,sat);
10         YT_start ← YT;
11    return Ξ;
```

In SATISFY_ONCE, after initializing Ξ, YT is created and filled with data computed from the contents of ξ and of the $\mathcal{F}$ local part, as explained in Section 12.3.2. A copy of this value is kept in an auxiliary data structure YT_start, which is of the same type as YT. Recall that the distance vector is kept within the PD array data structure. This is the reason why we use YT as parameter of the function COMPUTE_u, which returns the starting number of unsatisfied PDs by consulting the distance vector. This information is tested as the stop condition of the loop that controls encoding generation, in line 5 of the function. This loop consists in calling the function GENERATE_COLUMN to compute a new encoding column, receiving as return value a pair where the first coordinate, col, is the new column, and the second coordinate, sat, is the set of PDs satisfied by this column. The structure col is then concatenated with the other columns in Ξ. The variable u is global. The last task in the loop is to obtain a new YT data structure, by eliminating from it the satisfied PDs in sat. Since, for efficiency reasons, YT was passed by reference to GENERATE_COLUMN, it has possibly been changed. That is the reason why a

copy of it is kept in YT_start. After the updating of YT, YT_start is accordingly transformed into a copy of it for the next step. The last action of the function is to return Ξ, an encoding satisfying all PDs in $\mathcal{F}$.

Some comments are needed on the above pseudo-code. It corresponds to an algorithm that approximates the solution of the complete two-level SM/SA problem only, because the stop condition for the loop, in line 5, is $u \leq 0$, which means that the computation of columns stops only after satisfying all PDs. In fact, the user of ASSTUCE may choose between this approximation and the partial constrained encoding approximation. The program can be parameterized in various other aspects, but we overlook these details for ease of understanding of the basic algorithm.

One relevant comment about parameterization is that the initial value of the encoding vector ξ can be established in many ways. If, for example, an initial ξ is specified that contains no don't care values, a binary encoding is automatically generated, which is a useful choice to encode machines without compatible states using minimum length codes. Finally, although not implemented at present, the specification of fixed codes for states is an immediate extension. It suffices to choose ξ as a vector of columns filled with the chosen fixed codes that are stored initially locked in ξ.

We now present the pseudo-code of the function GENERATE_COLUMN.

```
GENERATE_COLUMN(ξ,F:ref,YT:ref)
1     sat ← COMPUTE_SAT(ξ,YT);
2     ω ← INITIALIZE_ω(YT);
3     COMPUTE_GAINS(ω,YT);
4     move ← NEXT_BEST_MOVE(ω)
5     ξ_best ← ξ;
6     while move≠ ∅ and u > 0
7       do if    MOVE_VALID(move,ξ,F)
8                LOCK_SYMBOL(move,ξ);
9                ELIMINATE_MOVED(move,ω);
10               UPDATE_GAINS(move,YT,ω);
11               PERFORM_MOVE(move,ξ);
12               sat_aux ← COMPUTE_SAT(ξ,YT);
13               if #(sat_aux) ≥ #(sat)
14                  sat ← sat_aux;
15                  ξ_best ← ξ;
16               u ← COMPUTE_u(YT);
17          move ← NEXT_BEST_MOVE (ω);
18    return (ξ_best,sat);
```

The function begins by computing the set of satisfied PDs in the local part of $\mathcal{F}$ by consulting the ξ and YT data structures. This routine is similar to COMPUTE_u. In fact, both routines perform just a look-up in YT, since during its set-up all satisfied PDs are identified. COMPUTE_u retrieve the number of unsatisfied PDs and COMPUTE_SAT retrieve the satisfied PDs themselves. The routines INITIALIZE_ω and COMPUTE_GAINS build the initial bucket lists as described in Section 12.3.2. The initialization phase ends with the computation of the first best move (in line 4) and the copy of ξ to the auxiliary vector ξ_best (in line 5).

The main loop extends from lines 6 through 17, and it is repeated while both a feasible move is available and there are unsatisfied constraints. The objective of the loop is to generate the best possible move in ξ at each iteration, if such a move is feasible. The whole loop execution depends upon the function MOVE_VALID, which returns a **yes** or **no** answer about the feasibility of taking the move. The tests performed by this function can be controlled externally by the user of the ASSTUCE program, who may choose to accept negative gains for a move or not, according to the first heuristic technique discussed in Section 12.4.1. If the move is not valid, the next best moves are computed and the loop restart, until either a stop condition of the loop is met or a feasible move is found. Suppose the last case occurs. Then, the associated symbol of ξ is locked (line 8), the corresponding elements of the bucket lists in ω are eliminated (line 9), gains are updated in YT and ω (line 10), and the move is taken (line 11). The last step before computing the next best move and iterating again is to find out if the move just taken is the best one up to now. In fact, this is useful only when the heuristic to accept negative gains is activated, otherwise the move taken makes the test succeed in all cases. If this is indeed the case, the return values ξ_best and sat are updated in lines 14 and 15, respectively. These are the values returned after the loop is finished.

One important observation about column encoding is that there is a heuristic technique to reduce code length in ASSTUCE. This technique consists in forcing the bit vectors generated to be balanced, i.e. to contain as many 0s as 1s, as far as possible. This can assure that each bit vector thus generated satisfies a maximum number of constraints, as described in Section 12.4.1.

14.1.1 The ASSTUCE Program Implementation Environment

We programmed ASSTUCE using the language C++ [108], under the UNIX operating system. Additionally, we employed a class library called LEDA [88], developed at the Max-Planck-Institut für Informatik at Saarbrücken, Germany, to help the implementation of programs dealing with combinatorial computing. The compiler used was G++, developed at the Free Software Foundation [110]. All these programming tools are freely available for non-profit use and research work.

The choice of C++ as programming language was motivated by various reasons. First, C++ is already the most widespread language that supports the object-oriented programming paradigm. This paradigm is seen today as a powerful technique to build modular, maintainable and portable software systems [84]. It allows reusable software to be more easily constructed through various mechanisms such as encapsulation, information hiding, polymorphism and inheritance [117]. One practical example of reusable software is the LEDA library, where data structures like lists, dictionaries and graphs are available, which can not only be used as they were originally designed, but most importantly, they can be adapted for particular applications through the use of inheritance mechanisms. This means, for instance, that if some application requires a special type of graph and the **graph** class of the LEDA library is insufficient to model it, the *user* may create a new **mygraph** class that *is* a graph (and as such inherits all characteristics of the LEDA **graph** class) but to which the *user* can add functionalities and specific data structures. C++ is also a good compromise between runtime execution efficiency and high-level programming, because it is derived from the very efficient language C [71], and yet it allows programming at a high level of abstraction to take place.

As for the program design technique, we used the pragmatic approach proposed in [117] to

guarantee a modular implementation of the prototype. Such care was taken with the ASSTUCE program design because we devise the present prototype as the seed of an FSM exploratory environment. A first requirements specification for such an environment is available in Appendix B.

14.2 Benchmark Tests

A widely available set of benchmark FSMs has been used to compare the results of running the ASSTUCE program with other approaches. The present Section discusses the results of this comparison. The tests were conducted in a single workstation, a Sun Sparcstation 10/40 with a single processor, in a UNIX environment. All execution times mentioned in the comparisons below will be stated in seconds of processing time on this workstation.

The main objective of implementing the program ASSTUCE is to show that the simultaneous strategy for solving the SM and SA problems *can* generate encodings of quality at least as high as the serial strategy, and that in some cases it is the best choice. The available literature [8, 6, 60, 75] has failed to demonstrate these possibilities. We hope that the greedy exploratory implementation will be able to provide results leading to the proposal of a more specific method to tackle the optimal generation of non-functional, non-injective encodings. In this way, the objective of implementing ASSTUCE has been to create an exploratory environment to test the efficiency of the theoretical developments proposed in the thesis, while allowing a good amount of user parameterization, to evaluate various alternative encoding schemes. That is why we call ASSTUCE an exploratory tool, in opposition to a "push-button" encoding program that works more like a black box receiving an input description and furnishing an output description.

14.2.1 The Benchmark FSMs

The FSM test set used is part of the MCNC benchmarks [118], distributed freely by the Microelectronics Center of North Carolina, and comprising various sets of sequential and combinational circuit descriptions. The MCNC FSM test set includes 53 synchronous machines, each identified by a specific name.

The FSMs are represented in the KISS2 format, described in Appendix A. This is the only input format accepted by the current version of the ASSTUCE program. The KISS2 format assumes the FSMs are to be implemented as synchronous machines and uses a cube table to describe the next state and output functions. In this format, machines are presented with input and outputs already binary encoded, while the state information is depicted symbolically.

From the 53 machines, we used only 47, because some machines have all states mutually compatible, which implies that these can be implemented as a combinational circuit, and are thus irrelevant for our purposes.

We divided the 47 machines into two groups, according to the presence or absence of non-trivial compatible state pairs in the original description. Remember that a machine with only trivial compatible pairs of states is already minimized, implying the uselessness of submitting it to state minimization tools. The groups we have separated the machines into are:

1. the ***C-group*** of FSMs, comprising descriptions where at least one non-trivial pair of compatible states exists (18 machines out of 47);

2. the ***I-group*** of FSMs, comprising descriptions where only trivial pairs of compatible states exists (29 machines out of 47).

The objective of dividing machines in two groups is that each group allows evaluating distinct aspects of the two-level SM/SA problem solution proposed by ASSTUCE. The C-group permits accessing the benefits and drawbacks of using a simultaneous strategy in place of a serial strategy. The I-group, on the other hand, is immune to state minimization. Thus, it only allows us to evaluate the benefits and drawbacks of using non-functional encodings instead of functional encodings.

The MCNC FSM benchmark test set input, output and state characteristics are presented in Table 14.1 for the C-group, and Table 14.2 for the I-group.

Table 14.1: Characteristics of the MCNC FSM benchmark test set - C-group

FSM	i_b	i_p	st	o_b	o_p	st_tr
s27	4	19	6	1	2	34
beecount	3	7	7	4	4	28
lion9	2	4	9	1	2	25
ex5	2	4	9	2	2	32
ex7	2	4	10	2	2	36
ex3	2	4	10	2	4	36
bbara	4	6	10	2	3	60
opus	5	12	10	6	8	22
train11	2	4	11	1	2	25
mark1	5	11	15	16	9	22
sse	7	23	16	7	15	56
bbsse	7	23	16	7	15	56
ex2	2	4	19	2	2	72
tma	7	12	20	6	20	44
ex1	9	48	20	19	60	138
tbk	6	49	32	3	5	1569
scf	27	45	121	56	39	166
s298	3	13	218	6	5	1096

i_b : number of input bits
i_p : number of distinct input bit patterns in initial description
st : number of state in initial description
o_b : number of output bits
o_p : number of distinct output bit patterns in initial description

In these Tables, each row corresponds to an FSM, identified by a name in the first column. Note that the FSMs cover a wide range of machine sizes (from 4 to 218 states, from 1 to 27 input bits and from 1 to 56 output bits).

14.2.2 Benchmark Tests Strategy

The present version of ASSTUCE can solve the complete constrained encoding problem approximation, and it can approach the solution of the partial constrained encoding problem

Table 14.2: Characteristics of the MCNC FSM benchmark test set - I-group

FSM	i_b	i_p	st	o_b	o_p	st_tr
lion	2	7	4	1	2	11
train4	2	4	4	1	2	14
dk15	3	8	4	5	11	32
mc	3	7	4	5	8	10
tav	4	22	4	4	12	49
bbtas	2	4	6	2	4	24
dk27	1	2	7	2	3	14
dk14	3	8	7	5	12	56
shiftreg	1	2	8	1	2	16
dk17	2	4	8	3	5	32
ex6	5	24	8	8	12	34
s386	7	39	13	7	11	64
ex4	6	11	14	9	11	21
dk512	1	2	15	3	4	30
cse	7	32	16	7	14	91
kirkman	12	65	16	6	32	370
s208	11	53	18	2	4	153
s420	19	52	18	2	4	137
keyb	7	77	19	2	4	170
s1	8	67	20	6	20	107
pma	8	28	24	8	24	73
s820	18	119	25	19	22	32
s832	18	128	25	19	22	245
dk16	2	4	27	3	5	108
styr	9	40	30	10	28	166
sand	11	122	32	9	36	184
s510	19	37	47	7	13	77
s1494	8	124	48	19	64	250
s1488	8	123	48	19	64	251

i_b : number of input bits
i_p : number of distinct input bit patterns in initial description
st : number of state in initial description
o_b : number of output bits
o_p : number of distinct output bit patterns in initial description

approximation. Accordingly, the strategy of the benchmark tests is to compare the results obtained by ASSTUCE with the results of two alternative serial strategies.

The tools we used to form the serial strategies to compare with ASSTUCE are:

1. STAMINA [59], from the University of Colorado, at Boulder;
2. NOVA [116], version 3.2, developed at the University of California, at Berkeley;
3. DIET [119], version 1.1, developed at the University of Massachussets at Amherst.

The first program, STAMINA, is a state minimization tool. The last two programs, NOVA and DIET are state assignment programs.

DIET employs a complete constrained encoding approach to solve state assignment, while NOVA employs a partial constrained encoding approach. None of these programs consider state minimization, and both generate functional, binary encodings. We have correspondingly performed two sets of benchmarks tests comparisons:

1. First set of comparisons - ASSTUCE versus complete encoding serial strategy - each original benchmark FSM is encoded with ASSTUCE, which is parameterized to perform *complete* encoding; the obtained results are compared with the following serial strategy:
   (a) first, minimize the state cardinality of each original benchmark FSM using STAMINA;
   (b) second, encode each state minimized FSM using DIET.
2. Second set of comparisons - ASSTUCE versus partial encoding serial strategy - each original benchmark FSM is encoded with ASSTUCE, which is parameterized to perform *partial* encoding; the obtained results are compared with the following serial strategy:
   (a) first, minimize the state cardinality of each original benchmark FSM using STAMINA;
   (b) second, encode each state minimized FSM using NOVA.

Some fundamental comments about these tools are necessary before discussing the tests results.

First, all comparison parameters are extracted from the minimized combinational part of the encoded FSM. All three programs, DIET, NOVA, and ASSTUCE, rely on the ESPRESSO program to perform the combinational part minimization after encoding. The same statement is true for the input constraints generation step. In this way, the comparisons reflect the differences arising from the encoding strategy alone, not from side effects originated from the use of distinct minimization schemes.

Second, each of the tools compared with ASSTUCE may be parameterized in some way. Thus, we need to choose an adequate set of run-time options for each program, to allow a fair comparison with the implemented ASSTUCE capabilities. We refer below to the options described in the tool manuals reproduced in Appendix C.

Both DIET and NOVA can effectuate a post-processing of the generated encoding to choose how to assign the all 0s code to a state. This post-processing is called *code rotation*, and the advantage of using it is the possibility of obtaining enhanced results after the combinational

part minimization. Since this is not a capability implemented in ASSTUCE, and since it is independent of a particular encoding algorithm, we have chosen not to employ this option in the tests (i.e. we include neither the option **-z** in DIET nor **-r** in NOVA.).

DIET has no other relevant run-time options. However, various state assignment algorithms are available in NOVA (exact, greedy, hybrid, simulated annealing, etc.). We have chosen to use the run-time option **-e ih**, which invokes a constrained encoding algorithm based on the satisfaction of the input constraints only, and which, according to the tool manual "has the best trade-off between quality of results versus computing time". We avoided the use of algorithms considering output constraints, to maintain a fair comparison with ASSTUCE, since these constraints are not considered in our implementation. Concerning the state minimization tool STAMINA, we have opted for the run-time option **-s 1**, all other options being the default choices. This invokes a tight upper bound heuristic algorithm for performing state minimization. We avoided the exact minimization default option **-s 0**, since it may lead to an exponential growth in the time to obtain a solution of the SM problem.

14.2.3 The Compared Parameters

Most logic level FSM synthesis programs consider area minimization as the primary concern. We have already discussed in Chapter 2 the existence of other criteria to measure the quality of an FSM implementation. In the benchmark tests, we shall consider a total of six distinct criteria to evaluate the quality of a two-level implementation:

1. combinational part area estimate (PLA);
2. number of product terms in PLA;
3. FSM encoding length;
4. PLA transistor cardinality;
5. PLA sparsity;
6. program execution time.

The first three criteria are often present in comparisons, together with the last criterion. Transistor cardinality and PLA sparsity, on the other hand, are important in other logic or layout level applications, such as multiple-level logic minimization and topological optimization, respectively. The number of transistors can be associated with dissipated power, while sparsity gives an idea of how easily topological optimization can be applied to the PLA. At the logic level, all six criteria can only be estimated, but their simultaneous consideration can tell much about the quality of an FSM synthesis program.

The presentation of the benchmark tests results will take place in the next three Sections. In Section 14.2.4, we provide information about running our program on the FSM benchmark test set alone. Sections 14.2.5 and 14.2.6 provide a comparison between ASSTUCE and the complete and partial encoding serial strategies, respectively.

14.2.4 Benchmark Tests with ASSTUCE

The raw data obtained from running ASSTUCE on the MCNC FSM benchmark test set are presented in Tables 14.3 and 14.4, for the C-group and I-group of machines, respectively. The I-group Table has no column accounting for the number of non-trivial compatible pairs in the machine (column **cpt**, since this value would always be zero for any machine in the group. All other columns are present in both Tables. We highlight the fact that the data in these Tables were obtained by running ASSTUCE with an option to perform complete constrained encoding. The other possibility would be to use the partial constrained encoding option, which in general give better results. However this changes little the relationship between modules with regard to the compared parameters.

The five modules comprising ASSTUCE have been enumerated in Section 14.1. The benchmark data concerning each of these modules appear as contiguous column subsets in Tables 14.3 and 14.4, and are distributed as follows:

1. SM constraints generation - the first two columns in the C-group Table, and the first column in the I-group Table;
2. input constraints generation - columns **ict_t** through **t_ict**;
3. input constraints relaxation and framework construction - columns **lpd_e** through **elim**;
4. ASSTUCE method application - columns **pds_it** through **t_pds**;
5. encoded FSM combinational part minimization - columns **t_min** through **a_area**.

Finally, the last column of Tables 14.2.4 and 14.2.4, **a_time**, gives the total time taken to run ASSTUCE, i.e. executing the five modules, including the time for the two ESPRESSO calls.

The PLA area estimate employed is the usual topological estimation for an unfolded PLA [6], i.e. considering it as a rectangle where the height is proportional to the number of product terms, and where the width is proportional to the sum of the number of output bits and twice the number of input bits. The factor 2 results from the fact that both direct and complemented values of the input variables are present in the input plane of the PLA. Using data from Tables 14.1 through 14.4, we may compute the area estimate for each machine as

$$((\mathbf{i_b} + \mathbf{a_cl_f}).2 + (\mathbf{a_cl_f} + \mathbf{o_b})).\mathbf{a_pt}.$$

14.2.4.1 ASSTUCE Benchmark Tests Discussion

The first conclusion we may extract from the ASSTUCE execution alone is that all MCNC benchmarks could be successfully encoded, which will not the case for the DIET program, for instance. This is a direct consequence of the use of efficient heuristic techniques.

Second, we may draw a parallel between the total execution time taken by ASSTUCE and the predicted upper bound on it, established in Section 12.3.2. We may observe in Tables 14.3 and 14.4 that the dominant component of the **a_time** value is the time taken to satisfy the PDs, **t_pds**. Thus, it should be possible to approximate the asymptotic complexity of running ASSTUCE by using the asymptotic complexity of the associated PD satisfaction method. This

Table 14.3: Raw data for the ASSTUCE run on the C-group of FSM benchmarks

FSM	cpt	t_cpt	ict_t	ict_d	t_ict	lpd_e	t_lpd_e	lpd_ne	t_lpd_ne	elim	pds_it	cl_i	a_cl_f	t_pds	t_min	a_pt	a_area	a_time
s27	1	0.00	17	11	0.02	16	0.03	35	0.02	no	4	3	3	0.18	0.03	11	198	0.28
beecount	4	0.02	12	12	0.02	16	0.03	40	0.02	no	5	3	3	0.33	0.05	10	190	0.45
lion9	9	0.00	10	10	0.00	31	0.07	57	0.02	no	3	5	5	0.32	0.00	7	140	0.38
ex5	26	0.03	25	14	0.02	11	0.02	17	0.02	yes	5	4	4	0.10	0.02	9	162	0.18
ex7	32	0.02	23	14	0.02	9	0.03	19	0.02	yes	5	4	4	0.12	0.03	9	162	0.25
ex3	37	0.05	24	14	0.03	7	0.02	9	0.02	no	3	4	4	0.07	0.03	8	144	0.20
bbara	6	0.03	34	14	0.03	20	0.08	59	0.02	yes	3	5	5	0.30	0.05	24	600	0.42
opus	1	0.02	19	11	0.03	32	0.05	60	0.02	yes	3	4	4	0.18	0.02	19	532	0.38
train11	25	0.02	15	13	0.02	30	0.03	58	0.02	yes	3	4	4	0.22	0.00	5	85	0.33
mark1	20	0.07	19	12	0.03	61	0.10	107	0.05	no	3	5	5	0.53	0.05	18	738	0.78
sse	36	0.07	30	18	0.08	51	0.18	122	0.03	no	4	6	6	0.87	0.10	27	1053	1.27
bbsse	36	0.07	30	18	0.08	51	0.22	122	0.07	no	4	6	6	0.85	0.10	27	1053	1.25
ex2	129	0.15	42	27	0.05	23	0.12	66	0.07	yes	3	11	11	0.38	0.03	21	819	0.85
tma	12	0.03	32	27	0.07	140	0.67	317	0.15	no	3	9	9	2.40	0.08	34	1598	2.87
ex1	2	0.05	44	27	0.35	69	0.63	292	0.13	no	3	8	8	2.38	0.48	42	2562	3.92
tbk	16	0.32	173	106	1.78	676	24.93	2254	1.17	yes	5	24	23	38.17	2.23	94	7896	68.82
scf	70	13.88	151	107	1.57	5623	81.68	8484	10.18	no	7	9	9	265.23	1.87	133	18221	298.17
s298	1565	12.43	697	181	7.48	4181	1914.00	27557	50.28	no	6	19	18	4710.00	2.53	300	19800	4828.00

cpt : number of non-trivial compatible pairs in initial description
t_cpt : time needed to generate all non-trivial compatible pairs

ict_t : total number of input constraints arising from symbolic minimization
ict_d : number of distinct input constraints arising from symbolic minimization
t_ict : time to generate input constraints with Espresso

l_pd_e : number of PDs in the framework local part with elimination
t_l_pd_e : time to build local part of the framework with elimination
l_pd_ne : number of PDs in the framework local part without elimination
t_l_pd_e : time to build local part of the framework without elimination
elim : "yes", if elimination produced smaller area, otherwise "no"

pds_it : number of iterations of encoding generation to find local minimum
cl_i : encoding length after initial iteration
a_cl_f : encoding length after final iteration
t_pds : time needed to generate encoding satisfying the PD framework

t_min : time to minimize the encoded FSM combinational part
a_pt : number of product terms in the minimized combinational part

a_area : area estimate of the minimized combinational part for the encoded FSM
a_time : time to execute the ASSTUCE run

Table 14.4: Raw data for the ASSTUCE run on the I-group of FSM benchmarks

FSM	t_cpt	ict_t	ict_d	t_ict	lpd_e	t_lpd_e	lpd_ne	t_lpd_ne	elim	pds_it	cl_i	a_cl_f	t_pds	t_min	a_pt	a_area	a_time
lion	0.00	9	6	0.02	6	0.00	10	0.00	yes	3	2	2	0.05	0.00	7	77	0.10
train4	0.00	10	6	0.02	4	0.02	10	0.00	yes	3	2	2	0.02	0.00	6	66	0.05
dk15	0.02	17	10	0.03	8	0.00	17	0.00	no	3	4	4	0.08	0.03	16	368	0.23
mc	0.02	10	4	0.02	6	0.00	6	0.00	yes	3	2	2	0.03	0.00	8	136	0.07
tav	0.00	12	5	0.03	6	0.00	6	0.02	no	3	2	2	0.03	0.03	11	198	0.15
bbtas	0.00	16	7	0.02	9	0.02	18	0.00	yes	4	3	3	0.10	0.02	10	150	0.18
dk27	0.00	10	9	0.02	21	0.02	39	0.02	no	7	4	4	0.38	0.02	8	128	0.47
dk14	0.00	25	12	0.08	27	0.05	54	0.02	yes	5	6	6	0.30	0.07	23	667	0.60
shiftreg	0.00	9	5	0.03	28	0.02	28	0.00	yes	5	4	4	0.23	0.02	5	75	0.33
dk17	0.02	20	14	0.02	31	0.10	67	0.02	no	6	4	4	0.58	0.05	18	342	0.72
ex6	0.02	23	16	0.05	22	0.10	74	0.02	no	3	5	5	0.37	0.10	23	759	0.62
s386	0.05	32	18	0.10	51	0.20	116	0.05	yes	4	7	7	0.53	0.10	26	1092	1.15
ex4	0.03	21	14	0.02	91	0.10	91	0.05	yes	5	4	4	0.65	0.05	19	627	0.88
dk512	0.03	21	19	0.03	106	0.30	207	0.07	no	3	6	6	1.03	0.03	19	437	1.22
cse	0.05	57	24	0.17	59	0.47	232	0.08	yes	3	7	7	0.60	0.17	52	2184	1.67
kirkman	0.15	58	23	0.62	50	0.27	168	0.08	yes	5	6	6	0.85	0.62	52	2496	3.20
s208	0.10	25	23	0.23	81	0.42	210	0.12	no	3	7	7	1.37	0.27	22	990	2.42
s420	0.08	25	23	0.37	81	0.50	210	0.08	no	3	7	7	1.43	0.43	22	1342	2.82
keyb	0.08	78	35	0.20	79	1.45	402	0.13	yes	3	9	9	1.20	0.23	68	2924	3.65
s1	0.05	92	25	0.23	131	0.52	279	0.17	no	4	5	5	1.80	0.18	90	3330	2.97
pma	0.07	43	31	0.45	252	1.48	521	0.22	yes	3	10	10	2.93	0.13	41	2214	5.28
s820	0.10	89	35	1.52	176	1.33	506	0.23	yes	8	7	7	4.67	1.10	71	5396	10.23
s832	0.13	89	35	1.37	176	1.27	506	0.22	no	5	8	7	4.72	1.12	68	5168	9.33
dk16	0.10	55	35	0.12	374	2.32	805	0.27	yes	3	12	12	4.35	0.07	54	2322	7.10
styr	0.18	111	46	1.68	193	3.67	854	0.48	no	4	9	8	8.18	0.35	99	5148	11.32
sand	1.00	114	37	1.25	363	1.90	643	0.37	yes	4	6	6	3.25	0.52	105	5145	9.02
s510	0.57	75	47	0.53	1081	3.45	1081	0.83	no	3	6	6	8.20	0.25	68	4284	10.97
s1494	1.20	121	67	7.95	766	20.25	2088	1.30	yes	7	18	18	81.25	1.20	117	10413	116.63
s1488	0.75	121	67	7.70	850	22.35	2095	1.23	no	6	19	17	81.52	1.12	118	10148	94.12

t_cpt : time needed to generate all non-trivial compatible pairs

ict_t : total number of input constraints arising from symbolic minimization
ict_d : number of distinct input constraints arising from symbolic minimization
t_ict : time to generate input constraints with Espresso

l_pd_e : number of PDs in the framework local part with elimination
t_l_pd_e : time to build local part of the framework with elimination
l_pd_ne : number of PDs in the framework local part without elimination
t_l_pd_e : time to build local part of the framework without elimination
elim : "yes", if elimination produced smaller area, otherwise "no"

pds_it : number of iterations of encoding generation to find local minimum
cl_i : encoding length after initial iteration
a_cl_f : encoding length after final iteration
t_pds : time needed to generate encoding satisfying the PD framework

t_min : time to minimize the encoded FSM combinational part
a_pt : number of product terms in the minimized combinational part

a_area : area estimate of the minimized combinational part for the encoded FSM
a_time : time to execute the ASSTUCE run

complexity has been computed in Section 12.3.2 as bounded by $\mathcal{O}(n^2 + c)$. In this bound, n is the number of symbols (i.e. states of the FSM) and c is the number of non-don't care components in the initial PD matrix P, which we know to be bounded by the product of the number of symbols by the cardinality of the initial set of PDs.

We have compared the expected upper bound with the measured values. The result is depicted in the chart of Figure 14.2. The horizontal axis of the Figure corresponds to the benchmark FSMs ordered by size. The curve representing the upper bound was obtained from each machine using the formula

$$(\mathbf{st}^2 + \mathbf{st}.\mathbf{lpd_ne})/250,$$

which approximates the predicted bound. The divisor 250 is the constant experimentally found that normalizes the formula with regard to the time unit.

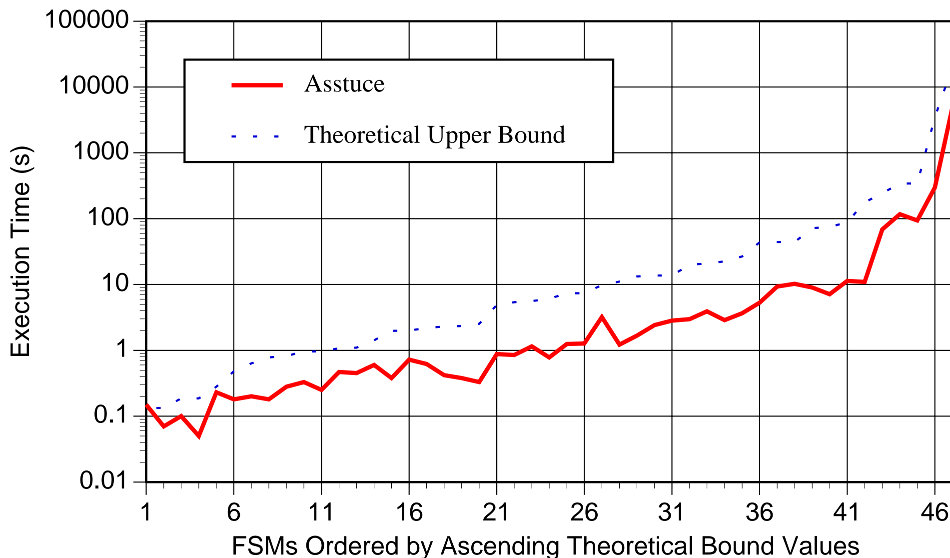


Figure 14.2: Theoretical bound versus measured execution time for ASSTUCE

In the chart the curve representing the measured time closely follows the predicted upper bound.

As a qualitative remark, we consider that the MCNC benchmarks are not very well adapted to measure the behavior of programs considering state minimization issues, since most of its machines have either no or only some compatible states, with a few exceptions.

14.2.5 ASSTUCE versus Complete Encoding Serial Strategy

The raw data derived from the comparison of ASSTUCE with the complete encoding serial strategy based on the STAMINA and DIET programs is depicted in Table 14.5 for the C-group of machines and in Table 14.6, for the I-group.

Columns are grouped by comparison parameter, and there are four of these groups in each Table. Two comparison parameters, the number of transistors in the final PLA and the PLA

Table 14.5: ASSTUCE versus complete encoding strategy for the C-group of FSMs

FSM	a_area	d_area	sd_area	a_pt	d_pt	sd_pt	a_cl_f	d_cl_f	sd_cl_f	a_time	d_time	sd_time
s27	198	234	234	11	13	13	3	3	3	0.28	0.10	0.10
beecount	190	264	171	10	12	9	3	4	3	0.45	0.20	0.10
lion9	140	153	84	7	9	6	5	4	3	0.38	0.40	0.10
ex5	162	414	132	9	23	11	4	4	2	0.18	0.40	0.10
ex7	162	414	120	9	23	10	4	4	2	0.25	0.60	0.00
ex3	144	414	144	8	23	12	4	4	2	0.20	0.50	0.10
bbara	600	600	418	24	24	19	5	5	4	0.42	0.50	0.20
opus	532	448	448	19	16	16	4	4	4	0.38	0.40	0.30
train11	85	280	110	5	14	10	4	5	2	0.33	1.00	0.00
mark1	738	738	697	18	18	17	5	5	5	0.78	1.50	1.00
sse	1053	1092	1092	27	28	28	6	6	6	1.27	1.50	1.40
bbsse	1053	1092	1092	27	28	28	6	6	6	1.25	1.50	1.40
ex2	819	1080	288	21	40	16	11	7	4	0.85	7.40	1.10
tma	1598	1271	1312	34	31	32	9	7	7	2.87	8.30	5.10
ex1	2562	2262	2146	42	39	37	8	7	7	3.92	8.00	5.60
tbk	7896	*f*	3591	94	*f*	63	23	*f*	14	68.82	*f*	15.1
scf	18221	*f*	*f*	133	*f*	*f*	9	*f*	*f*	298.17	*f*	*f*
s298	19800	*f*	*f*	300	*f*	*f*	18	*f*	*f*	4828.00	*f*	*f*

Prefixes:
a_ : results obtained by running ASSTUCE
d_ : results obtained by running DIET alone
sd_ : results obtained by running STAMINA followed by DIET

Suffixes:
area : area estimate of the minimized combinational part for the encoded FSM
pt : number of product terms in the encoded FSM minimized combinational part
cl_f : encoding length
time : total execution time

Obs: the appearance of the symbol *f* in some entry of the table indicates that the program failed to process the associated FSM

Table 14.6: ASSTUCE versus complete encoding strategy for the I-group of FSMs

FSM	a_area	d_area	a_pt	d_pt	a_cl_f	d_cl_f	a_time	d_time
lion	77	99	7	9	2	2	0.10	0.10
train4	66	99	6	9	2	2	0.05	0.10
dk15	368	391	16	17	4	4	0.23	0.20
mc	136	153	8	9	2	2	0.07	0.00
tav	198	198	11	11	2	2	0.15	0.00
bbtas	150	195	10	13	3	3	0.18	0.10
dk27	128	144	8	9	4	4	0.47	0.20
dk14	667	529	23	23	6	4	0.60	0.30
shiftreg	75	72	5	6	4	3	0.33	0.20
dk17	342	323	18	17	4	4	0.72	0.40
ex6	759	759	23	23	5	5	0.62	0.40
s386	1092	1092	26	28	7	6	1.15	1.40
ex4	627	594	19	18	4	4	0.88	1.50
dk512	437	360	19	18	6	5	1.22	2.40
cse	2184	1620	52	45	7	5	1.67	2.60
kirkman	2496	2544	52	53	6	6	3.20	4.80
s208	990	882	22	21	7	6	2.42	3.50
s420	1342	1218	22	21	7	6	2.82	3.60
keyb	2924	2886	68	78	9	7	3.65	7.90
s1	3330	2738	90	74	5	5	2.97	5.30
pma	2214	1968	41	41	10	8	5.28	24.40
s820	5396	3942	71	54	7	6	10.23	8.10
s832	5168	3953	68	59	7	6	9.33	9.60
dk16	2322	1643	54	53	12	8	7.10	51.20
styr	5148	4900	99	100	8	7	11.32	19.00
sand	5145	4949	105	101	6	6	9.02	32.70
s510	4284	*f*	68	*f*	6	*f*	10.97	*f*
s1494	10413	*f*	117	*f*	18	*f*	116.63	*f*
s1488	10148	*f*	118	*f*	17	*f*	94.12	*f*

Prefixes:
a_ : results obtained by running ASSTUCE
d_ : results obtained by running DIET alone

Suffixes:
area : area estimate of the minimized combinational part for the encoded FSM
pt : number of product terms in the encoded FSM minimized combinational part
cl_f : encoding length
time : total execution time

Obs: the appearance of the symbol *f* in some entry of the table indicates that the program failed to process the associated FSM

sparsity could not be easily computed from the output format of the DIET program. Accordingly, we will only use such parameters during the discussion of the partial encoding strategies comparison.

There is a difference between column groups in Table 14.5 and Table 14.6, namely the C-group Table has three columns per parameter, while the I-group Table has only two columns per parameter. This happens because we consider a third value as a reference for the C-group comparisons, which is the result of running the state assignment program alone, without the previous state minimization step with STAMINA[1]. Thus, we may better evaluate the influence of state minimization in the encoding results. For the I-group, the columns corresponding to the serial strategy are identical to the columns associated to the state assignment run alone, except for the time taken to find that there are no compatible pairs in the machine other than the trivial ones. We accordingly use the pure state assignment columns, even though this gives a little unfair time advantage to the serial strategy with regard to ASSTUCE, which always makes a compatibility analysis. This choice is maintained throughout the rest of this Chapter.

We now consider each of the compared parameters in some detail. Figures 14.3 and 14.4 are plots of the area values for the C-group and the I-group of FSMs, respectively. The order of the machines is the same as the one used in the raw data Tables, i.e. machines are plot in ascending order of the number of states in the original description. This will be true in all subsequent plots in this Chapter.

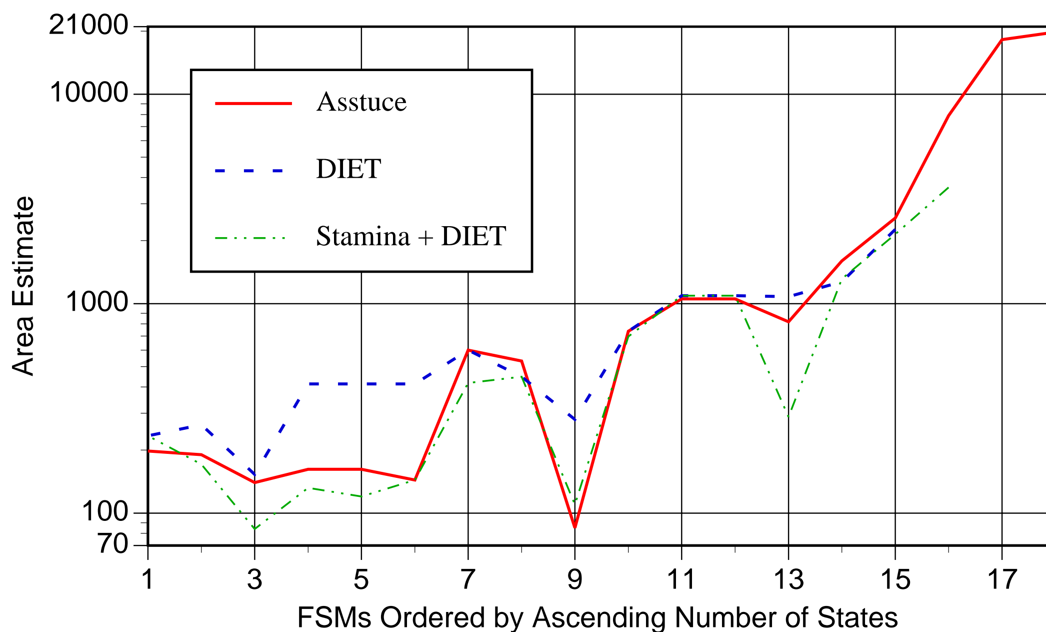


Figure 14.3: ASSTUCE versus complete encoding - area - C-group

We note in Figure 14.3 that the line depicting the ASSTUCE results is mostly below the pure state assignment results and above the serial strategy. In some cases, like machines 1 and 9 (corresponding to machines **s27** and **train11** in the Tables), ASSTUCE obtains the best result, but some machines pose a problem when ASSTUCE forces the satisfaction of all constraints in it. This is the case of machine 13, i.e. **tbk**, which has too many constraints. Concerning

[1]Recall that ASSTUCE *is* a state assignment method.

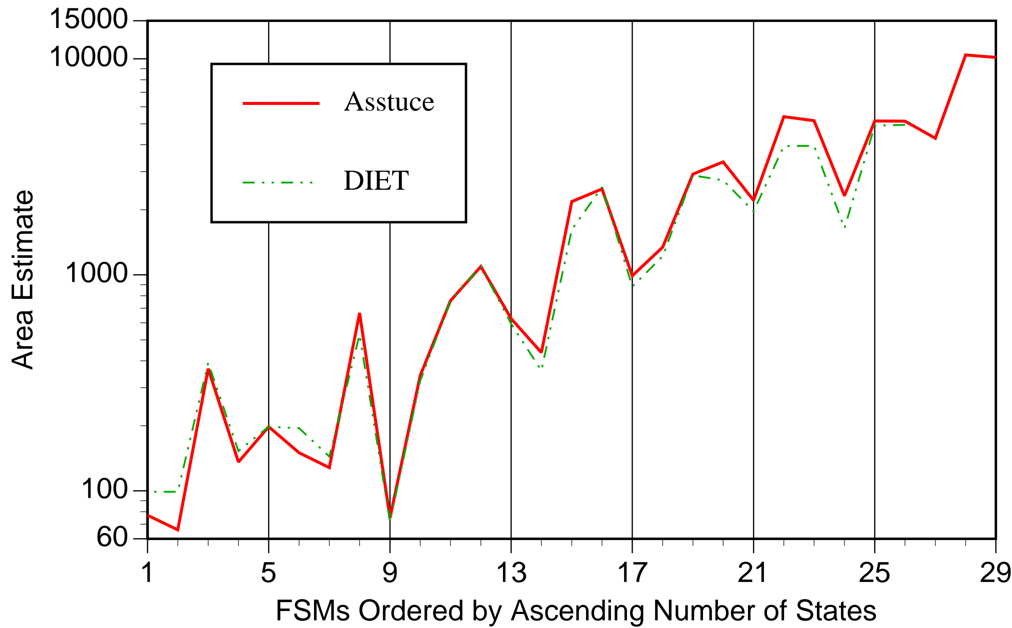


Figure 14.4: ASSTUCE versus complete encoding - area - I-group

the I-group area results, we see that both the serial strategy and ASSTUCE give very similar results, with ASSTUCE performing slightly better for smaller machines, and the serial strategy performing better for FSMs of intermediate size. In both groups, DIET failed to encode the biggest machines.

Figures 14.5 and 14.6 compare graphically the product term counts for the C-group and the I-group of FSMs, respectively.

The situation for the product terms is a little different from that for the area parameter. ASSTUCE performs better, obtaining the best absolute results in a good percentage of the machines in the C-group, and increasing the best result counts for the I-group. This indicates that the weak point in the complete constrained encoding in ASSTUCE is really the generated encoding length (big area with few product terms implies large code length). The PD covering formulation implemented by DIET often obtains shorter codes.

The complexity of the DIET tool algorithms are evident from the execution time plots of Figures 14.7 and 14.8.

These plots clearly show that, as the size of the FSMs increases, the execution time of ASSTUCE becomes comparable to and next smaller than the time taken by the serial strategy. Even if state minimization can cause a dramatical reduction on the execution time taken by DIET to perform state assignment, this has marked effects only for smaller machines. For the I-group, the increasing size of machines (from machine **sse** on) causes a steadily growing difference between the execution times of ASSTUCE and DIET.

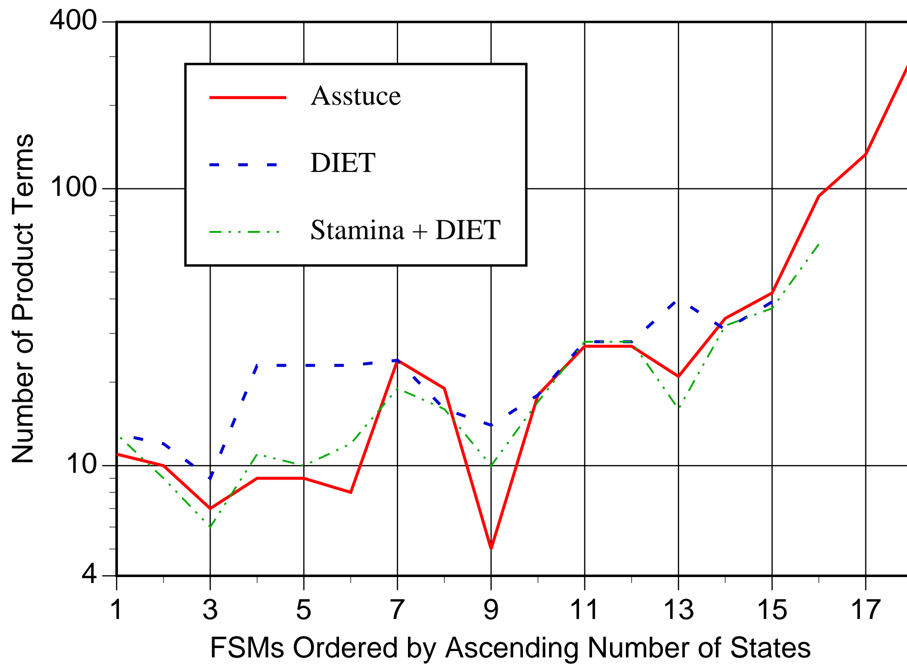


Figure 14.5: ASSTUCE versus complete encoding - product terms - C-group

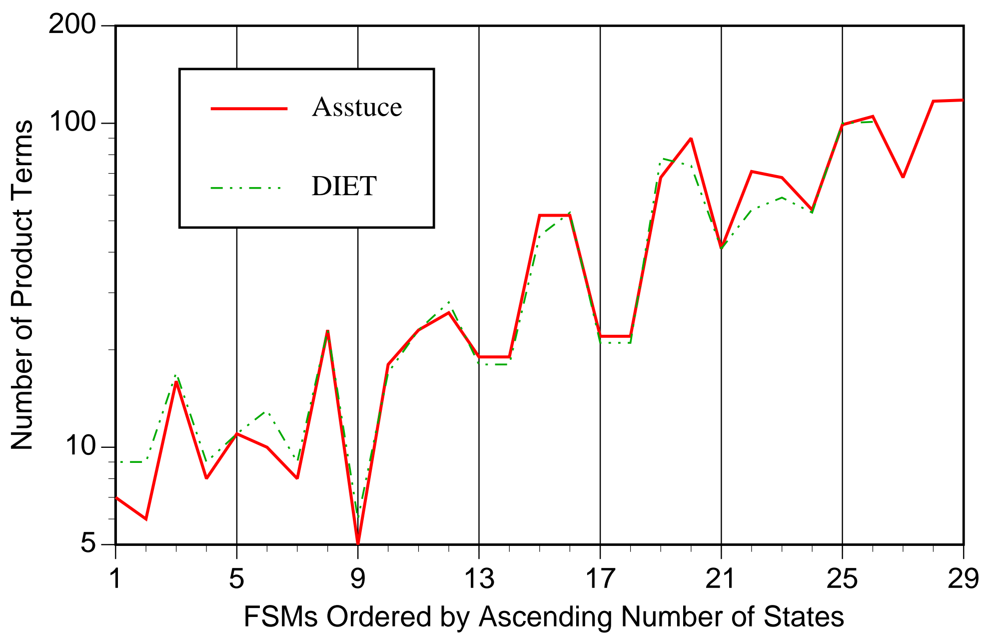


Figure 14.6: ASSTUCE versus complete encoding - product terms - I-group

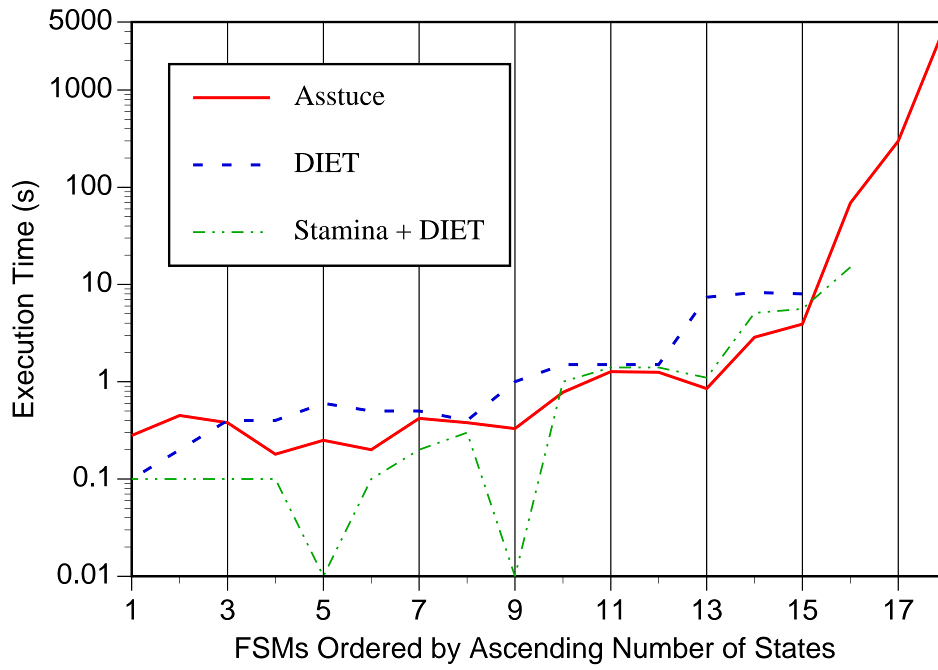


Figure 14.7: ASSTUCE versus complete encoding - execution time - C-group

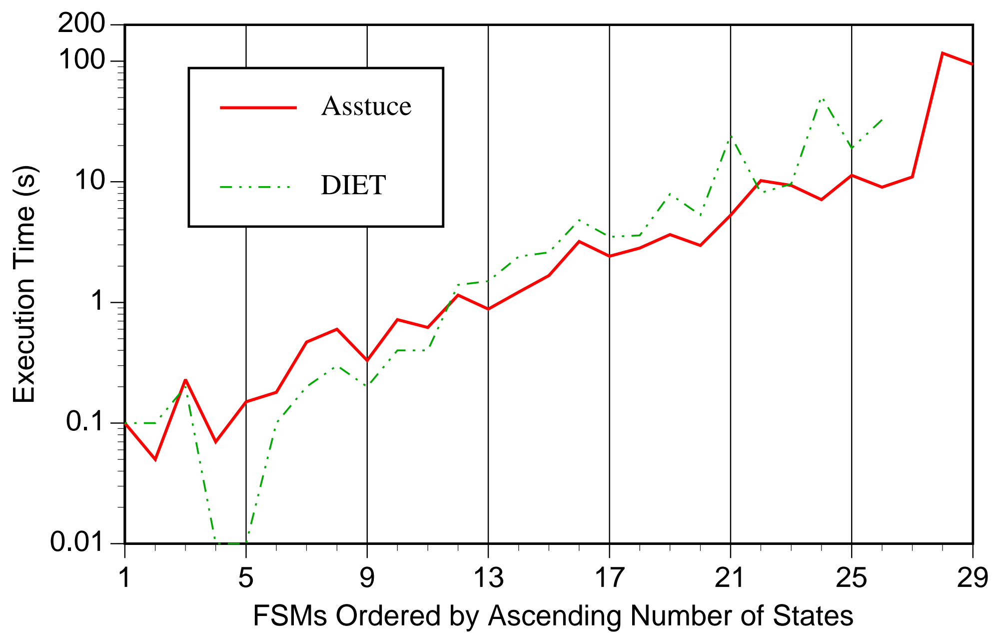


Figure 14.8: ASSTUCE versus complete encoding - execution time - I-group

14.2.5.1 ASSTUCE versus Complete Serial Strategy - Discussion

The main result of the comparison of ASSTUCE versus the complete encoding serial strategy based on DIET is that most results are rather close to each other, with a small advantage to the serial strategy, except on the execution time parameter. In fact, DIET gets shorter codes than ASSTUCE at the expense of an algorithm with higher complexity than our simple greedy encoding. This has the drawback of limiting the use of the serial strategy to intermediate size examples, even if the previous state minimization step can significantly reduce the processing needs of the original FSM. As an illustration, DIET alone failed to encode three out of eighteen machines in the C-group. After minimizing the state cardinality of these FSMs, only one of the three could then be encoded successfully.

The relative advantage obtained by ASSTUCE in the product term cardinality is a consequence of using non-functional, non-injective encodings. The generation of incompletely specified binary codes happens even if the original FSM contains only trivial compatible pair of states. In this way, cube merging is facilitated during the logic minimization step. Besides, the consideration of state minimization by ASSTUCE is guaranteed to respect the bounds predicted by symbolic minimization for the *original* FSM (cf. Corollary 9.1), not for the minimized one, which can be bigger than the original (cf. Example 6.5).

A last conclusion is that ASSTUCE does not control the encoding length as well as DIET. This is one bad consequence of using a heuristic technique with many degrees of freedom. However, ASSTUCE is in its first steps of refinement, and our approach can maybe be much enhanced. More important, the main weakness of ASSTUCE when parameterized to perform complete encoding is reduced when parameterizing the program to do partial encoding. Since complete encoding results are often inferior to those of partial encoding results, we consider more important to compare ASSTUCE with a partial encoding serial strategy.

14.2.6 ASSTUCE versus Partial Encoding Serial Strategy

The raw data that results from comparing ASSTUCE and the partial encoding serial strategy based on the STAMINA and NOVA programs is depicted in Table 14.7 for the C-group of machines and in Table 14.8, for the I-group.

Let us consider each of the compared parameters in some detail. Figures 14.9 and 14.10 are plots of the area values for the C-group and the I-group of FSMs, respectively.

We note in Figure 14.9 that again, the line depicting the ASSTUCE results is again between the pure state assignment results and above the serial strategy. For the I-group, in Figure 14.10, the area results are almost identical, with sometimes ASSTUCE obtaining the best absolute result, sometimes the serial strategy being best.

Figures 14.11 and 14.12 compare graphically the product term counts for the C-group and the I-group of FSMs, respectively.

The situation for the product terms is a quite different from that for the area parameter. ASSTUCE performs almost always better than both NOVA alone and STAMINA followed by NOVA, and this for both C-group and I-group machines. This indicates again that ASSTUCE necessarily computes a code length that is greater, in general, than the serial strategy. But the area estimate is not the only important parameter in this comparison.

Table 14.7: ASSTUCE versus partial encoding serial strategy for the C-group of FSMs

FSM	a_area	n_area	sn_area	a_time	n_time	sn_time	a_pt	n_pt	sn_pt	a_cl_f	n_cl_f	sn_cl_f	a_tr	n_tr	sn_tr	a_spty	n_spty	sn_spty
s27	198	234	216	0.28	0.10	0.10	11	13	12	3	3	3	43	62	51	78.28	73.5	76.39
beecount	144	247	160	0.23	0.10	0.10	9	13	10	2	3	2	50	68	54	65.28	72.47	66.25
lion9	102	136	77	0.43	0.30	0.10	6	8	7	4	4	2	24	35	24	76.47	74.26	68.83
ex5	120	252	96	0.25	0.50	0.00	8	14	8	3	4	2	34	88	34	71.67	65.08	64.58
ex7	120	306	96	0.27	0.30	0.10	8	17	8	3	4	2	38	107	36	68.33	65.03	62.5
ex3	144	324	96	0.18	0.20	0.10	8	18	8	4	4	2	33	103	35	77.08	68.21	63.54
bbara	380	550	380	0.78	0.20	0.20	20	25	20	3	4	3	94	129	94	75.26	76.55	75.26
opus	504	448	448	1.05	0.20	0.10	18	16	16	4	4	4	139	128	123	72.42	71.43	72.54
train11	85	153	66	0.37	0.60	0.10	5	9	6	4	4	2	26	47	24	69.41	69.28	63.64
mark1	697	684	646	0.93	5.10	4.30	17	18	17	5	4	4	145	115	117	79.20	83.19	81.73
sse	972	990	1023	1.67	0.50	1.10	27	30	31	5	4	4	196	191	206	79.84	80.71	79.86
bbsse	972	990	1023	1.62	0.40	1.20	27	30	31	5	4	4	196	191	206	79.84	80.71	79.86
ex2	594	609	195	1.28	0.50	1.00	18	29	13	9	5	3	91	171	66	84.68	71.92	66.15
tma	1230	1155	1295	5.97	6.60	13.50	30	33	37	7	5	5	241	230	257	80.41	80.09	80.15
ex1	2288	2496	2132	5.37	6.50	5.00	44	48	41	5	5	5	399	422	330	82.56	83.09	84.52
tbk	1680	4620	1431	103.22	140.7	24.5	56	154	53	5	5	4	614	1423	553	63.45	69.2	61.36
scf	17420	18471	16244	603.68	105.8	59.8	130	141	124	8	7	7	1541	1469	1383	91.15	92.04	91.49
s298	16632	22464	10332	10637.90	828.8	266.6	308	624	287	14	8	8	3824	6783	2586	77.01	69.81	74.96

Prefixes:
a_ : results obtained by running ASSTUCE
n_ : results obtained by running NOVA alone
sn_ : results obtained by running STAMINA followed by NOVA

Suffixes:
area : area estimate of the minimized combinational part for the encoded FSM
pt : number of product terms in the minimized combinational part of the encoded FSM
cl_f : encoding length
time : total execution time
tr : number of transistors in the minimized combinational part of the encoded FSM
spty : percentual sparsity of the minimized combinational part for the encoded FSM

Table 14.8: ASSTUCE versus partial encoding strategy for the I-group of FSMs

FSM	a_area	n_area	a_time	n_time	a_pt	n_pt	a_cl_f	n_cl_f	a_tr	n_tr	a_spty	n_spty
lion	66	66	0.08	0.10	6	6	2	2	24	24	63.64	63.64
train4	66	66	0.12	0.00	6	6	2	2	26	26	60.61	60.61
dk15	323	323	0.17	0.10	19	19	2	2	115	115	64.40	64.4
mc	136	153	0.07	0.00	8	9	2	2	40	49	70.59	67.97
tav	180	198	0.17	0.10	10	11	2	2	42	46	76.67	76.77
bbtas	150	135	0.20	0.00	10	9	3	3	42	40	72.00	70.37
dk27	91	117	0.42	0.10	7	9	3	3	31	46	65.93	60.68
dk14	506	580	0.50	0.30	22	29	4	3	138	182	72.73	68.62
shiftreg	48	48	0.28	0.10	4	4	3	3	8	8	83.33	83.33
dk17	320	304	0.40	0.10	20	19	3	3	101	104	68.44	65.79
ex6	690	675	0.95	0.20	23	25	4	3	173	211	74.92	68.74
s386	1056	1056	2.37	0.30	32	32	4	4	218	220	79.36	79.17
ex4	627	627	1.92	0.10	19	19	4	4	128	145	79.59	76.87
dk512	360	306	4.65	4.10	18	18	5	4	92	96	74.44	68.63
cse	1656	1518	1.97	2.30	46	46	5	4	407	392	75.42	74.18
kirkman	2475	2745	3.20	11.10	55	61	5	5	310	344	87.47	87.47
s208	882	975	2.67	3.60	21	25	6	5	112	154	87.30	84.2
s420	1218	1375	2.78	3.80	21	25	6	5	112	154	90.80	88.8
keyb	1519	1488	7.32	4.10	49	48	5	5	524	488	65.50	67.2
s1	3071	2960	2.53	1.20	83	80	5	5	730	741	76.23	74.97
pma	1848	1755	12.52	18.70	44	45	6	5	301	376	83.71	78.58
s820	5016	5320	9.70	3.40	66	76	7	5	511	577	89.81	89.15
s832	5037	5040	11.22	3.00	69	72	6	5	541	547	89.30	89.15
dk16	1650	1298	7.06	29.20	66	59	6	5	466	392	71.76	69.8
styr	4462	4042	14.18	11.70	97	94	6	5	966	1007	78.35	75.09
sand	4949	4646	12.92	6.60	105	101	6	5	999	1028	79.81	77.87
s510	3843	4284	13.15	1.4	61	68	6	6	426	582	88.91	86.41
s1494	7788	7367	94.03	66.8	132	139	8	6	1166	1254	85.03	82.98
s1488	7788	7049	172.57	70.7	132	133	8	6	1138	1123	85.39	84.08

Prefixes:
a_ : results obtained by running ASSTUCE
n_ : results obtained by running NOVA alone

Suffixes:
area : area estimate of the minimized combinational part for the encoded FSM
pt : number of product terms in the minimized combinational part of the encoded FSM
cl_f : encoding length
time : total execution time
tr : number of transistors in the minimized combinational part of the encoded FSM
spty : percentual sparsity of the minimized combinational part for the encoded FSM

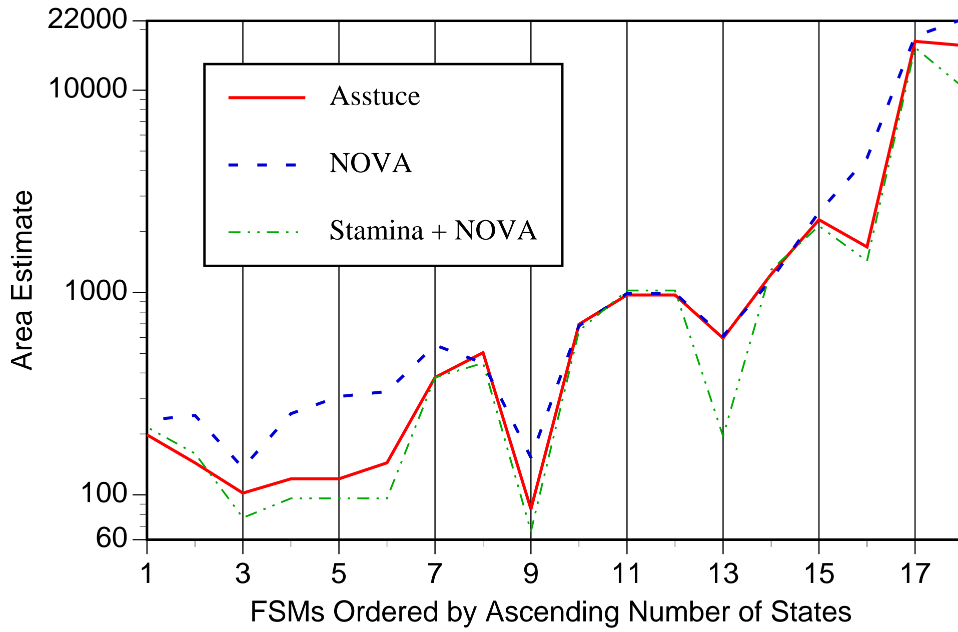


Figure 14.9: ASSTUCE versus partial encoding - area - C-group

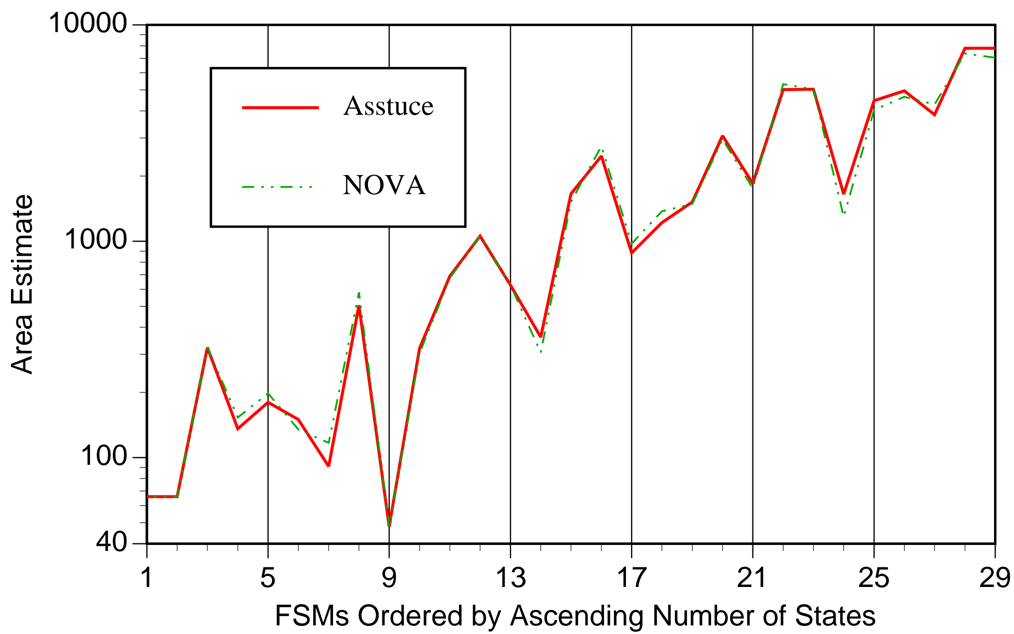


Figure 14.10: ASSTUCE versus partial encoding - area - I-group

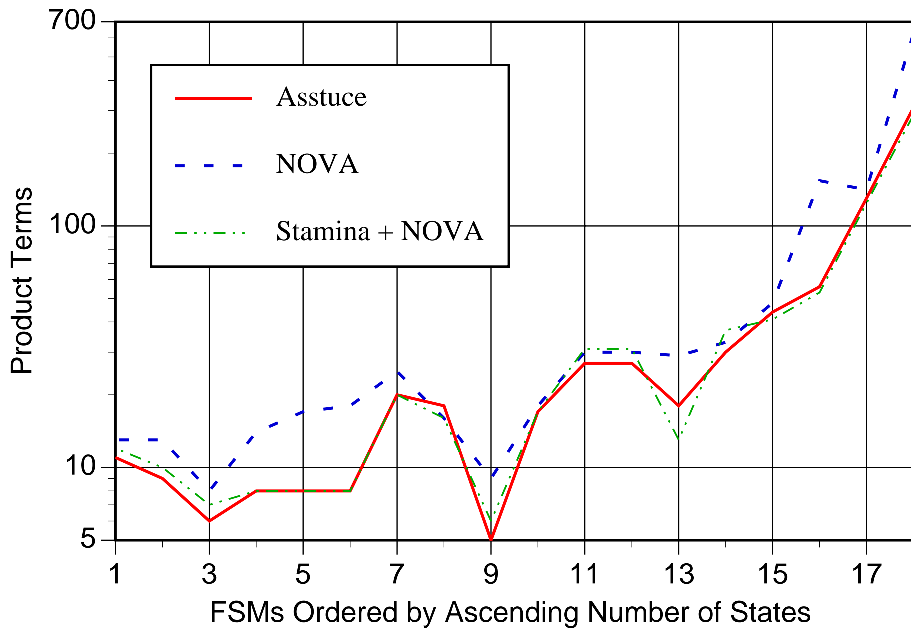


Figure 14.11: ASSTUCE versus partial encoding - product terms - C-group

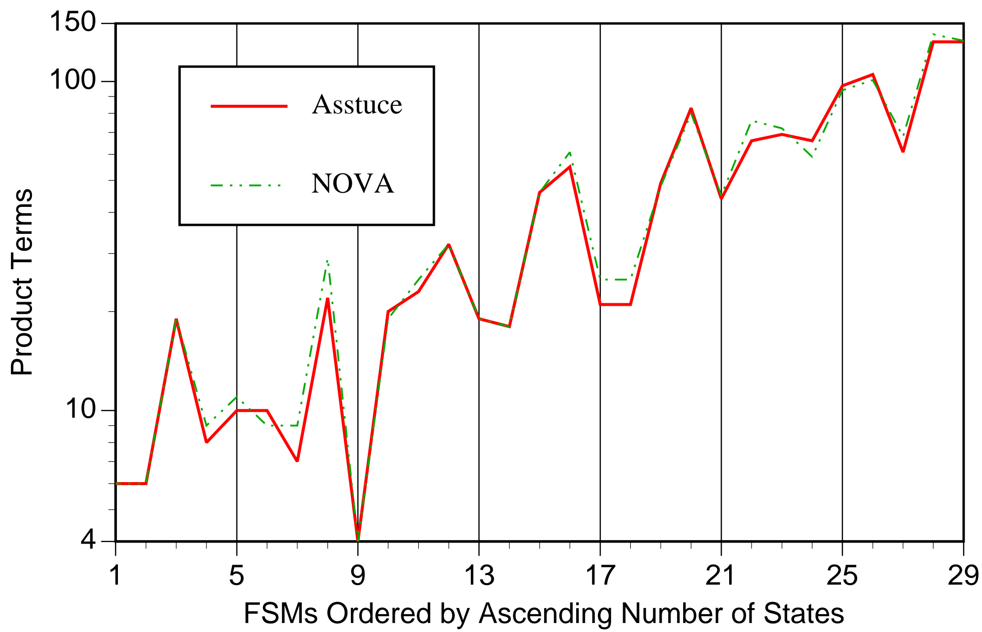


Figure 14.12: ASSTUCE versus partial encoding - product terms - I-group

Consider the final PLA sparsity parameter comparison showed in Figures 14.13 and 14.14.

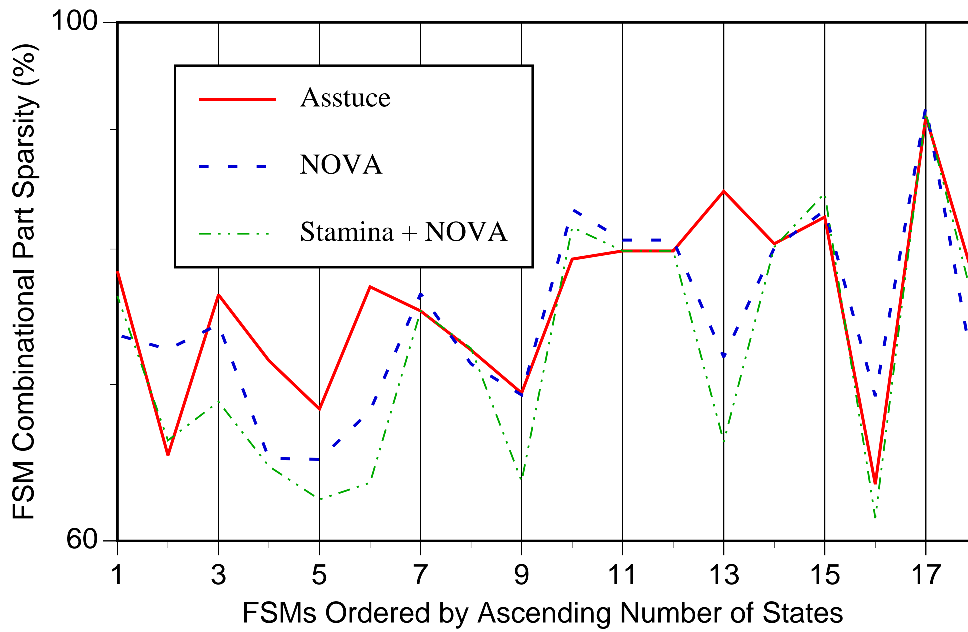


Figure 14.13: ASSTUCE versus partial encoding - sparsity - C-group

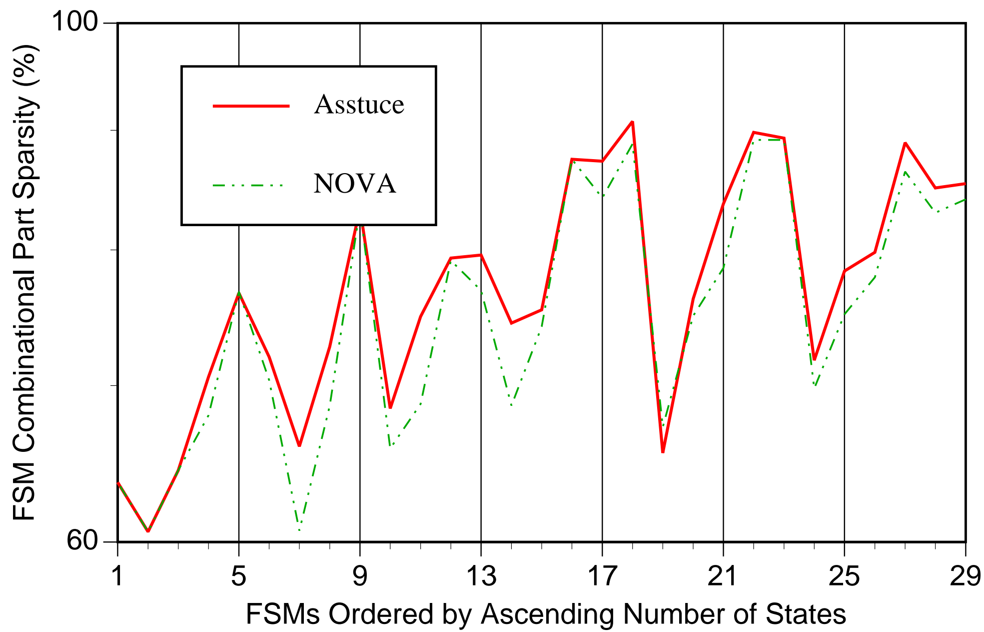


Figure 14.14: ASSTUCE versus partial encoding - sparsity - I-group

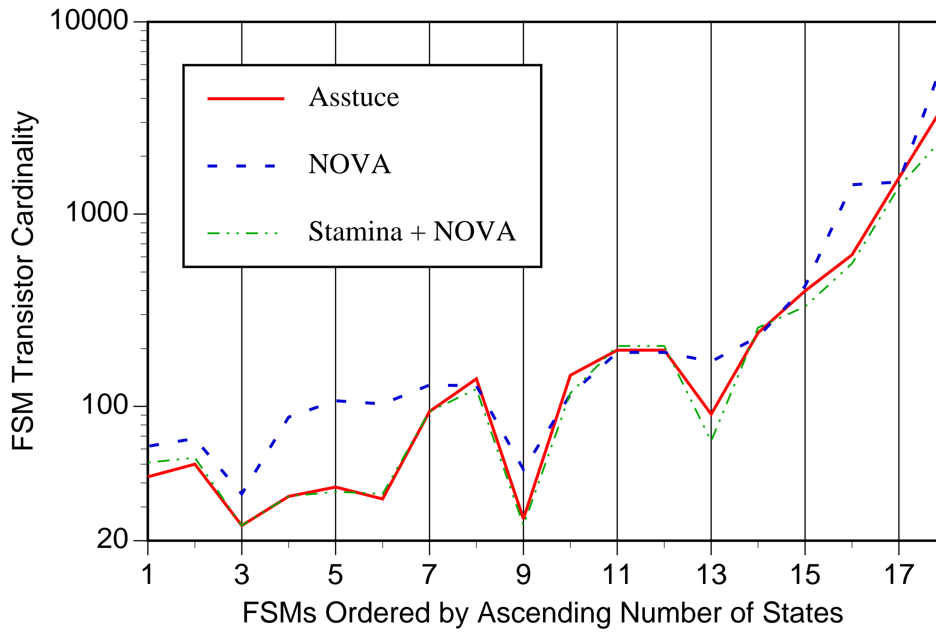


Figure 14.15: ASSTUCE versus partial encoding - transistor cardinality - C-group

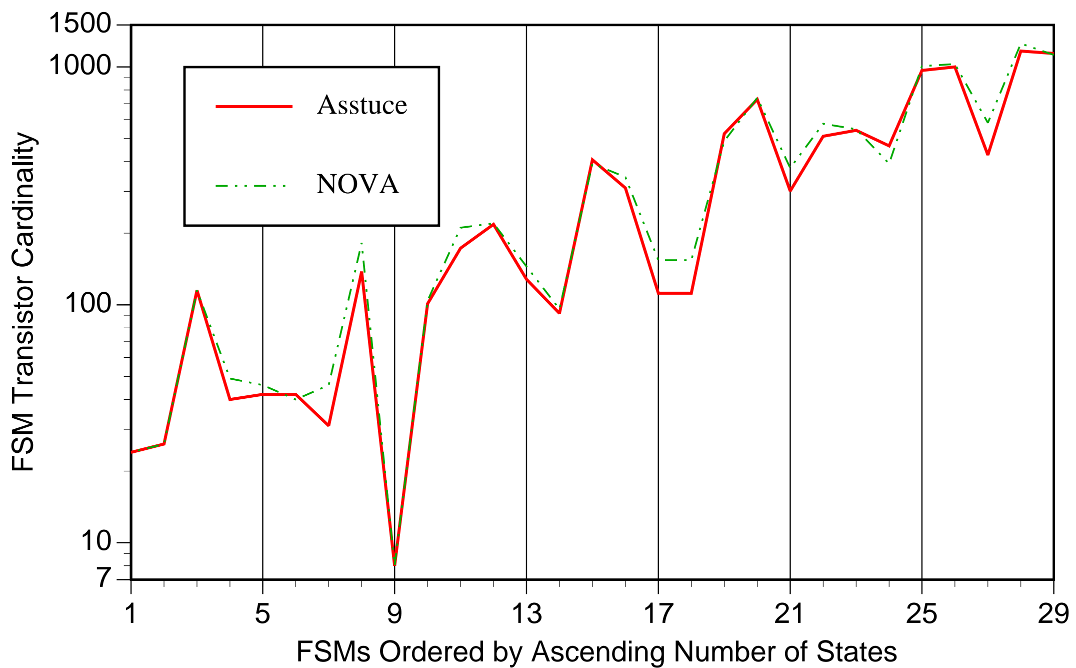


Figure 14.16: ASSTUCE versus partial encoding - transistor cardinality - I-group

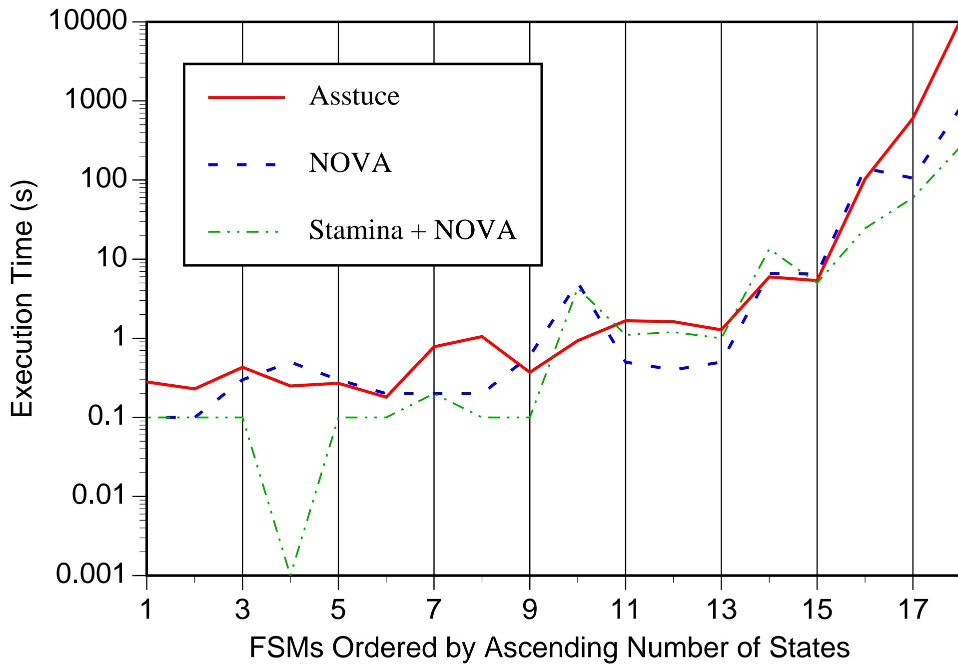


Figure 14.17: ASSTUCE versus partial encoding - time - C-group

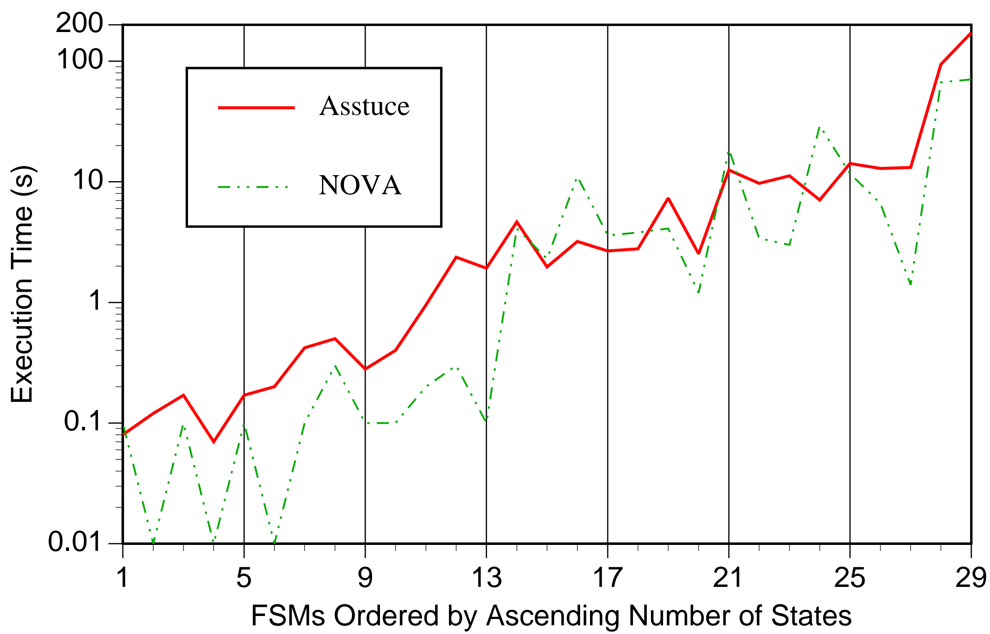


Figure 14.18: ASSTUCE versus partial encoding - time - I-group

These plots clearly show that ASSTUCE trades-off an increase in area against enhanced sparsity in the final PLA. An unusual situation arises here, since we remark that in more than 40% of the benchmark tests, ASSTUCE presents the greater sparsity without having the greatest PLA area (9 out of 18 machines in the C-group and 10 out of 29 in the I-group). To add interest to these observations, let us take into account the transistor cardinality comparison plots in Figures 14.15 and 14.16.

Here we see that ASSTUCE generates a PLA with approximately the same number of transistors as the serial strategy with NOVA for most machines in the C-group. For the I-group, ASSTUCE obtains transistor counts which are most often smaller than that obtained by NOVA.

Finally, we present the execution time comparison in Figures 14.17 and 14.18.

For both, the C-group and the I-group, the serial strategy gives better time results than ASSTUCE for most smaller machines, but the relationship becomes more complex for bigger FSMs, the best execution time being obtained by either ASSTUCE or NOVA.

14.2.6.1 ASSTUCE versus Partial Serial Strategy - Discussion

ASSTUCE and the partial encoding serial strategy based on NOVA are comparable for most parameters, with the serial strategy obtaining slightly better area results and ASSTUCE obtaining slightly sparser machines but with reduced number of transistors in it, and less product terms. The consequences of these differences is that we judge the ASSTUCE results more adapted to consider power dissipation issues in big PLAs, because of the combined effect of smaller areas corresponding to sparser PLAs. Besides, we know that sparser PLAs favor the use of topological optimization tools during the low level synthesis of the FSM.

The advantages related to ASSTUCE are a consequence of using non-functional, non-injective encodings. Cube merging is favored during the logic minimization step, and even if the encoding length increases, the final result may combine smaller areas with less dissipated power.

14.2.7 Benchmark Tests - Conclusions

In both benchmark test sets complete and partial encoding, ASSTUCE performed significantly better than pure state assignment approaches for machines with high potential for state reduction, reflected by an important number of compatibility classes in the original description. This situation indicates that the ASSTUCE method is in effect capable of capturing state minimization characteristics in FSMs, even if only partially, in the present prototype version.

As a general conclusion, we state that the ASSTUCE program, in its initial development stage is already competitive with the best serial strategies we could find. The consideration of SM constraints during the assignment adds complexity to the problem, but is not a limitation, since even machines that are larger than would be feasible to implement with PLAs could be assigned by it, e.g. the **s298** FSM. The program is suited to the treatment of both machines with or without a considerable number of compatibility classes. We consider that the efficiency of the program can be considerably increased in future versions, by either enhancing the implementation[2] or by refining the techniques of constraint satisfaction.

[2]The present version resulted from an effort of just five man-months for design, implementation and test.

Chapter 15

Overall Conclusions and Future Work

Let us start by restating the questions posed in Section 2.2, after the discussion of the case study.

> Given an FSM symbolic specification, how can we assign codes to its states such that we achieve:
>
> 1. the least number of distinct codes, which allows for extensive state minimization, and reduces the final PLA size?
> 2. the most sparse codes, to permit the greatest possibility of minimization of the final combinational part, allowing for enhanced topological minimization, and reduced power consumption?
> 3. codes with the least length in bits, to obtain small PLAs, and to generate the least number of outputs for the combinational part of the machine?

This is the set of specific questions we have tried to answer in this work. Section 15.1 establishes how close we have got to answer each of them, while Section 15.2 depicts which further efforts need to be made to approach still existing problems for the elucidation of these questions.

15.1 Overall Conclusions

We provide an overview of the work done in this thesis in Figure 15.1.

In the theoretical part of this work, we accomplished the integration of the SM and SA problems. This integration occurred in three phases. The first phase was the analysis of both problems, from a more or less precise initial statement, producing a set of constraints, which describes thoroughly, but separately, each problem. A second phase consisted in modeling each constraint kind as a distinct binary relation, reducing the original sets of constraints to sets of elementary constraints. At the end of the second phase the problems were still separately described, but they were now in a form amenable to unified treatment. The last phase was the study of the relationships arising among the constraint kinds, based on their elementary form. This study led to a set of theoretical findings relating the constraints classes with one

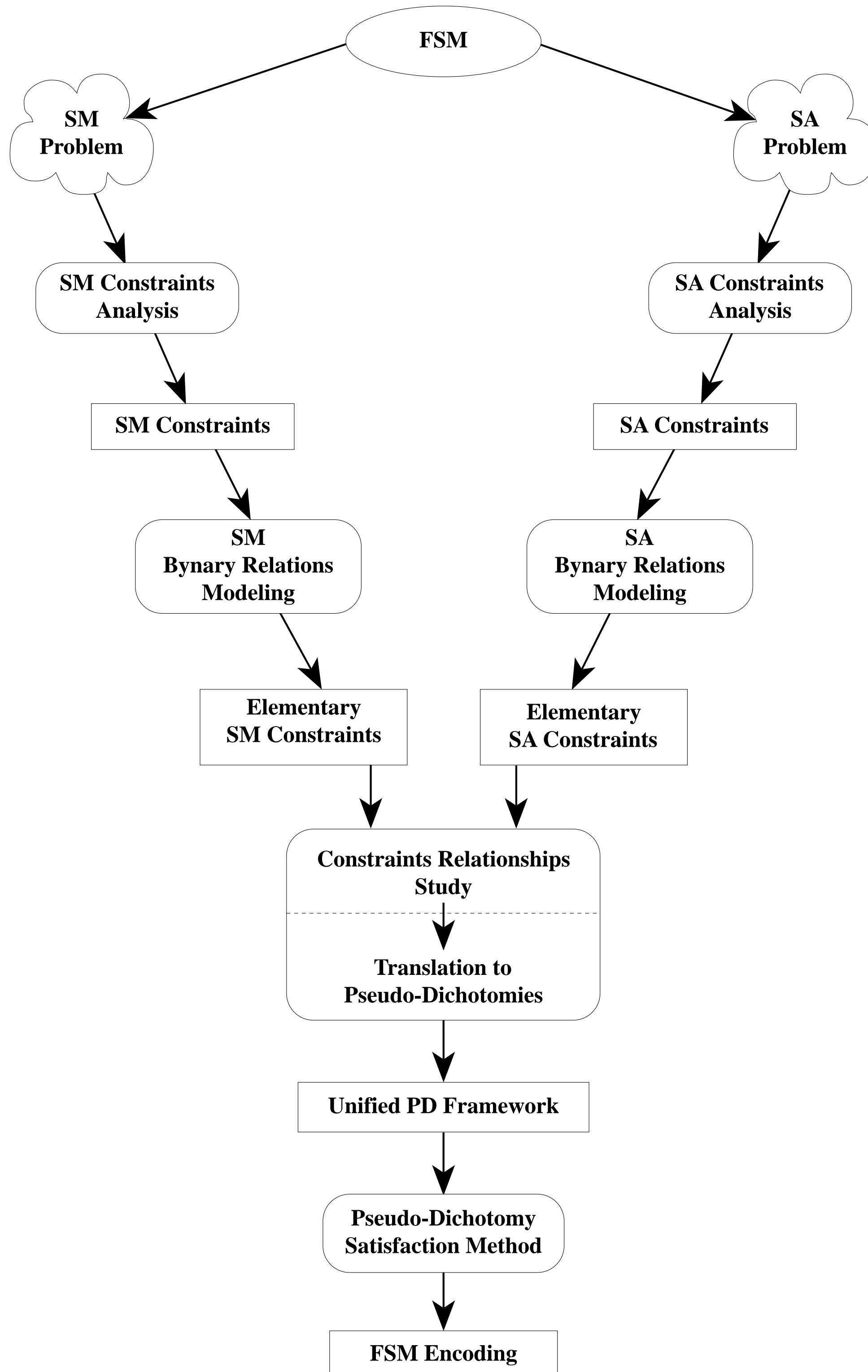


Figure 15.1: The ASSTUCE approach

another. The results found here are in no way complete, and the exploitation of the constraints relationship may still lead to unsuspected consequences, specially in the scope of the output constraints influence.

The resulting unification of the SM and SA problems occurred with the intervention of the pseudo-dichotomy concept, which provided an efficient way for uniformly representing the elementary constraints binary relations. The pseudo-dichotomy concept, after extended, served as the basis to the proposition of a unified framework adapted to solve approximations of the SM and SA problems.

We have generalized the statement of the Boolean constrained encoding problem, and we have congregated the original SM and SA problems into a single one, the two-level SM/SA problem. After this, we showed that the SM/SA problem can be reduced to the Boolean constrained encoding problem. Each of these is too complex to be treated exactly with existing techniques. Thus, the complete and partial approximations for each of them were proposed and shown to be easier to solve. The proposed framework supports techniques to tackle the solution of both approximations of the two-level SM/SA problem, which are also approximations to the optimal solution of both the original SM and SA problems.

Finally, we proposed the ASSTUCE method, based on our unified framework, to solve the above mentioned problem approximations. This method has been implemented as a computer program, and its results were extensively compared with existing serial strategies, revealing itself to be a promising approach. Despite the early development stage of the program ASSTUCE its results are already competitive with some well established assignment tools.

After analyzing the experimental results obtained with the ASSTUCE program, we are convinced that the refinement of the method implemented in it may render the use of the serial strategy dispensable for most practical cases of FSM assignment. The advantage of an approach like the one proposed here is twofold. First, it avoids the use of one tool in the design cycle of FSMs. More importantly, the simultaneous strategy seems more suited to find globally optimal implementations, because state minimization is carried out by the logic minimization tool, which is then allowed to employ less abstract cost functions to perform the merging of states. However, demonstrating this advantage still depends upon the results of the ASSTUCE method enhancement activities, presently under way.

In a recent work, Perkowski and Brown [92] mentioned that present circuit descriptions do not have as many don't care conditions as it would be possible to extract from the initial specification. They proposed a method to automatically generate don't cares from a high level description of the problem. The ASSTUCE method exploits the characteristics of problems defined at the logic level in order to do the same, i.e. to allow the generation of additional don't cares in the encoding of an FSM to permit greater freedom at lower abstractions levels, where more realistic cost functions can take advantage of these don't cares to further the quality of the final implementation.

Finally, of the three questions posed in the beginning of this Chapter, the ASSTUCE method allows that the answer to the first one be given by the logic minimization step. This happens because the present version of ASSTUCE generates an encoding where every compatible pair of states may be implemented as such or not by the logic minimizer. In this way, the first question can be best answered than in the case of a serial strategy, where less degrees of freedom are given to lower level abstraction tools. Instead of trying to minimize the number of states, the logic level tools are allowed to merge codes of states if this leads to an optimization of

less abstract parameters such as area of a two-level implementation, propagation delay time, dissipated power, literal count, etc.

As for the second question, within the ASSTUCE method, the encoding sparsity is traded off against code length, but since no compatible pair of states receive disjoint codes, the maximum desirable sparsity is attained. Thus, the second question has also received a convenient answer.

The third question is today the main point that makes ASSTUCE perform less well than it could. In fact, our experiments showed that the excessive code length that the present version of the method sometimes obtains can prevent that logic minimization be performed conveniently. A study on how to trade-off smaller code length while keeping the benefits of increased sparsity and low transistor count is a necessary step in the evolution of the ASSTUCE method.

15.2 Future Work

The immediate future of the ASSTUCE method has already been made clear in the last Section. It consists in rendering the prototype implementation more efficient by refining data structures and proposing new enhancements to the algorithm.

In the long term, however, many possibilities for further work were devised for the continuation of the research work started here. We will now discuss some of these possibilities.

The first possibility of future research results from the realization that the unified framework developed here is more general than previous similar propositions. These latter could already be applied to solve several distinct problems in VLSI design, and so does our framework. The task implied here is the modeling of the constraints describing several of these problems, followed by their formulation using PDs and their solution with the program ASSTUCE. One expected practical consequence of this work is the need for generalization of the ASSTUCE program structure to allow other problems' idiosyncrasies to be described.

What is still more interesting is to look for new applications where the unified framework proposed here can be applied, and which previous frameworks are prevented to deal with due to the limitations imposed by the use of functional and/or injective encodings. The two-level SM/SA problem has been the first one to take advantage of non-functional, non-injective encodings, to our knowledge.

A list of problems that have been treated using PDs (and which are thus amenable to be solved by our approach) is enumerated in Section 11.3 and comprises works about various logic and low-level VLSI design [42, 45, 26, 111, 114, 106, 44]. We have briefly studied these problems to verify their conformity with our PD framework. The result of this study was that the PD framework proposed here can be used to model all constraints considered in any of the previous works, with one exception, which is the work of Devadas et al in [44]. In this work, a state assignment method to implement fully testable sequential machines is proposed. In this method appears a kind of constraint where codes of states need to be at an exact *Hamming distance* of two from each other. The ***Hamming distance*** between two Boolean vectors is the number of positions in which the bits of the vectors differ. The PD framework can model constraints as long as they are required to be satisfied by one or by all codes. Thus, the present formulation of our framework cannot cope with this problem. However, the generalization needed to account for the constraint defined in [44] within our PD framework is quite simple. Let us associate a

counter to each PD in the local part. Whenever a PD is satisfied during the encoding column construction, its counter is decremented and tested. If its value is zero, the PD is discarded, since it has been satisfied "completely". In this way, a constraint may be "persistent" to any degree. In this scope, the persistence of a PD in the global part is conceptually infinite.

Another important extension of the present work is to devise the generality of the method with regard to the design style choices, i.e. to investigate the possibility of using the framework to tackle problems in encoding of sequential circuits implemented with architectures other than two-level, such as multiple-level logic implementations, programmable devices implementations, asynchronous implementations, etc. The general definition of dichotomy was proposed here to account for these future needs.

To cope with the problems discussed in the previous paragraphs, the general statement of the Boolean constrained encoding problem stated in Section 11.1 is fundamental, because we may start from it to look for reductions of problems to this formulation, and from there to a PD formulation of the problem's constraints.

The generalization of the approach proposed here must be followed by the generalization of the implementation as well. A first requirements specification for an exploratory environment is included in Appendix B. The ASSTUCE program was devised with this environment in mind. Further specification and implementation refinements are still required to set-up the general aspects of such an environment.

Bibliography

[1] Keumog Ahn and Sartaj Sahni. Constrained via minimization. *IEEE Transactions on Computer-Aided Design*, 12(2):273–282, February 1993.

[2] S. B. Akers Jr. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[3] R. Amann and U. G. Baitinger. Optimal state chains and state codes in finite state machines. *IEEE Transactions on Computer-Aided Design*, 8(2):153–170, February 1989.

[4] D. B. Armstrong. A programmed algorithm for assigning internal codes to sequential machines. *IRE Transactions on Electronic Computers*, EC-11:466–472, August 1962.

[5] P. Ashar, S. Devadas, and A. R. Newton. *Sequential Logic Synthesis*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic, Norwell, MA, 1992.

[6] M. J. Avedillo. *Una aproximación al diseño óptimo de máquinas de estados finitos*. PhD thesis, Universidad de Sevilla, Facultad de Física, Sevilla, Spain, 1992. (In Spanish).

[7] M. J. Avedillo, J. M. Quintana, and J. L. Huertas. State reduction of incompletely specified finite sequential machines. In *Proceedings of the IFIP Working Conference on Logic and Architecture Synthesis*, pages 107–115, Paris, May-Jun 1990. International Federation for Information Processing.

[8] M. J. Avedillo, J. M. Quintana, and J. L. Huertas. SMAS: a program for concurrent state reduction and state assignment of finite state machines. In *Proceedings of the IEEE International Symposium on Circuits and Systems - ISCAS*, pages 1781–1784, Singapore, June 1991. The Institute of Electrical and Electronics Engineers.

[9] R. G. Bennetts, J. L. Washington, and D. W. Lewin. A computer algorithm for state table reduction. *The Radio and Electronic Engineer*, 42(11):513–520, November 1972.

[10] C. Berthet, O. Coudert, and J. C. Madre. New ideas on symbolic manipulation of finite state machines. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors - ICCD*, Cambridge, MA, September 1990. The Institute of Electrical and Electronics Engineers.

[11] D. Bostick, G. D. Hachtel, R. Jacoby, M. R. Lightner, P. Moceyunas, C. R. Morrison, and D. Ravenscroft. The Boulder optimal logic design system. In *Proceedings of the IEEE International Conference on Computer-Aided Design - ICCAD*, pages 62–65, Santa Clara, CA, November 1987. The Institute of Electrical and Electronics Engineers.

[12] R. K. Brayton. Multi-level logic synthesis. In *Proceedings of the IEEE International Conference on Computer-Aided Design - ICCAD*, Santa Clara, CA, November 1989. The Institute of Electrical and Electronics Engineers. Tutorial Section, 60 pages.

[13] R. K. Brayton, R. Camposano, G. de Micheli, R. H. J. M. Otten, and J. van Eijndhoven. *Silicon Compilation*, chapter 7: The Yorktown silicon compiler system. Addison-Wesley Publishing Company, Reading, MA, 1988. Daniel D. Gajski, editor.

[14] R. K. Brayton, J. D. Cohen, G. D. Hachtel, B. M. Tragger, and D. Y. Y. Yun. Fast recursive boolean function manipulation. In *Proceedings of the IEEE International Symposium on Circuits and Systems - ISCAS*, pages 58–62, Rome, May 1982. The Institute of Electrical and Electronics Engineers.

[15] R. K. Brayton, G. D. Hachtel, L. A. Hemachandra, A. R. Newton, and A. L. M. Sangiovanni-Vincentelli. A comparison of logic minimization using ESPRESSO: An APL package for partitioned logic minimization. In *Proceedings of the IEEE International Symposium on Circuits and Systems - ISCAS*, pages 42–48, Rome, May 1982. The Institute of Electrical and Electronics Engineers.

[16] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. L. M. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic, Hingham, MA, 1984.

[17] R. K. Brayton, G. D. Hachtel, and A. L. M. Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78(2):264–300, February 1990.

[18] R. K. Brayton and C. McMullen. The decomposition and factorization of boolean expressions. In *Proceedings of the IEEE International Symposium on Circuits and Systems - ISCAS*, pages 49–54, Rome, May 1982. The Institute of Electrical and Electronics Engineers.

[19] R. K. Brayton, R. Rudell, A. L. M. Sangiovanni-Vincentelli, and A. Wang. MIS: A multiple level logic optimization system. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1062–1081, November 1987.

[20] R. K. Brayton and F. Somenzi. Boolean relations. In *Proceedings of the International Workshop on Logic Synthesis*, Research Triangle Park, NC, May 1989. 9 pages.

[21] D. W. Brown. A state-machine synthesizer - SMS. In *Proceedings of the ACM/IEEE Design Automation Conference - DAC*, pages 301–304, Nashville, June 1981.

[22] F. M. Brown. *Boolean Reasoning: the logic of Boolean equations*. Kluwer Academic, Norwell, MA, 1990.

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

[24] N. L. V. Calazans, R. P. Jacobi, Q. Zhang, and C. Trullemans. Improving BDDs manipulation through incremental reduction and enhanced heuristics. In *Proceedings of the Custom Integrated Circuits Conference*, pages 11.3.1–11.3.5, San Diego, CA, May 1991. The Institute of Electrical and Electronics Engineers.

[25] M. Ciesielski and Marc Davio. FSM assignment. Unpublished note, Philips Research Laboratory, Belgium, Summer, 1990.

[26] M. J. Ciesielski, J.-J. Shen, and M. Davio. A unified approach to input-output encoding for FSM state assignment. In *Proceedings of the ACM/IEEE Design Automation Conference - DAC*, pages 176–181, San Francisco, CA, June 1991.

[27] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. The MIT Electrical Engineering and Computer Science Series. McGraw-Hill Book Company, Cambridge, MA, 1990.

[28] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using Boolean functional vectors. In *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, Leuven, Belgium, November 1989. International Federation for Information Processing.

[29] F. Crowet, M. Davio, J Durieu, G. Louis, and C. Ykman. Boolean recursive algorithms. *Philips Journal of Research*, 43(3-4):324–345, August 1988.

[30] M. R. Dagenais. MCBOOLE: a new procedure for exact logic minimization. *IEEE Transactions on Computer-Aided Design*, CAD-5(1):229–238, January 1986.

[31] M. Damiani and G. de Micheli. Synthesis and optimization of synchronous logic circuits from recurrence equations. In *Proceedings of the European Conference on Design Automation - EDAC*, pages 226–231, Brussels, March 1992.

[32] J. A. Darringer, D. Brand, J. V. Gerbi, W. H. Joyner, Jr., and L. Trevillyan. LSS: A system for production logic synthesis. *IBM Journal of Research and Development*, 28(5):537–545, September 1984.

[33] M. Davio. Reduced dependence state assignment. Unpublished Letter, 1989.

[34] M. Davio and G. Bioul. Representation of lattice functions. *Philips Reasearch Reports*, 25:370–388, 1970.

[35] M. Davio, J.-P. Deschamps, and A. Thayse. *Discrete and Switching Functions*. Editions Georgi - McGraw-Hill, St-Saphorin - Switzerland, 1978.

[36] M. Davio, J.-P. Deschamps, and A. Thayse. *Digital Systems with Algorithm Implementation*. John Wiley & Sons, Chichester, 1983.

[37] Marc Davio, J.-P. Deschamps, and André Thayse. *Discrete and Switching Functions*, chapter 8: The Optimal Covering Problem. Editions Georgi - McGraw-Hill, St-Saphorin - Switzerland, 1978.

[38] G. de Micheli. *Computer-aided synthesis of* PLA*-based systems*. PhD thesis, University of California, Berkeley, CA, April 1984. Memorandum No. UCB/ERL M84/31.

[39] G. de Micheli. Symbolic minimization of logic functions. In *Proceedings of the IEEE International Conference on Computer-Aided Design - ICCAD*, pages 293–295, Santa Clara, CA, November 1985. The Institute of Electrical and Electronics Engineers.

[40] G. de Micheli. Symbolic design of combinational and sequential logic circuits implemented by two-level logic macros. *IEEE Transactions on Computer-Aided Design*, CAD-5(4):597–616, October 1986.

[41] G. de Micheli. Synchronous logic synthesis: algorithms for cycle time minimization. *IEEE Transactions on Computer-Aided Design*, 10(1):63–73, January 1991.

[42] G. de Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. KISS: a program for optimal state assignment of finite state machines. In *Proceedings of the IEEE International Conference on Computer-Aided Design - ICCAD*, pages 209–211, Santa Clara, CA, November 1984. The Institute of Electrical and Electronics Engineers.

[43] G. de Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Transactions on Computer-Aided Design*, CAD-4(3):269–284, July 1985.

[44] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. A synthesis and optimization procedure for fully and easily testable sequential machines. *IEEE Transactions on Computer-Aided Design*, 8(10):1100–1107, October 1989.

[45] S. Devadas and A. R. Newton. Exact algorithms for output encoding, state assignment, and four-level Boolean minimization. *IEEE Transactions on Computer-Aided Design*, 10(1):13–27, January 1991.

[46] M. Dietzfelbinger, A. Karlin, F. Mehlhorn, Meyer auf der Heide, H. Rohnert, and R. Tarjan. Upper and lower bounds for the dictionary problems. In *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, 1988.

[47] T. A. Dolotta and E. J. McCluskey. The coding of internal states of sequential machines. *IEEE Transactions on Electronic Computers*, EC-13(5):549–562, October 1964.

[48] Electronics Research Laboratory - University of California, Berkeley. *Octtools Distribution 3.0. Volume 2B:Tool Man Pages*, March 1989.

[49] H. Fleisher and L. I. Maissel. An introduction to array logic. *IBM Journal of Research and Development*, 19(2):98–109, March 1975.

[50] S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for binary decision diagrams. In *Proceedings of the ACM/IEEE Design Automation Conference - DAC*, Jun-Jul 1987.

[51] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvements of boolean comparison method based on binary decision diagrams. In *Proceedings of the IEEE International Conference on Computer-Aided Design - ICCAD*, pages 2–5, Santa Clara, CA, November 1988. The Institute of Electrical and Electronics Engineers.

[52] D. D. Gajski and R. H. Kuhn. New VLSI tools. *Computer*, 16(12):11–14, December 1983.

[53] C. Gane and T. Sarson. *Structured systems analysis: tools and techniques*. Prentice-Hall, Englewood Cliffs, NJ, 1979.

[54] M. R. Garey and D. S. Johnson. *Computers and Intractability: A guide to the theory of NP-completeness*. W. H. Freeman and Company, San Francisco, CA, 1979.

[55] S. Ginsburg. On the reduction of superfluous states in a sequential machine. *Journal of the Association for Computing Machinery*, 6:259–282, April 1959.

[56] 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-14:350–359, June 1965.

[57] D. Gregory, K. Bartlett, A. de Geus, and G. Hachtel. SOCRATES: a system for automatically synthesizing and optimizing combinational logic. In *Proceedings of the ACM/IEEE Design Automation Conference - DAC*, June 1986.

[58] G. D. Hachtel, A. R. Newton, and A. L. Sangiovanni-Vincentelli. Techniques for programmable logic arrays folding. In *Proceedings of the ACM/IEEE Design Automation Conference - DAC*, pages 147–152, Las Vegas, June 1982.

[59] G. D. Hachtel, J.-K. Rho, F. Somenzi, and R. Jacoby. Exact and heuristic algorithms for the minimization of incompletely specified state machines. In *Proceedings of the European Conference on Design Automation - EDAC*, pages 184–191, Amsterdam, February 1991.

[60] G. Hallbauer. Procedures of state reduction and assignment in one step in synthesis of asynchronous sequential circuits. In *Proceedings of the International IFAC Symposium on Discrete Systems*, pages 272–282, 1974.

[61] D. R. Haring. *Sequential-Circuit Synthesis: state assignment aspects*, volume 31 of *Research Monograph Series*. Thr M.I.T. Press, Cambridge, MA, 1966.

[62] J. Hartmanis and R. E. Stearns. Some dangers in state reduction of sequential machines. *Information and Control*, 5:252–260, September 1962.

[63] J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice-Hall International Series in Applied Mathematics. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1966.

[64] F. Hill and G. Peterson. *Computer aided logical design with emphasis on* VLSI, chapter 10 - Synthesis of clock-mode sequential circuits. John Wiley & Sons, Inc, New York, fourth edition, 1993.

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

[66] J. Hopcroft. *Theory of machines and computations*, chapter An $n \log n$ algorithm for minimizing states in a finite automaton. Academic Press, New York, NY, 1971. Z. Kohavi and A. Paz, eds.

[67] W. S. Humphrey, Jr. *Switching circuits with computer applications*. McGraw-Hill Book Company, Inc, York, PA, 1958.

[68] Sungho Kang. *Automated synthesis of* PLA *based systems*. PhD thesis, Stanford University, Stanford, CA, 1981.

[69] L. N. Kannan and D. Sarma. Fast heuristic algorithms for finite state machine minimization. In *Proceedings of the European Conference on Design Automation - EDAC*, pages 192–196, Amsterdam, February 1991.

[70] R. M. Karp. Some techniques of state assignment for synchronous sequential machines. *IEEE Transactions on Electronic Computers*, EC-13(5):507–518, October 1964.

[71] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[72] Zvi Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill Computer Science Series. McGraw-Hill Book Company, New Delhi, second edition, 1978.

[73] T. J. Kowalski. *Silicon Compilation*, chapter 5: The VLSI design automation assistant: an architecture compiler. Addison-Wesley Publishing Company, Reading, MA, 1988. Daniel D. Gajski, editor.

[74] K. B. Krohn and J. L. Rhodes. Algebraic theory of machines. In *Symposium on Mathematical Theory of Automata*, New York, NY, April 1962. Microwave Research Institute, Polytechnic Press.

[75] E. B. Lee and M. Perkowski. Concurrent minimization and state assignment of finite state machines. In *Proceedings of the 1984 International Conference on Systems Man and Cybernetics*, pages 248–260, Halifax, October 1984.

[76] C. Leiserson, F. Rose, and J. Saxe. Optimizing synchronous circuits by retiming. In *Third Caltech Conference on VLSI*, pages 87–116. Computer Science, 1983.

[77] B. Lin. Restructuring of synchronous logic circuits. In *Proceedings of the European Conference on Design Automation - EDAC*, pages 205–209, Paris, February 1993. The Institute of Electrical and Electronics Engineers, IEEE Computer Society Press.

[78] Bill Lin and A. Richard Newton. A generalized approach to the constrained cubical embedding problem. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors - ICCD*, pages 400–403. The Institute of Electrical and Electronics Engineers, October 1989.

[79] C. N. Liu. A state variable assignment method for asynchronous sequential switching circuits. *Journal of the Association for Computing Machinery*, 10:209–216, April 1963.

[80] S. Malik, E. M. Sentovich, R. K. Brayton, and A. Sangiovanni-Vincentelli. Retiming and resynthesis: optimizing sequential networks with combinational techniques. *IEEE Transactions on Computer-Aided Design*, 10(1):74–84, January 1991.

[81] Sharad Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proceedings of the IEEE International Conference on Computer-Aided Design - ICCAD*, pages 6–9, Santa Clara, CA, November 1988. The Institute of Electrical and Electronics Engineers.

[82] E. J. McCluskey. Minimization of Boolean functions. *Bell Laboratories Technical Journal*, 35:1417–1444, November 1956.

[83] G. H. Mealy. A method for synthesizing sequential circuits. *Bell Systems Technical Journal*, 34:1045–1079, September 1955.

[84] Bertrand Meyer. Reusability: the case for object-oriented design. *IEEE Software*, 4(2):50–64, March 1987.

[85] E. F. Moore. Gedanken experiments on sequential machines. *Automata Studies*, pages 129–153, 1956.

[86] Eugenio Morreale. Computational complexity of partitioned list algorithms. *IEEE Transactions on Computers*, C-19(5):421–427, May 1970.

[87] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney. The transduction method - design of logic networks based on permissible functions. *IEEE Transactions on Computers*, C-38(10):1404–1423, October 1989.

[88] Stefan Näher. *LEDA User Manual - Version 3.0.* Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1992.

[89] A. R. Newton and A. L. Sangiovanni-Vincentelli. CAD tools for ASIC design. *Proceedings of the IEEE*, 75(6):765–775, June 1987.

[90] C. A. Papachristou and Debabrata Sarma. An approach to sequential circuit construction in LSI programmable arrays. *IEE Proceedings*, 130(5):159–164, September 1983.

[91] M. C. Paull and S. H. Unger. Minimizing the number of states in incompletely specified sequential switching functions. *IRE Transactions on Electronic Computers*, EC-8:356–367, September 1959.

[92] M. A. Perkowski and J. E. Brown. Automatic generation of don't cares for the controlling finite state machine from the corresponding behavioral description. In *Proceedings of the IEEE International Symposium on Circuits and Systems - ISCAS*, pages 1143–1146, New Orleans, LA, May 1990. The Institute of Electrical and Electronics Engineers. volume 2.

[93] M. A. Perkowski and J. Liu. Generation of finite state machines from parallel program graphs in DIADES. In *Proceedings of the IEEE International Symposium on Circuits and Systems - ISCAS*, pages 1139–1142. The Institute of Electrical and Electronics Engineers, 1990.

[94] M. A. Perkowski and N. Nguyen. Minimization of finite state machines in SuperPeg. In *The Proceedings of the Midwest Symposium on Circuits and Systems*, pages 139–147, Lusville, Kentucky, August 1985.

[95] C. P. Pfleeger. State reduction in incompletely specified finite-state machines. *IEEE Transactions on Computers*, C-22(12):1099–1102, December 1973.

[96] J. Rabaey, H. de Man, J. Vanhoof, and F. Goossens, G.and Catthoor. *Silicon Compilation*, chapter 8: CATHEDRAL-II: a synthesis system for multiprocessor DSP systems. Addison-Wesley Publishing Company, Reading, MA, 1988. Daniel D. Gajski, editor.

[97] B. Reusch and W. Merzenich. Minimal coverings for incompletely specified sequential machines. *Acta Informatica*, 22:663–678, 1986.

[98] F. Romeo and A. Sangiovanni-Vincentelli. Probabilistic hill-climbing algorithms: properties and applications. In *Chapel Hill Conference on Very Large Scale Integration*, 1985.

[99] R. Rudell and A. Sangiovanni-Vincentelli. ESPRESSO-MV: algorithms for multiple-valued logic minimization. In *Proceedings of the Custom Integrated Circuits Conference*, pages 230–234, June 1985.

[100] R. Rudell and A. Sangiovanni-Vincentelli. Exact minimization of multiple-valued functions for PLA optimization. In *Proceedings of the IEEE International Conference on Computer-Aided Design - ICCAD*, pages 352–355, Santa Clara, CA, November 1986. The Institute of Electrical and Electronics Engineers.

[101] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):727–750, September 1987.

[102] D. E. Rutherford. *Introduction to lattice theory*, volume 2 of *University Mathematical Monographs*. Oliver & Boyd, Edinburgh, Scotland, 1965.

[103] Alexander Saldanha, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli. A framework for satisfying input and output encoding constraints. In *Proceedings of the ACM/IEEE Design Automation Conference - DAC*, pages 170–175, San Francisco, CA, June 1991.

[104] G. Saucier, M. C. de Paulet, and P. Sicard. ASYL: A rule-based system for controller synthesis. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1088–1097, November 1987.

[105] C.-J. Shi and J. A. Brzozowski. An efficient algorithm for constrained encoding and its applications. Technical Report CS-92-20, University of Waterloo, Waterloo, Canada, April 1992.

[106] C.-J. Shi and J. A. Brzozowski. Efficient constrained encoding for VLSI sequential logic synthesis. In *Proceedings of the European Design Automation Conference - EURO-DAC*, pages 266–271, Hamburg, Germany, September 1992. IEEE Computer Society Press.

[107] Jay R. Southard. *Silicon Compilation*, chapter 6: Algorithmic system compilation: silicon compilation for system designers. Addison-Wesley Publishing Company, Reading, MA, 1988. Daniel D. Gajski, editor.

[108] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, MA, second edition, 1991.

[109] S. Su and P. Cheung. Computer minimization of multivalued switching functions. *IEEE Transactions on Computers*, C-21(9):995–1003, September 1972.

[110] Michael Tiemann. *User's Guide to GNU C++*. Free Software Foundation, Inc., Cambridge, MA, March 1990. Version 2.4.5.

[111] James H. Tracey. Internal state assignment for asynchronous sequential machines. *IEEE Transactions on Electronic Computers*, EC-15(4):551–560, August 1966.

[112] C.-J. Tseng, A. M. Prabhu, C. Li, Z. Mehmood, and M. M. Tong. A versatile finite state machine synthesizer. In *Proceedings of the IEEE International Conference on Computer-Aided Design - ICCAD*, pages 206–209, Santa Clara, CA, November 1986. The Institute of Electrical and Electronics Engineers.

[113] S. H. Unger. A row assignment for delay-free realizations of flow tables without essential hazards. *IEEE Transactions on Computers*, C-17(2):146–158, February 1968.

[114] S. H. Unger. *Asynchronous sequential switching circuits*. Wiley-Interscience – John Wiley & Sons, New York, NY, 1969.

[115] T. Villa and A. Sangiovanni-Vincentelli. NOVA: state assignment of finite state machines for optimal two-level logic implementation. In *Proceedings of the ACM/IEEE Design Automation Conference - DAC*, pages 327–332, Las Vegas, June 1989.

[116] T. Villa and A. Sangiovanni-Vincentelli. NOVA: state assignment of finite state machines for optimal two-level logic implementation. *IEEE Transactions on Computer-Aided Design*, 9(9):905–924, September 1990.

[117] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[118] Saeyang Yang. Logic synthesis and optimization benchmarks. Technical report, Microelectronics center of North Carolina, Research Triangle Park, NC, January 1991. Version 3.0.

[119] Saeyang Yang and Maciej J. Ciesielski. Optimum and suboptimum algorithms for input encoding and its relationship to logic minimization. *IEEE Transactions on Computer-Aided Design*, 10(1):4–12, January 1991.

[120] C. Ykman-Couvreur and C. Duff. *Logic and architecture synthesis for silicon compilers*, chapter 1: Two level and multilevel synthesis. Article: Multi-level Boolean optimization for incompletely specified Boolean functions in PHIFACT. Elsevier Science Publishers B. V., Amsterdam, 1989.

[121] M. Yoeli. Decomposition of finite automata. *IRE Transactions on Electronic Computers*, EC-12(3):322–324, June 1963.

[122] Jacques Zahnd. *Machines Séquentielles*, volume XI of *Traité d'Électricité*. Editions Georgi, St-Saphorin - Switzerland, second edition, 1980. (In French).

[123] Jacques Zahnd. Private letter to the author. (in French), September, 1993.

Appendices

Appendix A

I/O Formats for FSM and Discrete Function Descriptions

Appendix B

Requirements for an FSM Exploratory Environment

Appendix C

Manual Pages for ESPRESSO, DIET, NOVA and STAMINA

Appendix D

Manual Pages for ESPRESSO, DIET, NOVA and STAMINA