SYNTHESIS OF VLSI DESIGNS WITH SYMBOLIC TECHNIQUES

by

Bill Lin

Memorandum No. UCB/ERL M91/105

27 November 1991

SYNTHESIS OF VLSI DESIGNS WITH SYMBOLIC TECHNIQUES

by

Bill Lin

Memorandum No. UCB/ERL M91/105

27 November 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Synthesis of VLSI Designs with Symbolic Techniques

Bill Lin

Ph.D.
University of California, Berkeley

Department of Electrical Engineering
and Computer Sciences

Abstract

Very large-scale integrated circuits are essential in modern digital electronic systems. Integrated circuits with over a million transistors are possible with current technology. The design of these circuits is an extremely difficult and time consuming process, virtually impossible without the use of design-aids to assist in some aspects of design. In this dissertation, a variety of techniques are presented for automating this design process starting from a register-transfer level hardware description of the desired functionality down to an optimized circuit implementation for fabrication. This automation is referred to as *synthesis*. Specifically, the synthesis of sequential designs is considered. To model real life situations, the hardware description may contain variables that carry *symbolic* values. This form of specification is referred to as a *symbolic specification*. The first contribution of this work is to present the concept of *symbolic relation* for specifying multiple choices of output mappings. This is useful for capturing the next-state behavior of a finite state machine in the presence of equivalent states. The optimization problem requires both the selection of output mappings and binary code assignments. A unified framework for solving this problem exactly for two-level implementations has been developed. The second contribution of this research is a set of encoding algorithms for multi-level logic implementations. These algorithms have the merit of being very fast and can be used to encode large hardware descriptions compiled from hardware description languages. Techniques for optimizing sequential circuits, once they have been encoded, have also been developed in this research. These techniques are based on the use of global state-space information with well-developed combinational logic optimization algorithms. A key problem that arises is the need for efficient algorithms to derive state-space information for large sequential circuits. Efficient algorithms based on binary decision diagrams have been developed for this purpose. An associated problem is the state minimization of large sequential circuits. New concepts and machinery for representing and manipulating equivalence classes efficiently are presented for solving this and related problems.

Prof. A. Richard Newton
Thesis Committee Chairman

Synthesis of VLSI Designs with Symbolic Techniques

Bill Lin

Acknowledgments

I would first like to thank my advisor, Prof. Richard Newton, for his guidance, inspiration, encouragement, and support during my graduate years at Berkeley. This work would not have been possible without his help and trust in me. I am also grateful to Prof. Bob Brayton for taking the burden of being on both my qualifying examination and thesis committees. He read my dissertation meticulously and offered many constructive criticisms. He has also influenced me to be more rigorous and precise in my thinking. I have also to thank Prof. Jan Rabaey for being on my qualifying examination committee. Although I have not had the opportunity to work with him directly, I have enjoyed all our stimulating conversations on a broad range of topics. Prof. Sarah Beckman has been a member of both my qualifying examination and thesis committees. I am thankful for her time and suggestions. My research was sponsored in part by National Science Foundation and Defense Advanced Research Project Agency. I am grateful for their financial support.

Part of the work reported here has been done in collaboration with others. The symbolic relations work was done together with Fabio Somenzi of University of Colorado, Boulder. The sequential optimization work using implicit enumeration was done together with Hervé Touati.

Being a part of the Berkeley CAD-group has really been an enriching experience. I have Richard Newton, Bob Brayton, Don Pederson, and Alberto Sangiovanni-Vincentelli to thank for creating such an extraordinary research environment and for recruiting a truly remarkable team of people. This might well be the "best" place in the world for research. I have been extremely fortunate and privileged to be a part of this talented group. I have also to thank the many interesting and brilliant people in the group for making the CAD-group an exciting place. Thanks go to Wendell Baker, Brian O'Krafka, and Masahiro Fukui for being great office-mates. I would like to thank Pranav Ashar, Wendell Baker, Mark Beardslee, Srinivas Devadas, Abhijit Ghosh, Tim Kam, Sharad Malik, Rajeev Murgai, Alex Saldanha, Hamid Savoj, Ellen Sentovich, Narandra Shenoy, Kanwar Jit Singh, Hervé Touati, Yosinori Watanabe, and Greg Whitcomb for many interesting discussions on a wide variety of topics. Thanks also to Andrea Casotto, Gary Jones, Chuck Kring, Luciano Lavagno, Brian Lee, Chris Lennard, Rick McGeer, Cho Moon, Jaijeet Roychowdhury, Henry Sheng, Paul Stephan, and Tiziano Villa for making 550-Cory a more lively place. Special thanks to Gary Jones, Henry Sheng, and Dan Preslar for "being there" in the last few months. I

don't think I would have gotten through without them.

Many thanks to Kia Cooper, Elise Mills, and Flora Oviedo for all the friendly assistance provided over the years. Brad Krebs and Mike Kiernan were always helpful in hardware and software problems.

During my graduate years at Berkeley, I have also had the fortune to interact with many people from other places. I have certainly enjoyed my interactions with Fabio Somenzi, Gary Hachtel, and Xuejun Du from University of Colorado, Boulder. I have also had the pleasure to interact with Olivier Coudert and Jean-Christophe Madre from the Bull Research Center in France, and Takayasu Sakurai from Toshiba Corporation in Japan. Apart from many stimulating discussions on research issues, David Ku from Stanford has also been a very good friend. I hope our friendship continues for many years to come.

Above all, I would like to thank Joyce Fung for her constant love and support, especially this past year. Words cannot express my feelings. We have been through a great deal together. I don't think I could have made it through without her. I would like to thank my family, especially my parents, Michael and Kitty Lin, for their support, encouragement, and enthusiasm in my life. I thank them for everything that I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Very Large Scale Integrated (VLSI) circuits are widely used in modern digital electronic systems. The development of a new system usually requires the design of several custom application-specific integrated circuits (ASIC's). Using current VLSI technology, it is possible to manufacture integrated circuits (IC's) with over a million transistors. This level of integration is still increasing at a rapid rate.

With the availability of commercial ASIC vendors, very complex VLSI designs can be readily implemented in silicon by using such ASIC design-styles as standard cells, gate arrays, or sea-of-gates (also known as channel-less gate arrays) [55]. Despite the ease of fabricating ASIC's today, their design time remains one of the most crucial bottlenecks in the overall product development cycle for most new systems. In a highly competitive global market, with many strong domestic as well as international contenders, *time-to-market* is a key strategic factor. Therefore, effective computer-aided design (CAD) tools are urgently needed for helping designers to reduce the time required to design new ASIC's. Techniques for automating VLSI design steps can greatly help in this direction. This automation is referred to as *synthesis*.

1.1 A Design Trajectory

The design of a custom ASIC involves a series of design steps. A typical design process begins with a specification of circuit behavior in the form of a hardware description language (HDL). ELLA [71], VHDL [50], ISPS [6], are examples of hardware description languages. In this dissertation, a hardware description where the specified *cycle-to-cycle*

behavior cannot be changed is referred to as a *register-transfer level* (RTL) model. This means that final implementation must produce the same output at every clock cycle when the same input sequence is applied, *i.e.*, the cycle-to-cycle behavior must be preserved. A hardware specification where the cycle-to-cycle behavior can be modified is referred to as a *behavioral* model. Behavioral synthesis transformations like scheduling allocation, and pipelining can be applied to a behavioral model to obtain an optimized register-transfer level model. The topic of behavioral synthesis is beyond the scope of this research. The reader is instead referred to the following literature for detailed expositions on some representative work in behavioral synthesis [84, 30, 22]. At this time, behavioral synthesis is still an active area of research and many open issues remain unresolved.

If the hardware is only specified in terms of Boolean values, *i.e.*, in terms of 0's and 1's, then software compilation-like techniques [1, 83] can be used to map the RTL model into an interconnection of combinational logic blocks and synchronous memory elements directly. Many RTL synthesis systems assume this type of description [83]. However, to model real life situations more naturally, the hardware specification may include *symbolic* (also referred to as multi-valued) variables, that can assume finite sets of values, in addition to Boolean variables, which can only assume values from the set {0,1}. This form of specification is referred to as a *symbolic specification*. For example, symbolic specification is widely used in describing finite state machines (FSM's) where the internal states are initially specified symbolically rather than with Boolean vectors. This is, however, a highly restricted form of symbolic specification. In general, the concept can be used for describing complete hardware designs with potentially many symbolic variables. A number of hardware description languages, including VHDL [50] and ELLA[71], provide abstract data typing mechanisms for specifying the variables and functional behavior symbolically. As an example, a partial hardware description written in VHDL for a process control interface module is shown in Figure 1.1. Here, the variables representing process instructions may take on four possible values from the symbol set {ADD, SUB, MUL, CMP}, and the variables representing the control stages may assume the values {WAIT, RESET, FETCH}. This example is shown to illustrate the use of symbolic variables and the description of symbolic functions.

Since symbolic descriptions cannot be implemented in digital logic directly with conventional logic families [1], a binary representation must be derived. This can be obtained

[1] Some multi-valued logic circuit families do exist, but they are not yet practical for commercial use.

```
--
-- declaration of symbolic data types
--
package data_type is
    type stages is (PAUSE, RESET, FETCH);
    type instructions is (ADD, SUB, MUL, CMP);
end data_type;

--
-- interface specification
--
use work.data_type.all;
entity process_control is
    port (clk, c, flag1, flag2:  in bit;
        mode:  out bit; opcode1, opcode2:  out instructions);
end process_control;

--
-- model specification
--
architecture behavior of process_control is
    component latch port
        (clk:  in bit; d:  in bit; q:  out bit);
    end component;
    signal pfork, pjoin:  stages;
begin
    process (c, flag1, flag2, pfork)
    begin
        if (pfork = PAUSE) and (flag1 = '0') then
            mode <= '0'; opcode1 <= SUB; opcode2 <= MUL; pjoin <= PAUSE;
                        :
                        :
        elsif (pfork = FETCH) and (flag1 = '0') then
            mode <= '0'; opcode1 <= CMP; opcode2 <= SUB; pjoin <= FETCH;
        end if;
    end process;
    state:latch port map (clk, pjoin, pfork);
end behavior;
```

Figure 1.1: A Symbolic Specification Fragment written in VHDL.

by *encoding* the symbolic variables with binary-valued variables. This process is referred to as *symbolic encoding*. During symbolic encoding, a binary pattern must be determined to represent uniquely each symbolic value in the symbol-set. Each symbolic variable is then replaced by a set of binary-valued variables, known as *encoding variables*, to represent the binary encoding patterns. For example, in the case of the symbol set representing the process instructions in Figure 1.1, the four admissible values {`ADD`, `SUB`, `MUL`, `CMP`} might be represented with the binary codes `00`, `01`, `11`, `10`, respectively, using two binary-valued encoding variables per symbolic value. Depending on the choice of encoding, the resulting Boolean logic may differ substantially. Therefore, it is crucial to develop effective encoding algorithms that can optimize for the eventual logic implementation under some criteria, *e.g.*, area, performance, or testability.

After symbolic encoding, an interconnection of combinational logic blocks and memory elements is generated. Each combinational logic block is made up of logic gates interconnected in a prescribed way to implement a particular Boolean logic function. The memory elements are used to store data between successive evaluations of the logic blocks. The memory elements are also referred to as *latches* or *registers*. This block-level description is referred to as a *sequential logic network*. The next step in the design process is to optimize the sequential logic network under some criteria. This process is referred to as *sequential logic synthesis* or *sequential logic optimization*. This step is extremely crucial in the overall synthesis process since the initially translated description may contain a significant amount of unnecessary (redundant) logic. Again, the optimization criteria may be in terms of area, performance, testability, or some combination. In this research, a restricted class of sequential circuits called *synchronous circuits* is assumed. These circuits have the property that a common global clock is used to determine when the memory elements latch in new data. A large percentage of circuits designed fall into this category. This is especially the case for semi-custom designs that target gate array or standard cell technologies.

The result of sequential logic synthesis is an optimized gate-level description of the circuit. The next step in the synthesis process is to produce a mask-level description for fabrication. There are many approaches to performing this transformation. One widely used approach is to map the logic gates and memory elements from an optimized gate-level description on to a set of pre-designed library cells. This mapping step is referred to as *technology mapping* [52, 31, 87]. Since the actual physical layouts for the library cells are available, accurate area and load information for each cell can be obtained easily. The final

step in the synthesis process is to place and route the mapped library cells. A detailed exposition on various placement and routing techniques can be found in [55]. The placed and routed mask-level description can be used to manufacture the final product.

1.2 Previous Work

1.2.1 Symbolic Encoding Techniques

The encoding problem is perhaps one of the oldest problems in switching and automata theory [46]. In general, a symbolic specification of hardware may have either symbolic input variables, symbolic output variables, or both. Input encoding refers to the version of the problem where only symbolic input variables are considered, and output encoding refers to the version where only symbolic outputs are considered. In the case of state assignment for finite state machine synthesis, the two symbolic variables representing the present-state input and the next-state output are in fact used to represent the same set of symbolic values. Additional constraints must be imposed such that the same encoding is selected for both variables. This instance of the encoding problem is referred to as the input-output encoding or the state assignment problem. For optimization, the target implementation can be either two-level or multi-level logic. Depending on the target implementation, the goal for the encoding step, be it input, output, or input-output encoding, varies. Encoding problems are difficult because they typically have to model a complicated optimization step that follows.

Two-Level Encoding: In the case where the target implementation is two-level logic, many approaches have been proposed. The encoding problem has been studied since the 1960's [38, 46], but these earlier approaches do not have a strong correlation with the logic minimization step that follows. De Micheli *et al.* [70] introduced a new paradigm where the two-level encoding problem was divided into two stages: an encoding independent optimization phase called *symbolic minimization*, and a constraint satisfaction step. In the symbolic minimization step, two-level logic minimization techniques [78, 74, 67] were extended to perform minimization with symbolic variables. The minimized symbolic result can then be mapped to a binary implementation by satisfying a set of encoding constraints. This paradigm was originally introduced for the input encoding problem as an approximation to state assignment. It was later generalized to solve the output encoding and input-output

encoding problems [68]. Similar approaches were followed in [92, 95]. These approaches are heuristic in nature. In [35], Devadas *et al.* presented exact symbolic minimization algorithms that can solve the problems of input, output, and input-output encoding for the two-level case.

In all of these approaches, symbolic functions were assumed. That is, for every possible input condition, either the output is left unspecified or there is exactly one mapping. For example, in the classical state assignment problem, the next-state function is a symbolic function since every primary input and present-state combination is deterministically mapped to a unique next-state. Thus, the techniques described above are applicable, and the problem can be regarded as well-solved. However, in many important applications in synthesis, the need for capturing *multiple output choices* arises [60, 63, 61]. Symbolic *relations* (one-to-many mapping) rather than functions must be considered, meaning that the possible output response for a given input condition may be one of several symbolic values rather than just one. For example, when synthesizing a finite state machine, either in isolation or in the context of an interacting network, there may be a number of equivalent states. Equivalent states may be exploited by permitting the next-state to be any one of the equivalent states [60]. Previous encoding methods did not consider the selection of output mapping (*viz.* symbolic relation). However, the cost of the final implementation depends heavily on this mapping. A careful selection is therefore extremely crucial. Unfortunately, the choice of output mapping interacts in complex ways with the encoding process. Ideally, both problems should be solved simultaneously. The problem of minimizing symbolic relations with both degrees of freedom considered has not be been addressed so far. Also, equivalent states may be assigned the same binary code, provided that appropriate constraints are satisfied. This additional degree of freedom has also not been explored in previous work.

Multi-Level Encoding: The problem of encoding an arbitrary symbolic specification as to minimize the area of the eventual multi-level logic implementation is an extremely difficult problem in general. This is partly due to the inherent difficulties of multi-level logic optimization. Current encoding techniques that target multi-level logic implementation can be classified into two classes: symbolic-minimization-based and estimation-based. In the work of Malik *et al.* [56], a symbolic minimization procedure was proposed for encoding specifications with symbolic input variables. Analogous to two-level symbolic

minimization, multi-level logic transformations like algebraic decomposition were extended to minimize under symbolic inputs. These multi-level symbolic minimization techniques can be used to solve the input encoding problem. Although the proposed approach has been shown to produce good results for the input encoding problem, the current implementation relies on an expensive simulated annealing based encoding step that uses two-level logic minimization [12] in the the inner loop of the annealing procedure to determine the size of the algebraic factors [65]. Multi-level symbolic minimization for output encoding remains an open problem.

An alternative approach is to *estimate* the outcome of multi-level logic optimization. Devadas *et al.* [33] proposed an approach for state assignment based on enhancing the likelihood of finding common subexpressions in the encoded logic prior to logic optimization. In this approach, weights were statically computed between pairs of states based on the observation that encoding states close together in the Boolean space may result in many common subexpressions. Following the weight computation phase, a *graph embedding* step is solved to encode heuristically state pairs with large weights closer together in the Boolean space. Recently, new encoding procedures based on this encoding paradigm have been presented in [59] and [39]. These new procedures make use of improved estimation models and new graph embedding algorithms that have been shown to produce consistently better results.

In contrast with the symbolic minimization approach, the choice of code assignments in estimation-based approaches are determined by modeling the effects of encoding on the multi-level logic optimization process. The main drawback is the weak correlation that these approaches have with the actual logic transformations that are typically used in multi-level logic synthesis. However, they have been shown to consistently yield good results, and have the merit of being extremely fast.

1.2.2 Sequential Optimization Techniques

Combinational logic optimization has reached significant maturity in the past decade. This is partially reflected by the number of successful combinational logic optimization systems that have been developed in universities as well as the industry [78, 13, 10, 29, 45]. A detailed review of the state-of-the-art in combinational logic optimization techniques can be found in [12, 15]. While combinational logic optimization techniques

are relatively well-developed, research in sequential optimization is still in its earlier stages. Techniques for sequential optimization can be classified into two categories: those that work on a state transition graph (STG) model of the finite state machine, and those that work directly at the structure of the sequential circuit.

STG-Based Optimization: Sequential optimization techniques that work at the state transition graph level have been studied since the 60's [46, 54]. Traditionally, a state transition graph model is converted to a sequential logic network *viz.* the process of state assignment. While effective state assignment techniques are available, they do not explore possible state assignments from other equivalent state transition graph structures. Therefore, the solutions achieved using even *exact* state assignment may be suboptimal. One heuristic is to apply a state minimization procedure [44, 54] to reduce the number of internal states prior to state encoding. The primary objective of classical state minimization is to minimize the number of states. It is well known, however, that non-reduced state machines may actually lead to superior logic implementations [47]. Techniques for relating state minimization and state assignment have not been developed to date.

Algorithms for decomposing a finite state machine into a set of interacting FSM's have been proposed. Earlier work on FSM decomposition was based on a theory of partition [46, 54]. Researchers have recently developed more sophisticated methods that attempt to relate FSM decomposition with state assignment and two-level logic minimization [36, 5]. Methods for synthesizing interacting FSM's have also been actively pursued [32, 4].

Structural-Based Optimization: Transformations that attempt to optimize the sequential logic at the circuit-level have been proposed. Leiserson and Saxe [57] developed a transformation called *retiming* for improving the performance of a synchronous digital system by re-positioning the registers. This technique was further developed in [69] in the context of sequential logic optimization. In [66], retiming was used to temporarily reposition the registers such that combinational logic optimization can be applied to a larger portion of the network. Although these retiming-based techniques have been found to be useful for clock cycle minimization, they have had only limited success for area optimization.

Another important source of sequential optimization is the use of *don't-care* information. Don't-care conditions represent the degrees of freedom that the circuit can be modified without affecting its intended behavior. In [7, 15], various don't-care sets for

combinational logic circuits have been identified. Recently, efficient procedures have been developed for computing them [81]. For hierarchically-defined logic networks, Brayton and Somenzi [16] proposed the use of Boolean relations to capture the possible degrees of freedom. This is related to the work of Cerny and Marin [23] of *observability relations* for hierarchical combinational network specification and synthesis.

Combinational don't-care conditions are not sufficient when sequential machines are considered. For finite state machines, Devadas *et al.* [34] described a don't-care based optimization procedure for two-level logic implementations that can produce a fully-testable sequential machine, under the *single-stuck-at* fault model in testing [18], without access to the memory elements. In their work, invalid and equivalent state information was extracted from a state transition graph representation of the FSM. This information was then used by a two-level Boolean minimizer [12, 14] as don't-care conditions to obtain an optimized implementation. This procedure was limited to those sequential circuits whose state-space can be enumerated explicitly. Multi-level logic optimization was not considered.

In [28], the notion of synchronous don't-cares was proposed to capture sequential don't-care conditions at the circuit-level. New synchronous don't-care sets were proposed along with computation algorithms. Sequential don't-care sets that correspond to an acyclic portion of the sequential circuit can be captured using their techniques. Larger sets of don't-cares may be derived by considering different acyclic portions of the feedback network. It is not currently known if all degrees of freedom can be captured this way.

1.2.3 Symbolic Representations and Computation

Recently, it has become apparent that many tasks in logic synthesis are intimately related in that they are often fundamentally dependent on the same set of basic logic manipulations. Therefore, it is important to improve existing and develop new logic manipulation methods. The binary decision diagram (BDD) [20, 11, 2] is an efficient data structure for representing logic. Bryant proposed a restricted version of BDD that requires an ordering on the variables. It was shown that the resulting reduced decision graph is canonical with respect to a given variable ordering. It also has the important advantage that Boolean operations (*e.g.*, Boolean and and Boolean or) can be performed directly on the data structure. Another important idea is the concept of characteristic functions proposed by Cerny and Marin [23]. Their original idea was to capture the input-output behavior of a combina-

tional circuit as a single Boolean function. This form of specification can naturally express multiple output mappings, as in Boolean relations. The concept of characteristic function can be applied in general to represent and manipulate finite sets of elements. Recently, Coudert *et al.* developed the concept of implicit enumeration for efficiently traversing large state-spaces [26, 27]. This method is based on the use of BDD's and characteristic functions for implicitly representing the state space. While BDD's, characteristic functions, and implicit enumeration provide the basic machinery and concepts to manipulate logic functions, sets, and state-spaces efficiently, efficient representations and manipulation algorithms have not been developed for handling *equivalence classes*. The ability to represent and manipulate equivalence classes efficiently is important in a number of problems in sequential logic synthesis, *e.g.*, manipulation of equivalent states in finite state machines.

1.3 Overview of Dissertation

The goal of this research has been to develop new synthesis techniques for automating the design process for transforming a symbolic specification of hardware into an optimized VLSI circuit. The designs considered are synchronous sequential circuits that are composed of combinational logic blocks and synchronously-clocked memory elements. The first part of the dissertation is focussed on the optimization problem of encoding a symbolic specification into a binary representation. This is necessary since conventional digital circuits can only implement Boolean logic.

In Chapter 2, the encoding problem for two-level logic implementations is considered. Although this problem has been well-investigated in the past, techniques developed thus far have not considered the interactions between the encoding process and sequential optimization. In the case of finite state machine (FSM) synthesis, optimizations at the FSM-level can be implicitly captured in the encoding process if multiple output choices are considered. This gives rise to the symbolic relations problem where each input condition may be mapped to one of many output choices. The optimization problem requires both the selection of output mappings and code assignments. A unified framework is presented for finding exact solutions, in terms of area, to the symbolic relation minimization problem. The formulation is based on prime generation and minimum-cost covering. Various versions of the minimization problem can be uniformly treated by incorporating *problem-specific* constraints to the covering problem. The problem of FSM synthesis using a symbolic

relations formulation is also described. Extensions to symbolic relations are presented to consider implicitly the merging of equivalent states in the synthesis process. Based on these extensions, the traditionally separate problems of state minimization and state assignment are formulated as a single unified optimization problem.

In Chapter 3, new algorithms are described for solving the encoding problem for multi-level logic implementations. For encoding large symbolic specifications, such as those translated from hardware description languages (*e.g.,* VHDL [50] or ELLA [71]), fast, but effective, algorithms are required. The techniques in this chapter build on the work of Devadas *et al.* [33]. The new techniques presented include modified estimation models for general symbolic specifications rather than state machines, and new optimization algorithms for solving the graph embedding problem.

The second part of this dissertation is concerned with the problem of optimizing an already encoded circuit description. Specifically, the optimization of *synchronous* sequential circuits is considered. In Chapter 4, an approach based on combining global state-space information with logic-level transformations is presented. In particular, efficient algorithms are presented for computing invalid and equivalent state information from a structural-level description. The extracted information can then be used as *sequential don't-care* conditions in logic optimization. These don't-care conditions provide more degrees of freedom for combinational logic optimization algorithms to optimize the circuit, both for area and performance improvement. Although synthesis for testability is not directly addressed in this research, it has been shown that the use of these sequential don't-cares in optimization can greatly enhance the testability of the resulting circuit, if not making it fully testable [34]. The main bottleneck in previous methods for extracting similar don't-care information has been the size of the problems that they can handled. Many circuits considered in practice typically contain a large number of latches. Hence, the size of the corresponding state-spaces are usually quite vast, rendering existing methods, based on state diagrams or cube enumeration, impractical. New algorithms based on binary decision diagrams (BDD's [20]) are presented in Chapter 4 that can greatly extend current capabilities in computing these don't-care conditions for large circuits. These algorithms build on the implicit state enumeration techniques recently introduced by Coudert *et al.* [25].

The work presented in Chapter 5 is concerned with efficient techniques for representing and manipulating *equivalence classes.* The main motivation for investigating this problem in this research is in the application of sequential optimization. Specifically, equiv-

alent states information can also be used in synthesis to reduce the size of the state-space by merging them together, *i.e.*, state minimization. To perform this transformation for large sequential circuits, efficient techniques for representing and manipulating equivalent classes are required. These problems are addressed in Chapter 5 by introducing a new representation for equivalence classes and a new Boolean operator for manipulating them that can be applied to very large problem instances.

The work presented in Chapter 6 is concerned with optimizing sequential circuits by means of removing redundant state latches. Unoptimized sequential circuits may contain latches and next-state functions that can be re-derived from information contained in the other latches in the circuits. These latches and next-state functions can be deleted from the circuit if appropriate re-encoding logic is added. A simple algorithm is presented for performing this transformation that has been shown to heuristically reduce the size of the circuits for verification purposes and physical implementations.

Finally, in Chapter 7, conclusions from this work are summarized.

Chapter 2

Symbolic Relations and Two-Level Encoding

2.1 Introduction

Many problems in synthesis can be posed as encoding problems and so effective encoding methods are fundamental to high-quality results from synthesis. As mentioned already in Chapter 1, different encoding techniques must be developed depending on whether the eventual logic implementation is two-level or multi-level. This is, in part, because state-of-the-art logic optimization techniques for both styles of implementations are different, and the cost functions for physical realizations are different. In this chapter, techniques targeting two-level implementations are considered.

For two-level encoding, the eventual physical realization is in most cases a programmable logic array (PLA). Since a PLA is typically implemented as a two-dimensional transistor array, the area complexity is roughly proportional to the number of rows times the number of columns. The number of rows corresponds to the number of product terms (*i.e.*, implicants), and the number of columns is determined by the code lengths used to encode the symbolic variables, as well as the number of primary inputs and outputs of the finite state machine in the most general case. Hence, a widely accepted first-order measure of area complexity of a PLA is simply the number of product terms (which is a first-order approximation). Therefore, the goal of two-level encoding has been to produce a realizable binary implementation with as few product terms as possible. Based on this cost measure,

a significant amount of work has been published since the 1960's, as reviewed briefly in Chapter 1. The most successful approaches for two-level encoding to date have been based on the paradigm of **symbolic minimization**. Symbolic minimization is the process of minimizing directly the logic specification with symbolic variables such that the minimized result can be translated into a binary representation by satisfying some prescribed encoding constraints. Such an approach was used in [70] to minimize functions with only symbolic inputs. The basic approach was later generalized to minimize functions with both symbolic inputs and outputs [68, 92] (*e.g.*, the state assignment problem). Recently, Devadas *et al.* [35] proposed an exact symbolic minimization procedure for minimizing symbolic functions with both symbolic input and output variables.

In all of the previous work on symbolic encoding, **symbolic functions** were assumed. That is, for every possible input condition, either the output is left entirely unspecified or there is exactly one mapping. However, in many interesting applications, **relations** rather than functions have to be considered, meaning that several output choices of symbolic values may be possible for each input condition rather than just one.

Symbolic relations arise in many contexts in synthesis [60, 63, 61]. Many optimization problems can in fact be naturally formulated as a symbolic relation minimization problem. For example, when synthesizing a finite state machine, there may be a number of equivalent states in the original specification. Equivalent states may be exploited by permitting the next-state to be any one of the equivalent states. Let the behavior of a completely specified finite state machine be given by the function $T : I \times \Sigma \rightarrow \Sigma \times \mathcal{O}$. In the classical state machine synthesis problem, T is treated as a symbolic function with $I \times \Sigma$ as the domain (*i.e.*, the machine inputs and present-states) and $\Sigma \times \mathcal{O}$ as the co-domain (*i.e.*, the next-states and machine outputs). For every combination of input and present-state, exactly one next-state and one output pattern is produced. However, if a set of equivalent states $E \subseteq \Sigma \times \Sigma$ is known, where $(y, y') \in E$ means the state y and the state y' are equivalent, then the behavior of the finite state machine, under the degrees of freedom of equivalent states, can in fact be formulated as a symbolic relation $T' \subseteq I \times \Sigma \times \Sigma \times \mathcal{O}$ as follows: for every $(i, x) \in I \times \Sigma$, (y, o) is a possible output mapping for T' if and only if either (y, o) was the mapping for the original function $T(i, x)$, or there exists a state y' such that y and y' are equivalent states and (y', o) was the original mapping for $T(i, x)$.

In this way, it may be possible to implement a more efficient machine using the equivalent states. In essence, T' is a symbolic relation that captures a complete family of

equivalent FSM's that can be obtained by re-directing state transitions to equivalent states in a state transition graph. The idea of using equivalent states to formulate a symbolic relation was first introduced in [58, 60]. Another FSM synthesis problem is the state minimization of incompletely specified finite state machines. In this problem, a set of prime classes are selected to represent the minimized machine. However, a state in the original machine may be covered by a number of prime classes. There is an optimization problem that must be solved in choosing which prime class implements a next-state. This is referred to as the **mapping** problem [75] and can be naturally formulated as a symbolic relation minimization problem.

Other problems that can be formulated with symbolic relations include any applications for Boolean relations with the added degree of re-encoding the binary outputs. For example, consider the problem of minimizing a pair of cascaded PLA's. The signals passing from the first PLA to the second may be re-encoded to simplify the two combinational blocks. A symbolic relation may arise if some output vectors of the first PLA are not differentiable by the second PLA. This is the same phenomenon that gave rise to Boolean relations. However, these intermediate signals can be re-encoded. Straightforward re-encoding without considering relations does not exploit all degrees of freedom.

The goal of two-level symbolic relation minimization is to find the minimum (or a minimal) two-level binary implementation to realize the symbolic relation. The problem of minimizing symbolic relations is fundamentally more difficult than the problem of minimizing symbolic functions. In addition to the degrees of freedom of choosing different encodings for the symbolic variables (both symbolic inputs and outputs), the choice of output mapping for each input condition must also be decided. The choice of output mapping can, and often do, affect tremendously the optimality of the final result. In fact, the choice of output mapping and the choice of encoding have complex interactions with each other. Hence, it is important that they be considered together during the optimization process.

In this chapter, a unified framework is presented for solving various two-level symbolic relation minimization problems **exactly**. The overall framework is based on the classical paradigm of prime implicant generation and covering, but generalized to minimize symbolic relations. The notion of generalized prime implicants (GPI's) proposed by Devadas *et al.* [35] for symbolic function minimization is extended to the concept of **generalized candidate primes** (or simply **generalized c-primes**). This is similar to the notion of candidate primes in the Boolean relation problem [14]. After the generalized c-primes have

been generated, a covering problem must be solved whereby a subset of the generalized c-primes is selected to implement the symbolic relation. In [35], the problem was formulated as a minimum unate covering problem with encodeability check. The branch-and-bound strategies are similar to those used in classical two-level minimization [78], but encodeability must be checked before a solution is declared valid. The encodeability check was performed using a separate graph resolution algorithm.

Here, a **binate covering** formulation of the problem is proposed whereby the covering step, the encodeability check, and the **all-zero code** problem[1] are solved simultaneously. A succinct set of covering constraint equations is given that considers the complete covering process. When formulated as a binate covering problem (BCP), a number of developed bounding strategies (*e.g.*, implications) can be exploited to prune the search space. Also, the formulation into one unified framework permits a deeper understanding of the underlying problem. Most importantly, a variety of symbolic relation minimization problems can be solved uniformly by modifying the covering constraints without the need to change the core mechanisms for solving the binate covering problem.

The framework for symbolic relation minimization is further generalized for solving the FSM synthesis problem. As already mentioned, knowledge of equivalent states can be exploited in FSM synthesis by permitting the next-state transition to be any one of the equivalent states. Formulating the symbolic relation problem this way accounts for all the degrees of freedom of state encoding and re-direction of state transitions in a state transition graph. However, this is not the most general case since equivalent states can be merged to reduce the number of states. Merging of equivalent states can be considered **implicitly** in the minimization process by permitting them to assume an identical code[2]. This is in contrast with the classical statement of the encoding problem where the assignment of distinct codes to each symbolic value is required. Relaxing this requirement adds yet another dimension in the optimization space, but doing so captures implicitly the state minimization process as one of state encoding. The key difference is the use of actual logic area as the cost function rather than the number of states. However, new constraints must be imposed such that resulting solution is realizable. In particular, the implicit merging of some equivalent states may necessitate the merging of other equivalent states (*cf.* Section

[1] In a PLA implementation, if the outputs of a product term are all 0's, then the product term can be discarded. This condition must be accounted for during the minimization process.

[2] The idea of assigning identical codes to equivalent states as a means for considering state merging implicitly was first proposed in [58, 60].

2.3.2 and Section 2.6.3). These additional constraints can be treated uniformly using the unified framework described in this chapter.

The remainder of the chapter is organized as follows. In the next section, definitions and notation used throughout the chapter are given. In Section 2.3, different symbolic relation minimization problems are classified along with a global view of the unified framework, namely prime generation, constraints generation, and minimum covering. The problem of generating generalized candidate primes is examined in Section 2.5. Then in Section 2.6, succinct sets of covering constraints are given for various symbolic relation minimization problems. Specifically, in Section 2.6.3, the complete set of constraints for capturing the degrees of freedom of state encoding, re-direction of state transitions, and implicit merging is succinctly stated. This set of constraints essentially captures a complete family of equivalent FSM's under state minimization and state assignment. In Section 2.7, the binate covering problem, which is also known as the **minimum-cost satisfiability problem**, is addressed. Specifically, a novel approach based on the use of binary decision diagrams (BDD's) is presented that can handle constraints specified in multi-level form directly. It is shown that the minimum cost solution can be found in **linear time** if the corresponding BDD can be built. Experimental results for the algorithms described in this chapter are given in Section 2.8. A complete example is worked out to illustrate the overall minimization process.

2.2 Background and Terminology

2.2.1 Boolean Functions and Relations

Let $B = \{0,1\}$ be the set of Boolean[3] values. A **Boolean variable**, or binary variable, is a variable that can accept values from the set B. A **Boolean function** f (also called a binary function) with r binary input variables and n binary output variables is a mapping function from an r-dimensional Boolean space to a n-dimensional Boolean space, denoted as

$$f : B^r \rightarrow B^n.$$

B^r is called the **domain** and B^n is called the **co-domain** of the function f. Each element in the domain of the function is called a **minterm** of the function. Given a subset of the domain

[3]Throughout the dissertation, the term "Boolean" will be used to refer to the two-valued set {0, 1}.

$X \subseteq B^r$, the **image** of X with respect to f is the set $f(X) = \{y \in B^n \mid \exists \mathbf{x} \in X : y = f(\mathbf{x})\}$ that the minterms in X can map to. The **range** of a function is the image of the entire domain. A single-output function is a special case where $m = 1$. A multi-output function $f : B^r \rightarrow B^n$ is in fact a collection of single output functions $[f_1, f_2, \ldots, f_n]$.

A Boolean function $f : B^r \rightarrow B^n$ is said to be **incompletely specified** if there exists some minterms in the domain, $X^{DC} \subseteq B^r$, where an output f_i of f is not specified (*i.e.*, the output can take on either value in $B = \{0,1\}$). This set of minterms X^{DC} is called a **don't-care set** of f_i. A more general form of incomplete specification is a Boolean relation defined as follows:

Definition 2.1 *A* **Boolean relation** *is a* **one-to-many** *multi-output Boolean mapping* $\mathcal{R} \subseteq B^r \times B^n$. *For each* **minterm** $\mathbf{x} \in B^r$, $\mathcal{R}(\mathbf{x}) = \{y \in B^n \mid (\mathbf{x}, y) \in \mathcal{R}\}$ *is the set of possible mappings for* x, *also called the* **image** *of* x.

The image for a set of points in the domain is similarly defined. Given a subset of the domain $X \subseteq B^r$, the image of X with respect to $\mathcal{R}$ is the set $\mathcal{R}(X) = \{y \in B^n \mid \exists \mathbf{x} \in X : (\mathbf{x}, y) \in \mathcal{R}\}$. A Boolean relation, in essence, captures a set of completely specified functions over the domain B^r and the co-domain B^n. In the following definitions, the relationship between a Boolean function and a Boolean relation is established.

Definition 2.2 *A Boolean relation* $\mathcal{R} \subseteq B^r \times B^n$ *is said to be* **well-defined** *if* $\forall \mathbf{x} \in B^r$, $\mathcal{R}(\mathbf{x})$ *is not empty.*

Definition 2.3 *A multi-output Boolean function* f *is a* **mapping compatible with** $\mathcal{R}$ *if* $\forall \mathbf{x} \in B^r$, $f(\mathbf{x}) \in \mathcal{R}(\mathbf{x})$. *This is denoted by* $f \prec \mathcal{R}$.

2.2.2 Symbolic Functions

Definition 2.4 *Let* $D_1 = \{d_0, d_1, \ldots, d_{|D_1|-1}\}$, $D_2 = \{d_0, d_1, \ldots, d_{|D_2|-1}\}$, ..., *and* $D_r = \{d_0, d_1, \ldots, d_{|D_r|-1}\}$, *and* $\Sigma_1 = \{\sigma_0, \sigma_1, \ldots, \sigma_{|\Sigma_1|-1}\}$, $\Sigma_2 = \{\sigma_0, \sigma_1, \ldots, \sigma_{|\Sigma_2|-1}\}$, ..., *and* $\Sigma_n = \{\sigma_0, \sigma_1, \ldots, \sigma_{|\Sigma_n|-1}\}$ *be finite sets of elements. Each* D_i *and* Σ_i *represents a finite set of* **symbolic values**. *A* **symbolic function** *with* r *input variables and* n *output variables is a mapping*

$$f : D_1 \times D_2 \times \cdots \times D_r \rightarrow \Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_n.$$

Each variable is called a **symbolic variable**. Associated with each input variable u_i is a set of admissible values $D_i = \{d_0, d_1, \ldots, d_{|D_i|-1}\}$ that the symbolic variable u_i can

assume. Similarly, associated with each output variable v_i is a set of admissible values $\Sigma_i = \{\sigma_0, \sigma_1, \ldots, \sigma_{|\Sigma_i|-1}\}$ that v_i can assume.

The Cartesian product $D \equiv D_1 \times D_2 \times \cdots \times D_r$ is called the **domain** of the symbolic function, and the Cartesian product $\Sigma \equiv \Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_n$ is called its **co-domain.**

For compactness, the symbolic function $f : D_1 \times D_2 \times \cdots \times D_r \rightarrow \Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_n$ may simply be written as $f : D \rightarrow \Sigma$ with a single symbolic input variable taking values from $D_1 \times D_2 \times \cdots \times D_r$ and a single output variable taking values from $\Sigma_1 \times \Sigma_2 \times \cdots \times \Sigma_n$. As in the Boolean case, each point $\mathbf{x} \in D$ in the domain is called a **minterm.**

A completely specified symbolic function is a symbolic function that maps every minterm of the domain to exactly one element in the co-domain. A symbolic function is said to be **incompletely specified** if for some inputs, the value of the some symbolic outputs are left entirely unspecified (*i.e.,* can take any value from the corresponding set of symbolic values). The collection of such points of the domain for a particular output is called the **don't-care set** for that output. To make the terminology more precise, the following distinctions are made.

1. A function $f : D \rightarrow B^n$ with symbolic inputs but only binary outputs is referred to as a **symbolic-input-binary-output function.**

2. A function $f : B^r \rightarrow \Sigma$ with only binary inputs but symbolic outputs is referred to as a **binary-input-symbolic-output function.**

3. A function $f : D \rightarrow \Sigma$ with both symbolic inputs and outputs is referred to as a **symbolic-input-symbolic-output function.**

The latter is the most general case. To avoid confusion, in the remainder of the dissertation, unless otherwise qualified, a **symbolic function** is referred to as a function that maps to symbolic outputs (*i.e.,* either a binary-input-symbolic-output function or a symbolic-input-symbolic-output function), and a **Boolean function** is referred as to a function that maps to binary values with possibly symbolic inputs (*i.e.,* either a binary-input-binary-output function or a symbolic-input-binary-output function)[4]. A symbolic-input-binary-output function will also be called a **symbolic-input Boolean function.**

In the literature [79, 77, 56], a variable that can take on multiple, but finite, number of values has often been referred to as a **multiple-valued** (or multi-valued) variable instead

[4]These points are belabored here because these distinctions are usually not clarified in the literature.

of a symbolic variable. The set of values that the variable can take is often denoted by a set of integers $P = \{0, 1, \ldots, n-1\}$ instead of a set of mnemonics or symbols. The terminology is in fact interchangeable. Note that there is a one-to-one correspondence between a set of n symbols $\Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_{n-1}\}$ and the multiple-valued set $P = \{0, 1, \ldots, n-1\}$. A **multiple-valued function** is a function with multiple-valued variables. It is in fact the same as a symbolic function. The difference is only again in the terminology. However, a **multiple-valued function** is typically referred to as a function with only symbolic inputs (*i.e.*, no symbolic outputs) [79, 77, 56][5]. For historical reasons, some researchers prefer to use the terminology of multi-valued variable and multi-valued function [79, 77, 56] while others prefer to use the terminology of symbolic variable and symbolic function [68, 35, 70, 61]. In this dissertation, terminology of symbolic variable and symbolic function is used.

2.2.3 Symbolic Relations

In this section, a symbolic relation is defined. As in the Boolean case, symbolic functions can only be used to specify a one-to-one mapping and cannot capture fully specifications in the general case with multiple output mappings (the use of don't-cares can capture a limited set of output choices). Hence, a definition for specifying a one-to-many mapping relation with symbolic inputs and symbolic outputs is needed.

Definition 2.5 *A* **symbolic relation** $\mathcal{R} \subseteq D \times \Sigma$ *is a* **one-to-many** *relation over the two sets* D *and* Σ. D *is the* **domain** *of the relation, and* Σ *is the* **co-domain** *of the relation.*

Symbolic relation was originally introduced in [60] as a means for formulating the problem of minimizing state machines under equivalent states.

Definition 2.6 *For each* **minterm** $\mathbf{x} \in D$, *the* **image** *of* $\mathbf{x}$ *is the set of possible mappings* $\mathcal{R}(\mathbf{x}) = \{y \in \Sigma \mid (\mathbf{x}, y) \in \mathcal{R}\}$.

The **image** for a subset of the domain $X \subseteq D$ is the set $\mathcal{R}(X) = \{y \in \Sigma \mid \exists \mathbf{x} \in X : (\mathbf{x}, y) \in \mathcal{R}\}$. This is similar to the terminology used for Boolean relation (*cf.* Section 2.2.1). The domain D may in fact be the Cartesian product of r sets $D_1 \times D_2 \times \ldots \times D_r$. Likewise, the co-domain Σ may actually be the Cartesian product of n sets $\Sigma_1 \times \Sigma_2 \times \ldots \times \Sigma_n$.

[5]The optimization techniques presented in [79, 77, 56] for multiple-valued functions are for functions with symbolic inputs only.

Definition 2.7 *A symbolic relation $\mathcal{R} \subseteq D \times \Sigma$ is said to be* **well-defined** *if $\forall \mathrm{x} \in D$, $\mathcal{R}(\mathrm{x})$ is not empty.*

Definition 2.8 *A multi-output Boolean function $f : B^r \rightarrow B^n$ is a* **compatible mapping** *for $\mathcal{R} \subseteq D \times \Sigma$ if there exists an input encoding $\xi : D \rightarrow B^r$ and an output encoding $\psi : \Sigma \rightarrow B^n$ such that $\forall \mathrm{x} \in D$, $\exists y \in \mathcal{R}(\mathrm{x})$ such that $f(\xi(\mathrm{x})) = \psi(y)$. This is denoted by $f \prec_{\{\xi,\psi\}} \mathcal{R}$.*

Like in the function case, the optimization problems are different for relations depending whether symbolic inputs and/or symbolic outputs are considered. Here, a sub-classification similar to Section 2.2.2 is given:

1. A relation $\mathcal{R} \subseteq D \times B^n$ with symbolic inputs but only binary outputs is referred to as a **symbolic-input-binary-output relation.**

2. A relation $\mathcal{R} \subseteq B^r \times \Sigma$ with only binary inputs but symbolic outputs is referred to as a **binary-input-symbolic-output relation.**

3. A relation $\mathcal{R} \subseteq D \times \Sigma$ with both symbolic inputs and outputs is referred to as a **symbolic-input-symbolic-output relation.**

In [94], the problem of minimizing **multiple-valued relations** is considered where only relations with symbolic-inputs are permitted. Symbolic output variables are not considered[6]. In the remainder of the dissertation, unless otherwise qualified, a **symbolic relation** refers to a relation with symbolic output mappings (*i.e.*, either a binary-input-symbolic-output relation or a symbolic-input-symbolic-output relation), and a **Boolean relation** refers to a relation with binary output mappings (*i.e.*, either a binary-input-binary-output relation or a symbolic-input-binary-output relation). A symbolic-input-binary-output is also referred to as a **symbolic-input Boolean relation.**

2.2.4 Finite State Machines

A finite state machine (FSM) is defined as a 6-tuple $M = \langle I, \mathcal{O}, \Sigma, \delta, \lambda, \sigma_1 \rangle$ where $I = B^r$ represents the primary-input space, $\mathcal{O} = B^n$ represents the primary-output space, and $\Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_{|\Sigma|-1}\}$ represents the state-space. The next-state (symbolic) function

[6]Symbolic outputs are handled in [94] by giving them *one-hot* binary encodings, not by dealing with them in symbolic form.

is defined as $\delta : I \times \Sigma \rightarrow \Sigma$ where each combination of primary-input vector $i \in I$ and present state $\sigma_{ps} \in \Sigma$ is mapped to a next-state $\sigma_{ns} \in \Sigma$. The output function λ is defined as $\lambda : I \times \Sigma \rightarrow \mathcal{O}$ for the case of a **Mealy machine** and $\lambda : \Sigma \rightarrow \mathcal{O}$ for the case of a **Moore machine.** The output is dependent on both the primary-inputs and present-state for a Mealy machine whereas the output is only dependent on the present-state in the case of a Moore machine.

A machine is said to be **fully specified** if for every combination of primary-input and present-state, the next-state and all primary-outputs are defined. Otherwise, the machine is said to be incompletely specified.

A **reset state** $\sigma_r \in \Sigma$ is assumed to be given for the machine. In general, there may be a set of legal reset states, denoted as $X \subseteq \Sigma$. A state σ is said to be **reachable** if there exists an input sequence that will transform the machine from a reset state $\sigma_r \in X$ to σ. Such an input sequence is called the **justification sequence.** A reachable state is also called a **valid state.** If a state is unreachable, then it is called an **invalid state.**

For a completely specified machine, two states σ_i and σ_j are said to be **equivalent** if all possible input sequences when the machine is initially in either of the two states produce the same output response. For an incompletely specified machine, two states σ_i and σ_j are said to be **compatible** if all possible **applicable** input sequences when the machine is initially in either of the two states do not produce conflicting output response. An input vector from a state of an incompletely specified machine is said to be **applicable** if the next-state response for this input is defined.

A FSM can also be specified in terms of a two-level truth table called a **state transition table** where the left hand side represents the domain, corresponding to primary-inputs and present-states, and the right hand side represents the co-domain, corresponding next-states and primary-outputs. This truth table is essentially a specification of the symbolic function $T(= \delta \times \lambda) : I \times \Sigma \rightarrow \Sigma \times \mathcal{O}$.

A FSM can also be represented by a **state transition graph** (STG) $G(V, E, L)$ where each vertex $v \in V$ corresponds to a state $\sigma \in \Sigma$. An edge $e_{ij} \in E$ connects v_i to v_j if there is a primary-input that causes the FSM to evolve from state v_i to state v_j. Associated with each edge is a label $l(e) \in L$ that carries the information of the value of the input that caused that transition and the values of the primary-outputs corresponding to that transition. Let the number of primary-inputs and primary-outputs be r and n respectively. The fanin of a state σ is a set of edges and is denoted $FI(\sigma)$. The fanout of a state σ

is denoted $FO(\sigma)$. The number of fanout edges for a given state is bounded by $|I| = 2^r$. However, in general, the edges are labeled with cubes and are hence considerably more compact. A state transition graph G_1 is said to be **isomorphic** to another state transition graph G_2 if and only if they are identical except for a renaming of states. In the remainder of this dissertation, extended definitions related to finite state machines will be given or clarified when necessary.

2.2.5 Binary Decision Diagrams

A binary decision diagram (BDD) [2, 20] is a data structure to represent a (single-output) Boolean function $f : B^r \rightarrow B$. Specifically, a BDD is a directed acyclic graph (DAG) representation where each internal node in the graph is associated with a Boolean variable and two outgoing arcs. One outgoing arc corresponds to the case when the variable is set to 0, and the other corresponds to when the variable is set to 1. At the leaves of the graph are two constant nodes 0 and 1. In the form introduced by Bryant [20], several restrictions apply. First, a variable ordering is imposed such that all nodes reachable from the current node must have a higher ordering index, except for the constant nodes (which are not associated with a variable). Also, a variable may not appear more than once along any directed path. The third requirement is that all isomorphic subgraphs are combined. Such a resulting BDD is called a a **reduced ordered binary decision diagram** (ROBDD), which is commonly referred to simply as a BDD. A fundamental property of Bryant's BDD is that the representation is unique for a given variable ordering. This canonical property makes it possible verify the equivalence of two Boolean functions simply by comparing their BDD representations. Also, standard Boolean operations like intersection ($f \cdot g$), union ($f + g$), and negation ($\overline{f}$) can be implemented directly on the data structure. Hence, these operations are very efficient with BDD's. Recently, efficient implementation techniques such as the use of strong canonical forms, run-time function caches, and automatic garbage collection have been proposed [11]. Using these techniques, an order of magnitude improvement in performance can be achieved. A particularly useful operator with BDD's is the **if-then-else** (ITE) operator.

Definition 2.9 *Given three Boolean formulae f, g, and h over some Boolean space B^r, the* **if-then-else** *(ITE) operator is defined as follows:*

$$ITE(f, g, h) = \overline{f} \cdot g + f \cdot h. \tag{2.1}$$

This operation forms the foundation of many basic Boolean operations, as is seen later.

2.3 Minimization of Symbolic Relations

2.3.1 Definitions of Symbolic Relations Problems

In this section, relationships between different two-level minimization problems are examined. First, the minimization problems where different symbolic values must be given distinct codes are considered. These minimization problems can be parameterized along three major axes depending on the degrees of freedom permitted. These dimensions corresponds to whether 1) symbolic inputs, 2) symbolic outputs, or 3) multiple output choices are considered. The Cartesian product of these three degrees of freedom forms a problem space of eight possible minimization problems. In Figure 2.1, a lattice is depicted to show the relationships between these different minimization problems. A directed arrow in the diagram indicates that the minimization problem at the head of the arrow subsumes the minimization problem at the tail. The simplest is the minimization of Boolean functions where neither symbolic inputs, symbolic outputs, or multiple output choices is considered. The most general and the most difficult problem is the minimization of relations with both symbolic inputs and outputs since all three degrees of freedom are permitted.

In the case of Boolean function minimization, very mature methods have been developed [12]. The problems of minimizing functions with symbolic inputs and/or symbolic outputs are the classical symbolic minimization problems for which the techniques presented in [70, 95, 78] can be used to handle functions with symbolic inputs and the techniques presented in [68, 92, 35] can be used to handle functions with both symbolic inputs and outputs[7].

Minimizing relations is more difficult. Techniques for minimizing Boolean relations and Boolean relations with symbolic inputs (*i.e.*, a symbolic-input-binary-output relation) have been developed. An exact procedure for minimizing Boolean relations was presented in [14]. Recently, heuristic Boolean relation minimizers have also been proposed [93, 41]. These minimizers consider only binary variables. More recently, the heuristic minimizer described in [93] was generalized to handle symbolic inputs [94]. However, symbolic outputs were not

[7] In this section, for the sake of comparisons, no distinction is made between the problems of symbolic minimization and encoding.

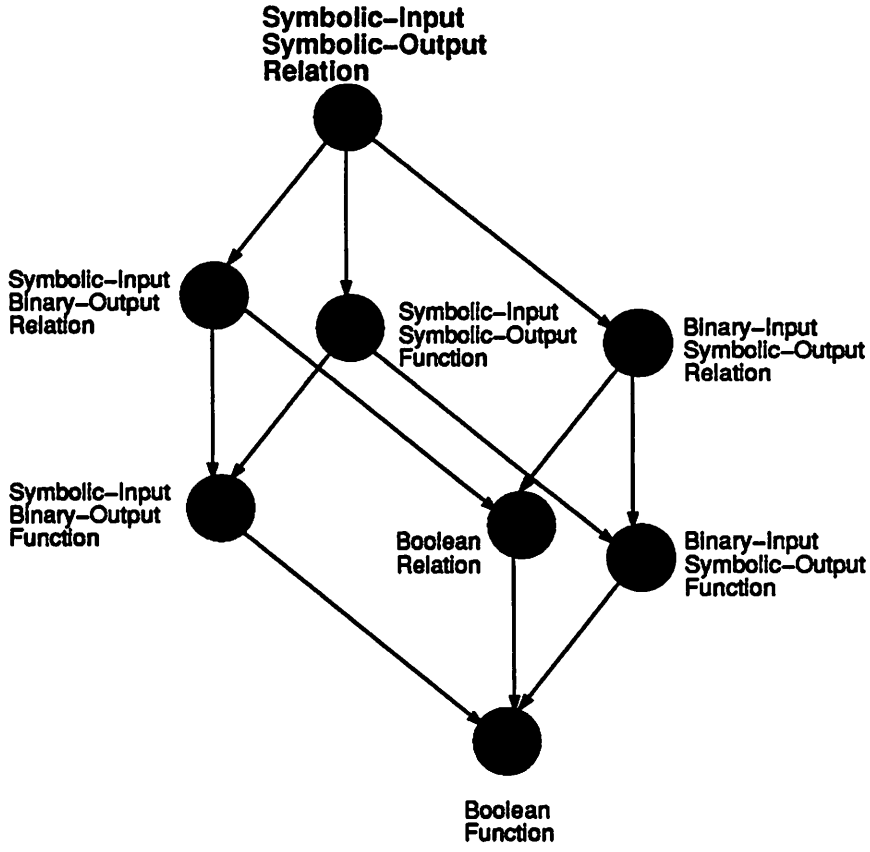


Figure 2.1: Lattice of Symbolic Minimization Problems.

considered[8].

When formulating the FSM synthesis problem as a symbolic relation problem, the final codes assigned to the states need not be orthogonal if knowledge of equivalent states is known. In particular, the freedom of assigning identical codes to equivalent states should be considered in order to fully exploit the presence of equivalent states. This effectively corresponds to performing state minimization implicitly. However, one must take care to

[8]In [94], a symbolic-input-binary-output relation was referred to as a multiple-valued relation. The multiple-valued relation minimizer handles symbolic outputs by implicitly *one-hot* encoding the symbolic outputs, and hence it does not fully exploit the degrees of freedom permitted with symbolic outputs.

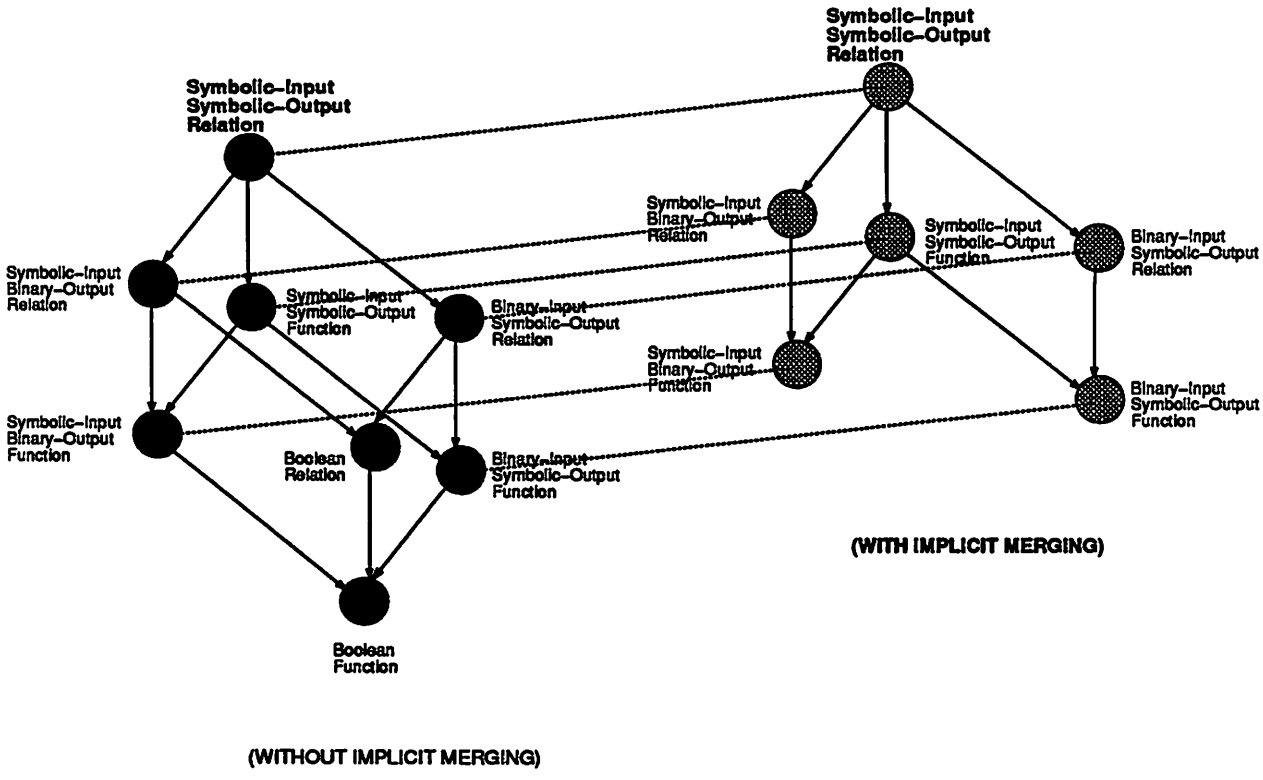


Figure 2.2: Symbolic Minimization Problems with Implicit Merging.

ensure that the merging of states is consistent. Nonetheless, permitting **implicit merging** in the problem definition adds a fourth dimension to the problem space. Each minimization problem with symbolic variables can be generalized to a broader problem that considers the assignment of identical codes with constraints. The relationships between these minimization problems is depicted in Figure 2.2. The formulation of FSM synthesis as one of symbolic relation minimization is addressed further in the next section.

2.3.2 FSM Synthesis: Restructuring and State Assignment

The classical approach to synthesizing finite state machines from a state transition graph model of behavior has been by means of a state assignment step. The goal of state

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

(a) Machine M_1

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

(b) Machine M_2

S	$Codes$
s_1	00
s_2	01
s_4	10

(c) Code S_2

$I \times S$	$S \times O$
000	011
100	010
–01	101
–10	000

(d) Logic N_2

Figure 2.3: Limitation of State Assignment.

assignment is to optimally assign binary codes to the internal states of a finite state machine such that the resulting synthesized logic is minimized. The assignment is restricted such that no two states are assigned the same binary code. While effective state assignment methods exist, they generally do not explore possible state assignments from other equivalent state machine structures. Therefore, the solutions obtainable by means of standard state assignment may be suboptimal. That is, a better realization may have been possible from an equivalent machine with a different structure. In general, two machines can exhibit the same overall terminal behavior even though their respective structures are different. Equivalent state machines can be realized by merging or splitting states. Such transformations are referred to as **machine restructuring**. Consider the following example:

Example 2.1 Referring to Figure 2.3, logic implementation N_2 is a valid implementation of machine M_1. However, one cannot find a conventional state assignment that will result in such a logic implementation. □

N_2 is a valid implementation of machine M_1 because it corresponds to a state assignment from an equivalent machine M_2. Thus, even if optimum state assignment can be found for a given machine specification, there may still exists a superior valid implementation that can only be obtained if other equivalent machines are considered.

In the above example, M_2 was obtained by state minimizing M_1. In general, different equivalent machines can be derived by splitting or merging states. In effect, equivalent states are introduced or eliminated.

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

(a) Machine M_1

merging ⇒
⇒ splitting

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

(b) Machine M_2

Figure 2.4: Splitting and Merging of States.

Example 2.2 Referring to Figure 2.4, both machines M_1 and M_2 are equivalent. Machine M_2 can be derived from machine M_1 by merging the states s_2 and s_3 of M_1 into a single state s_2 in M_2. Similarly, machine M_1 can be derived from machine M_2 by splitting the state s_2 in M_2 into two states s_2 and s_3 of M_1. Machine M_2 is a reduced machine and machine M_1 is a non-reduced machine. □

One commonly used method of restructuring is state minimization[54]. In this process, equivalent states are systematically identified and merged. The primary objective of state minimization is to obtain an equivalent machine with the minimal number of states. However, it is well known that a non-reduced machine may in fact lead to a superior logic implementation[47]. This point may be better appreciated in the following example (taken from [47]).

Example 2.3 Consider the two machines shown in Figure 2.5. Machine M_1 is a reduced machine (ie. it has no equivalent states). Machine M_2 is not a reduced machine since the states s_3 and s_4 are equivalent. Machine M_2 was derived from M1 by splitting the state s_3 of M_1 into s_3 and s_4 in M_2. The optimum two-level result for the reduced machine M_1 is eight product terms, while optimum result for the non-reduced machine M_2 is only seven product terms. □

Therefore, it is crucial to explore possible state assignments not from just one machine specification, but from a number of equivalent ones.

Example: (taken from Hartmanis, 1962)

I	S	S	O
0	s_0	s_2	0
1	s_0	s_6	0
0	s_1	s_3	0
1	s_1	s_7	0
0	s_2	s_0	0
1	s_2	s_5	0
0	s_3	s_1	0
1	s_3	s_3	0
0	s_5	s_0	1
1	s_5	s_1	1
–	s_6	s_3	0
–	s_7	s_2	0

(8 terms)

(a) reduced

I	S	S	O
0	s_0	s_2	0
1	s_0	s_6	0
0	s_1	s_3	0
1	s_1	s_7	0
0	s_2	s_0	0
1	s_2	s_5	0
0	s_3	s_1	0
1	s_3	s_4	0
0	s_4	s_1	0
1	s_4	s_3	0
0	s_5	s_0	1
1	s_5	s_2	1
–	s_6	s_3	0
–	s_7	s_2	0

(7 terms)

(b) non-reduced

Figure 2.5: Limitation of State Minimization.

In this research, it is proposed that the state minimization and state assignment problems be solved together by formulating the combined problem as a general case of symbolic relations. Precisely, the problem is formulated as one of **minimizing a symbolic-input-symbolic-output relation with implicit merging.** To capture the effects of implicit merging, new constraints must be imposed on the covering problem, as described in Section 2.6.3. However, as will be seen in the next section, these new constraints can be handled uniformly using a unified framework. The main result of formulating the FSM synthesis problem as one of symbolic relation minimization under implicit merging is that the formulation captures an entire family of equivalent state machines that can be obtain by state minimization and re-direction of state transitions simultaneously. Each of these steps is elaborated in greater detail in remainder of the chapter.

2.4 A Unified Framework

A unified framework is now presented for solving various two-level symbolic relation minimization problems exactly. The approach is based on the classical paradigm of prime implicant generation and covering, but generalized to minimize symbolic relations. The overall exact minimization process involves the following three steps.

1. **Prime Generation:** In this step, all the prime implicants of the symbolic relation are generated. To handle symbolic relations, the classical notion of a prime implicant must be extended. In [35], Devadas *et al.* introduced the concept of generalized prime implicants (GPI's) for minimizing symbolic functions. To handle multiple output choices in the symbolic relation problem, the notion of GPI's is extended to the concept of **generalized candidate primes** (or simply generalized c-primes), similar to the notion of candidate primes in the Boolean relation problem [14]. Depending on the degrees of freedom considered (*e.g.*, symbolic inputs, symbolic outputs, or multiple output choices), different sets of generalized candidate primes must be generated. However, the same prime generation procedure can be used, as described in Section 2.5.

2. **Constraint Generation:** After the generalized c-primes have been generated, a covering problem must be solved whereby a subset of generalized c-primes is selected to implement the symbolic relation. Unlike classical Boolean minimization, constraints must be imposed on the selection process such that the selected set of generalized c-primes can be encoded and does actually realize a compatible mapping of the relation. To handle arbitrary constraints, the covering step is formulated as a single binate covering problem. These constraints include encoding constraints, selection constraints, and general constraints like the all-0 code problem. For different symbolic relation minimization problems, the constraints are modified appropriately.

3. **Binate Covering:** To solve the binate covering problem, which is also called a **minimum-cost satisfiability problem**, a binary decision diagram (BDD) based formulation is proposed whereby a BDD is built for a Boolean formula that implicitly captures all possible selections of generalized c-primes that satisfy all the imposed constraints. It is shown in Section 2.7 that the minimum cost solution actually corresponds to the shortest weighted path in the BDD, which can be found in linear time

if the corresponding BDD can be built (*i.e.*, $O(|V|)$ where $|V|$ is the size of the BDD).

2.5 Prime Generation

2.5.1 Candidate Primes for Boolean Relations

Before addressing the problem of prime generation for symbolic relations, prime generation for the restricted case of Boolean relations is first reviewed. The idea behind the use of relations in optimization problems is to anticipate all possible choices. For this reason, in Boolean relations one tries to anticipate all possible combinations of output values. One method for deriving all primes for a Boolean relation $\mathcal{R} \subseteq B^r \times B^n$ is to write down all mappings compatible with $\mathcal{R}$ and generate all the primes for each of these mappings. However, if minterm $\mathbf{x} \in B^r$ has $m_{\mathbf{x}} = |\mathcal{R}(\mathbf{x})|$ choices, then there are $\prod_{\mathbf{x} \in B^r} m_{\mathbf{x}}$ different compatible mappings. In [14], a prime generation procedure for Boolean relations was given that can produce the complete set of primes without having to enumerate all mappings compatible with $\mathcal{R}$ explicitly. The exposition of this procedure requires some definition first (a more detailed explanation of the theory and algorithms can be found in [14]).

Definition 2.10 *A* **candidate prime** *(or c-prime) of a Boolean relation $\mathcal{R}$ is a prime of a mapping $\mathbf{f} \prec \mathcal{R}$.*

Definition 2.11 *A* **cube,** *$C \subseteq B^r \times B^n$, is a Boolean relation, denoted by the pair $C = (\mathbf{c}|I)$, such that $\mathbf{c}$ is a cube of B^r, $I \subseteq B^n$, and*

$$C(\mathbf{x}) = I, \forall \mathbf{x} \in \mathbf{c}$$
$$C(\mathbf{x}) = \{0\}, \forall \mathbf{x} \notin \mathbf{c}$$

$\mathbf{c}$ is called the **support set** *of C and I is called the* **influence set** *of the cube. The* **size** *of C is $|\mathbf{c}|$.*

Definition 2.12 *Let $\mathcal{R} \subseteq B^r \times B^n$ be a Boolean relation, and $C \subseteq B^r \times B^n$ be a cube, C is an* **implicant** *of $\mathcal{R}$ if and only if*

$$\forall \mathbf{x} \in B^r, \forall \mathbf{y}' \in C(\mathbf{x}), \exists \mathbf{y} \in \mathcal{R}(\mathbf{x}) \text{ such that } \mathbf{y}' \leq \mathbf{y}.$$

(note: $\mathbf{y}' \leq \mathbf{y}$ means $\mathbf{y}$ bit-wise contains $\mathbf{y}'$.) A **prime implicant,** *$(\mathbf{c}|I)$, of $\mathcal{R}$ is a cube with the property that $i \in I$ if and only if $(\mathbf{c}|i)$ is a c-prime of $\mathcal{R}$.*

Definition 2.13 *A* **fundamental implicant** *of a Boolean relation $\mathcal{R} \subseteq B^r \times B^n$ is an implicant, $(\mathbf{x}|I)$, where $\mathbf{x} \in B^r$ and $I = \mathcal{R}(\mathbf{x})$.*

Definition 2.14 *Two cubes, C' and C'', of $\mathcal{R}$ are* **adjacent** *if and only if their supports $\mathbf{c}'$ and $\mathbf{c}''$ are distance one (differ in only one variable) from each other. The* **merge** *(denoted $C' \circ C''$) of C' and C'' is a cube C with support set*

$$\mathbf{c} = \mathbf{c}' \cup \mathbf{c}''$$

and influence set

$$I = \{\mathbf{y} \in B^n : \mathbf{y} = g.l.b.(\mathbf{y}', \mathbf{y}''), \mathbf{y}' \in I', \mathbf{y}'' \in I''\}$$

where g.l.b stands for the greatest lower bound, the bit-wise AND *of $\mathbf{y}'$ and $\mathbf{y}''$.*

The procedure for generating c-primes proposed in [14] is based on iterated consensus. It is summarized in Figure 2.6 (taken directly from [14]). The procedure begins with a set of fundamental implicants and iteratively construct larger cubes. At each iteration, the set of maximal implicants of size s are produced. Also at each iteration, implicants from the previous iteration that are contained by the current set of implicants are removed. A proof of correctness can be found in [17].

2.5.2 Generalized Candidate Primes for Symbolic Relations

When symbolic variables are considered, provisions must be made for all possible combinations of code assignments and output choices. Recall that given a symbolic relation $\mathcal{R} \subseteq D \times \Sigma$, a multi-output Boolean function $f : B^r \rightarrow B^n$ is said to be a compatible mapping for $\mathcal{R}$ if there exists an input encoding $\xi : D \rightarrow B^r$ and an output encoding $\psi : \Sigma \rightarrow B^n$ such that $\forall \mathbf{x} \in D$, $\exists \mathbf{y} \in \mathcal{R}(\mathbf{x})$ such that $f(\xi(\mathbf{x})) = \psi(\mathbf{y})$ (denoted by $f \prec_{\{\xi,\psi\}} \mathcal{R}$). Therefore, all primes corresponding to all possible compatible mappings must be considered. This leads to the following definitions of generalized fundamental implicant and generalized c-primes.

Definition 2.15 *A* **generalized fundamental implicant** *of a symbolic relation $\mathcal{R} \subseteq D \times \Sigma$ is an implicant $(\mathbf{x}|\sigma)$, where $\mathbf{x} \in D$ and $\sigma = \mathcal{R}(\mathbf{x})$.*

Definition 2.16 *A* **generalized candidate prime** *(or simply generalized c-prime or gc-prime) of a symbolic relation $\mathcal{R}$ is a cube $(\mathbf{c}|\sigma) \subseteq D \times \Sigma$ such that there exists an input encoding $\xi : D \rightarrow B^n$ and an output encoding $\psi : \Sigma \rightarrow B^n$ for which $(\xi(\mathbf{c})|\psi(\sigma))$ is a prime of a mapping $\mathbf{f} \prec_{\{\xi,\psi\}} \mathcal{R}$.*

```
input: The set of fundamental implicants of R
output: The set P of all prime implicants of R

A_0 = {all fundamental implicants of R};
P = ∅;
for s = 1,...,r {
    B_s = ∅;
    for each pair (a', a'') ∈ A_{s-1} × A_{s-1}, a' ≠ a'' {
        if a' is adjacent to a'' {
            b = a' ∘ a'';
            mark all i' ∈ I' such that ∀i'' ∈ I'', i' ≤ i'';
            mark all i'' ∈ I'' such that ∀i' ∈ I', i'' ≤ i';
            B_s = B_s ∪ {b};
        }
    }
    remove all marked vertices from the influence sets of the cubes in A_{s-1};
    P = P ∪ A_{s-1};
    A_s = B_s;
}
P = P ∪ A_r;
```

Figure 2.6: Prime Generation Procedure for c-primes of Boolean Relations.

These definitions can be trivially extended when multiple symbol sets are considered. In the remainder of the chapter, generalized c-primes will occasionally be simply referred to as primes where no confusion arises. The fundamental concept behind the minimization of symbolic relations is similar for Boolean relations, *viz.* finding all the candidate primes and then forming a covering problem.

In [35], it was shown that generalized primes for symbolic functions can be generated using already existing prime implicant generation procedures. This is accomplished by transforming a symbolic truth table into a multi-valued input multi-output binary-valued output function. Precisely, the output symbols are assigned *0-hot* codes [35]. Given N symbolic values, N bits are used (*i.e.*, σ_1 is encoded as $0111\cdots1$, σ_2 as $1011\cdots1$, and so on). It was shown that the prime implicants of this new multi-output function have a one-to-one

correspondence with the generalized primes of the original symbolic function. A proof can be found in [35]. This transformation is independent of the eventual code-length to encode the symbolic values.

The procedure for generating generalized c-primes in the symbolic relation problem is similar, but it must account for multiple choices of output symbols. The generation of generalized c-primes proceeds as follows. First, the output symbols are 0-hot encoded as in the case for symbolic functions. If the output for a particular input value is a complete don't-care, as in the case when a given input value cannot take place, then the all-one code is used. The resulting Boolean relation is then submitted to the c-prime generation procedure described in Figure 2.6 [14], modified to take into account the following results. Since a Boolean relation c-prime generation procedure is used, the following result can be used to remove locally-redundant c-primes. At the end of the process, primes with the all-one output part are discarded, as they only cover vertices for which the output is completely free.

Theorem 2.1 *Generalized c-primes with all-one output part can be discarded.*

Proof: Using the 0-hot encoded scheme, an all-one output part means the corresponding cube does not assert any output symbols. Therefore, it cannot be used to form a minimal cover. ■

For theses primes, no constraints need be generated for the minimum-cost covering problem. Constraints are required for all other primes for which some restrictions apply. Depending on the problem, different constraints must be satisfied in achieving a minimum-cost cover.

If eventually the selection of primes will be encoded using a fixed code-length L, then it is possible to discard some primes already before solving the minimum-cost covering problem.

Lemma 2.2 *The intersection of $n > 2^{L-1}$ codes of L bits is the all-zero code.*

Proof: For every bit in the code there are at most 2^{L-1} codes with a 1 in that position. ■

The application of this lemma is straightforward. When a new cube is generated (in Figure 2.6), the number of zeroes in its output part is counted. If it is greater than 2^{L-1}, then the cube is marked as non-prime.

Example 2.4 Let $\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$. Let $L = 2$. A cube with output part $\sigma_1 \cap \sigma_2 \cap \sigma_3$ (output part represented as 0001) can be dropped immediately. □

2.5.3 Reduced Prime Implicant Table

The generalized c-prime generation procedure can be further optimized by considering the construction of a *reduced generalized c-prime table.* In classical Boolean minimization, many techniques have been developed for determining all the prime implicants [67, 12, 77]. Unlike Quine-McCluskey's (Q-M) procedure [67] where a row is introduced for each minterm, minimizers like ESPRESSO-EXACT [77] produces a reduced prime implicant table where each row corresponds to a collection of minterms (ie. a larger subspace), all of which are covered by the same set of prime implicants. In principle, this general idea can also be applied in the case of relations to reduce the size of the table. However, whether the prime generation procedure for Boolean relations, described in Figure 2.6 can be extended to exploit this idea remains an open question.

2.6 Exact Symbolic Relation Constraints

After the generation of generalized c-primes, a constrained covering problem is solved to select a suitable subset of generalized c-primes. This is formulated as a binate covering problem. As a by-product, the encoding of the symbols are derived automatically. In this section the generation of constraints, expressed as Boolean formulas, is considered in detail for three forms of the symbolic relation problem that involve symbolic outputs and multiple output choices. They naturally correspond to three encoding problems: output encoding for two-level combinational circuits, state encoding for two-level implementations of finite state machines, and simultaneous state minimization and encoding for two-level implementations of finite state machines. In describing the constraints for these three problems, relations are assumed.

2.6.1 Output Encoding

First, the problem of minimizing symbolic relations with only binary inputs is considered. This is referred to as the output encoding problem. More formally, the problem is stated as follows. Given a symbolic relation $\mathcal{R} \subseteq B^r \times \Sigma$, where $B = \{0, 1\}$ and $\Sigma =$

$\{\sigma_1, \ldots, \sigma_N\}$, encode Σ with at most L bits, so that $\sigma_k = b_{k1}b_{k2}\cdots b_{kL}$. There are 2^L possible codes and the inequality $N \leq 2^L$ must be satisfied. The encoding must be such that the number of product terms in the minimized relation is minimum. In this problem, a fixed code length L is given.

The satisfaction of two sets of constraints are required for the output encoding problem. One set of constraints is required for all vertices for which some restrictions on the output values apply (corresponding to constraints on encoding and multiple output choices), and the other set is used to express the fact that no two symbols may have the same code[9]. These constraints are expressed as follows.

Output Constraints: Consider a minterm $\mathbf{x} \in B^r$. Without loss of generality, suppose the image (the possible output mappings) of $\mathbf{x}$ is $\mathcal{R}(\mathbf{x}) = \{\sigma_1, \ldots, \sigma_p\}$. Let $g_1, \ldots, g_n$ be all the gc-primes that cover minterm $\mathbf{x}$. Let $\prod_{j=1}^{q_i} \sigma_{i_j}$ be the output part of g_i. Let $I = \{1, \ldots, n\}$ and

$$I_k = \{i | i \in I \wedge (\exists j)(\sigma_{i_j} = \sigma_k)\}$$

In other words, I_k is the index set of the gc-primes covering $\mathbf{x}$ that contain σ_k in their output parts. Then the following output covering constraints must be satisfied.

$$\sum_{k=1}^{p} \left(\left(\prod_{i \in (I - I_k)} \overline{g}_i \right) \left(\left(\sum_{i \in I_k} \left(g_i \left(\prod_{j=1}^{q_i} e(\sigma_{i_j}) \right) \right) \right) \equiv e(\sigma_k) \right) \right) = 1,$$

where $e(\sigma_{i_j})$ and $e(\sigma_k)$ represent the encodings given to σ_{i_j} and σ_k, respectively. These can be rewritten as

$$\sum_{k=1}^{p} \left(\left(\prod_{i \in (I - I_k)} \overline{g}_i \right) \prod_{l=1}^{L} \left(\left(\sum_{i \in I_k} \left(g_i \left(\prod_{j=1}^{q_i} b_{i_j l} \right) \right) \right) \overline{\oplus}\, b_{kl} \right) \right) = 1,$$

and simplified to

$$\sum_{k=1}^{p} \left(\left(\prod_{i \in (I - I_k)} \overline{g}_i \right) \prod_{l=1}^{L} \left(\overline{b}_{kl} + \sum_{i \in I_k} \left(g_i \left(\prod_{j=1, i_j \neq k}^{q_i} b_{i_j l} \right) \right) \right) \right) = 1. \qquad (2.2)$$

Here, the g_i's represent the selection variables corresponding to the gc-primes, and the b_{ij}'s represent the encoding variables (*i.e.*, b_{ij} corresponds to the j-th encoding bit of the i-th symbolic value).

[9]This restriction can be relaxed if two symbols are *equivalent*. In which case, they can be given the same code (*cf.* Section 2.6.3) if the required constraints are satisfied.

Disjointness Constraints: The disjointness constraints are imposed to ensure that the final codes are orthogonal and can be simply expressed for L bits and N symbolic values as:

$$\prod_{i=1}^{N-1} \prod_{j=i+1}^{N} \sum_{k=1}^{L} (b_{ik} \oplus b_{jk}) = 1. \tag{2.3}$$

The derivation of these constraints is similar to the one presented in [35], with three major points of departure.

1. The introduction of the variables g_i makes it possible to cast the whole problem into a single, uniform, binate covering problem.

2. The solution of the **all-zero state** problem is incorporated in the formulation. This can be seen by observing that in Equation 2.2, the requirement that a prime be selected to cover a vertex, expressed by

$$\sum_{i \in I_k} \left(g_i \left(\prod_{j=1, i_j \neq k}^{q_i} b_{i_j l} \right) \right),$$

is implied by $b_{kl} = 1$. This result is quite significant since the all-zero code problem was previously handled by performing $N + 1$ exact minimizations, where N is the number of symbolic values [35]. No other approach was previously known. Clearly, performing $N + 1$ exact minimizations is extremely CPU intensive.

3. As the problem here is the minimization of relations rather than functions, one has to account for the fact that the simultaneous selection of a set of primes may be incorrect. These restrictions are embodied in the term

$$\left(\prod_{i \in (I - I_k)} \overline{g}_i \right)$$

of Equation 2.2.

These constraint can be put in product-of-sum (POS) form or, if the BDD formulation of the binate covering problem is used, the BDD can be built directly from these constraints. The weights are as follows:

- weight(g_i) = 1;
- weight(b_{ij}) = 0;

The minimum cost assignment using the above weights that satisfies both Equation 2.2 and Equation 2.3 corresponds exactly to the minimum two-level solution to the output encoding problem.

2.6.2 State Encoding

In the exact state encoding problem, symbolic inputs must be considered in addition to symbolic outputs. The achievement of the minimum number of terms for the exact state encoding problem requires the satisfaction of **face embedding constraints** [70] in addition to those imposed by exact output encoding. These face embedding constraints are derived from the primes of the symbolic relation describing the state transition graph. The primes are derived taking into account that the present-state field is a single symbolic variable. Unlike the previous approaches, the entire set of face embedding constraints that correspond to all the primes of the relation is considered. However, the covering constraints are form in such a way that a given face embedding constraint will have to be satisfied only if some primes are selected.

Face Embedding Constraints: More specifically, there are as many non-trivial face embedding constraints as there are unique present-state parts in the primes, having eliminated the present-state parts that cover either one or all states. These last present-state parts are known to represent trivially-satisfied face embedding constraints.

For each non-trivial face embedding constraint ϕ there is an associated group of primes. Let $\{g_1, \ldots, g_n\}$ be this set of primes. Furthermore, let $\sigma_{t_1}, \ldots \sigma_{t_T}$ be the states that ϕ requires to be in the same face and let $\sigma_{r_1}, \ldots \sigma_{r_R}$ be the states that ϕ requires not to be in that face. The following covering constraints must be satisfied:

$$\prod_{i=1}^{n} \overline{g}_i + \prod_{k=1}^{R} \left(\sum_{i=1}^{L} \prod_{j=1}^{T} (b_{t_j i} \oplus b_{r_k i}) \right) = 1. \tag{2.4}$$

Equation 2.4 must be satisfied in addition to Equation 2.2 and Equation 2.3 to obtain the minimum-cost solution to state encoding. The exact solution corresponds to the minimum implementation for entire family of FSM's under state encoding and equivalent next-state transitions.

2.6.3 State Minimization and State Encoding

As described in Section 2.3.2, the two tasks of state minimization and state encoding for completely specified finite state machines can be solved **exactly** as a modified symbolic relation problem as follows:

1. Apply a standard procedure [54, 44] for identifying equivalent states and **implied state pairs**. This process can be performed with $O(N\ log\ N)$ time complexity where N is the number of states. Then construct an **implication graph** $\mathcal{G}(V,E)$ where each $v \in V$ corresponds to an equivalent state pair $(\sigma_{v1}, \sigma_{v2})$ and each directed edge $e : u \rightarrow v$ between two vertices corresponds to a merging constraint whereby the merging of state pair $(\sigma_{u1}, \sigma_{u2})$ **implies** the merging of state pair $(\sigma_{v1}, \sigma_{v2})$.

2. Convert the state encoding problem into a symbolic relation problem by permitting the next-state of any transition to be any one of the equivalent states, as in the exact encoding problem.

3. Solve the symbolic relations problem modified to permit equivalent states to have the same code and ensure all implied conditions are satisfied.

To solve exact state encoding under state minimization, the covering constraints must be modified to take into account the following two conditions: 1) Equivalent states need not have orthogonal codes. The disjointness constraints of Equation 2.3 can be modified to account for this flexibility. 2) If two states are implicitly merged (by assigning the same code), then all the **implied state pairs** must also be merged. This requires the addition of a set of constraints called **implied equivalence constraints.** These constraints are given below.

Modified Disjointness Constraints: A modified set of disjointness constraints must be imposed to ensure that the final codes for non-equivalent states are orthogonal. The codes given to equivalent states however need not be orthogonal. This can be simply expressed for L bits and N symbolic values as:

$$\prod_{i=1}^{N-1} \prod_{j=i+1, \sigma_i \not\equiv \sigma_j}^{N} \sum_{k=1}^{L} (b_{ik} \oplus b_{jk}) = 1. \tag{2.5}$$

Implied Equivalence Constraints: Another set of constraints is imposed to guarantee that if two states are implicitly merged, then all implied state pairs are also merged. This is expressed as follows. Let (σ_i, σ_j) be an equivalent state pair, and $\{(\sigma_{p_1}, \sigma_{q_1}), \ldots, (\sigma_{p_I}, \sigma_{q_I})\}$ be the set of I implied equivalence pairs. Then

$$\prod_{r=1}^{I} (e(\sigma_i) \equiv e(\sigma_j)) \Rightarrow (e(\sigma_{p_r}) \equiv e(\sigma_{q_r})) \;=\; 1.$$

must be satisfied for every equivalent state pair. These can be rewritten as:

$$\prod_{r=1}^{I} \left(\sum_{k=1}^{L} (b_{ik} \oplus b_{jk}) + \prod_{k=1}^{L} (b_{p_r k} \,\overline{\oplus}\, b_{q_r k}) \right) \;=\; 1. \tag{2.6}$$

The minimum cost assignment under the constraints of Equation 2.2, Equation 2.5, and Equation 2.6 corresponds exactly to the minimum two-level solution to the state minimization and state encoding problem. This is complete for an entire family of FSM's under state encoding, state merging, and state transition re-direction.

2.7 Solving the Binate Covering Problem

2.7.1 The Problem

Solving the binate covering problem efficiently has many important applications in logic and sequential synthesis. They include Boolean relation minimization, state minimization, exact encoding, and technology mapping. It is well known that minimum cost binate covering can be formulated as a 0-1 integer linear program (ILP). However, optimum solution to the ILP problem is known to be intractable. Alternatively, binate covering is better viewed as a **minimum-cost satisfiability problem** where effective logic manipulation techniques may be applied. For clarity, a statement of the minimum binate covering problem is given.

Problem 2.1 (The Minimum Binate Covering Problem) Let the Boolean formula $T(x_1, \ldots, x_n)$ represent the covering constraints where an input assignment $\mathbf{x}$ is said to be a satisfying assignment if and only if $T(\mathbf{x}) = 1$. Let cost of a positive literal x_i be given by w_i, and the cost of a negative literal $\overline{x_i}$ be 0. Find a minimum cost assignment $\mathbf{x}$ satisfying T, *i.e.*, find an assignment $\mathbf{x}$ such that $T(\mathbf{x}) = 1$ and $\sum_{i=1}^{n} x_i \cdot w_i$ is **minimum.** □

This is referred to as the **general** or **factored form** binate covering problem. One approach to the binate covering problem is to express the constraints in POS form, also called the conjunctive normal form, and apply branch-and-bound techniques to solve the problem. This instance of the problem is referred to as the **two-level** binate covering problem. When formulated in this form, effective algorithms have been developed for exploring the search space [14, 78, 44]. In the POS form, the covering constraints T can be written in a binary matrix form with coefficients from the set $\{0, 1, 2\}$, and the problem is formulated in the following way.

Problem 2.2 Find a subset C of columns of minimum cost, where the cost of selecting column c_i is w_i, such that for every row r_k, either

1. $\exists j : T_{kj} = 1 \wedge c_j \in C$ or
2. $\exists j : T_{kj} = 0 \wedge c_j \notin C$.

□

In this section, a branch-and-bound strategy for solving the binate covering in POS form is first addressed. Empirically, the techniques for two-level binate covering are quite effective. However, a limiting factor is that the covering problem must be expressed in two-level POS form. This is not always easy (if even possible) for many interesting applications. This motivates the need to explore binate covering techniques that attack the factored form representation directly. The exact symbolic relation problem being solved in this chapter is an example of such application. Instead, a graph-oriented formulation based on binary decision diagrams (BDD) [20] is proposed in Section 2.7.3. Pruning strategies are described in Section 2.7.4 for expediting BDD construction.

2.7.2 Branch-and-Bound Techniques

The efficiency of binate covering in POS form depends heavily on the pruning and branching strategies used. Here, direct extensions to the implementation describe in [14] for reducing the covering matrix and bounding the search are described. First, the reduction rules described in [14] are summarized.

Definition 2.17 *An* **essential** *row of T is a row r_i where only one coefficient is different from* 2.

Essential rows can be removed. The corresponding column c_j is included in the selection set S if $T_{ij} = 1$. Column c_j can then be removed.

Definition 2.18 *A row r_i* **dominates** *another row r_l if r_i is satisfied whenever r_l is satisfied.*

Dominating rows can be eliminated. Row dominance corresponds to single cube containment.

Definition 2.19 *Let c_j and c_k be two columns of T. Then c_j* **dominates** *c_k if, for each row of T, $T_{ij} = 1$ or $T_{ij} = 2$ and $T_{ik} \neq 1$ or $T_{ij} = 0$ and $T_{ik} = 0$.*

The dominated column can be eliminated. These reduction rules are essentially the ones proposed by Grasselli and Luccio [44]. An additional reduction rule that can be applied is directly adapted from a paper by House and Stevens [48].

Theorem 2.3 *Let T be satisfiable. Let c_k be a column of T. Suppose for some rows r_i and r_j, $T_{ik} = 1$ and $T_{jk} = 0$, and $T_{jl} = 1$ implies $T_{il} = 1$. Then a minimum solution to the modified matrix by setting $T_{ik} = 2$ is also a minimum solution to the original matrix.*

This is essentially a restricted form of consensus where the clauses are simplified. Like logic minimization, the overall simplification is order dependent. This technique can be effective in some cases. However, note that the simplification of constraints interplays with other reduction steps and the column selection algorithm.

All bounding techniques presented can be used to reduce the size of the covering matrix. In general, branching heuristics are required to explore the search space. In [14], **maximal independent sets** were used to give a lower bound. Columns that intersect many short rows were favored. In the symbolic relation minimization problem, a large number of the variables (those corresponding to encoding bits) have zero cost. These variables are introduced to impose additional selection constraints and do not contribute to the final cost. A possible strategy is to use the same branching heuristics as before, but defer selecting the zero cost columns as late as possible. When the only branching columns are zero cost columns, then the problem degenerates to simply a satisfiability problem. The maximal independent set method should be able to quickly bound the recursion.

2.7.3 BDD-Based Formulation of Binate Covering

An alternative approach to the binate covering problem is to formulate it with binary decision diagrams (BDD's) (*cf.* Section 2.2.5). The essence of the BDD approach can be captured in the following definition and theorem.

Definition 2.20 *The cost of a path in a BDD, D, is defined to be the sum cost of the arcs along the path where a '0' arc costs 0 and a '1' arc costs w (or 1 for an unweighted problem).*

Theorem 2.4 *Let D be a reduced (ordered) BDD a Boolean formula* $T(x_1, \ldots, x_n)$. *Then the minimum cost assignment satisfying T is given by the minimum cost (or shortest) path connecting the '1' leaf to the root in D* **independent of the choice of variable ordering.**

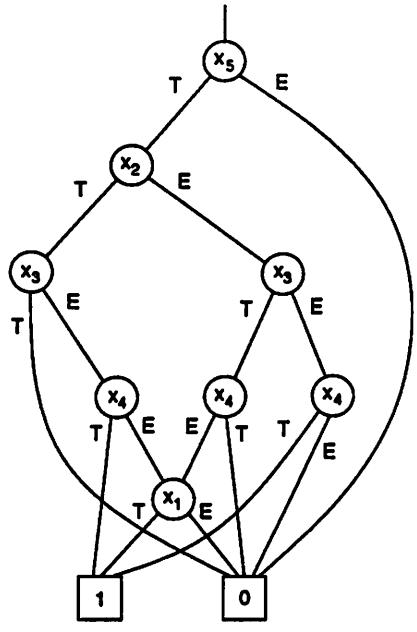
Proof: Every path from '1' to the root represents a set of satisfying assignments. By taking all the variables that do not appear along the path as 0, one obtains a unique minimal assignment corresponding to the path. For this unique assignment, the cost is equal to the weighted length of the path. Let $\mathbf{x}$ be the unique minimal assignment corresponding to the shortest path P of D. $\mathbf{x}$ is proved to be minimum. Suppose there is another assignment $\mathbf{x}'$ such that cost($\mathbf{x}'$)<cost($\mathbf{x}$). $\mathbf{x}'$ corresponds to at least one path from '1' to the root of D. Let P' be the shortest such path. Then cost($\mathbf{x}'$)$\geq$length(P')$\geq$length(P) = cost($\mathbf{x}$), a contradiction. ■

The above theorem states that binate covering problem can be solved by constructing the BDD for the Boolean formula T and solving the shortest path problem using Dijkstra's shortest path algorithm which is $O(E)$. For reduced BDD's the algorithm is also $O(V)$. The BDD can be constructed using a state-of-the-art BDD package.

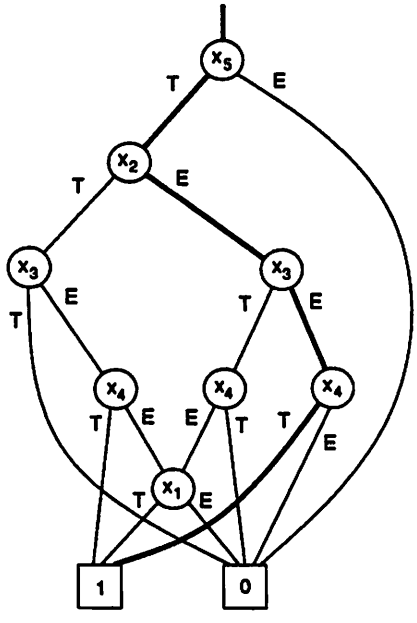
Example 2.5 Suppose the minimum cost solution to the following constraint formula is sought:

$$T = (\overline{x}_2 + \overline{x}_3)(\overline{x}_3 + \overline{x}_4)(x_2 + x_3 + x_4)(x_1 + x_3 + x_5)(x_1 + x_4)x_5.$$

Assuming a variable ordering of $x_5 \prec x_2 \prec x_3 \prec x_4 \prec x_1$, a BDD can be constructed for formula T, as shown in Figure 2.7(a). Assuming all the variables have a weight of 1 for x_i and 0 for $\overline{x_i}$, then the minimum cost binate covering solution can be derived from the shortest weighted path (*cf.* Figure 2.7(b)). Using the variable ordering $x_5 \prec x_2 \prec x_3 \prec x_4 \prec x_1$, a shortest path solution is $x_5\overline{x}_2\overline{x}_3x_4$. The variables that do not appear (x_1) are set to 0. This transformation leads to the minimum cost solution $\overline{x}_1\overline{x}_2\overline{x}_3x_4x_5$ (with cost equal to 2)



(a) A Binary Decision Diagram for T.



(b) A Corresponding Shortest Path Solution.

Figure 2.7: Binary Decision Diagram and Shortest Path Solution.

for satisfying T. If instead the variable ordering $x_5 \prec x_4 \prec x_1 \prec x_3 \prec x_2$ is used, a more compact BDD can be derived, as in Figure 2.8(a). Using this variable ordering, a shortest path solution is $x_5 x_4 \overline{x}_3$. Here, x_1 and x_2 are set to 0 and the same solution as before is obtained (*cf.* Figure 2.8(b)). □

Since the minimum cost solution is **independent** of the variable ordering, any existing variable ordering heuristics may be applied to derive a more compact BDD representation.

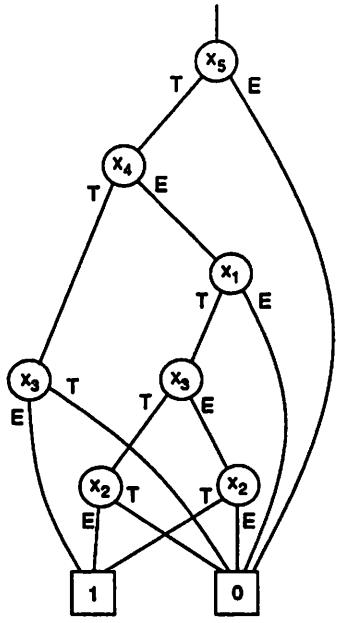
However, the straightforward approach to building the BDD is unnecessarily laborious. In essence, the BDD captures all possible input assignments. Paths leading to the '1' leaf represent valid assignments while paths to the '0' leaf represent invalid solutions. However, among the valid solutions, one is usually only interested in a potentially very small subset, those that have a low cost. Typically, a reasonably tight upper bound on the problem can be derived. For example, in technology mapping a fast tree-mapper can be invoked to obtain a tight bound. In state minimization, the minimum cost solution is bounded by the number of states and the number of maximal compatibles. In Boolean relations, a fast heuristic two-level minimization solution on a function that approximates the relation can provide a tight bound. If the equations can be written in POS form, then a quasi-greedy covering strategy can always be used to bound the solution cost. If an upper bound is provided, then more costly input assignments can be pruned. To do this, a new Boolean operator called the **threshold operator** is introduced.

2.7.4 The Threshold Operator

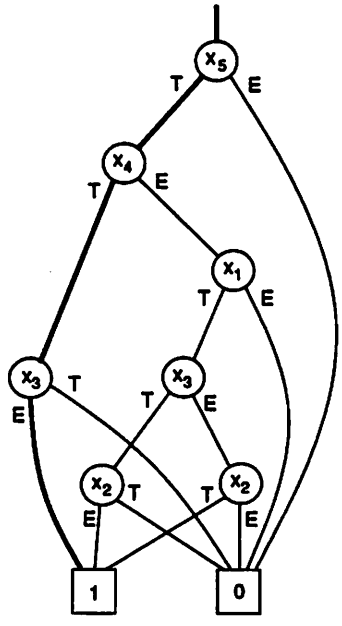
A naive approach to pruning away input assignments with cost greater than some upper bound u is to delete from the on-set of T those minterms with cost greater than u.

Lemma 2.5 *Let $T(x_1, \ldots, x_n)$ be the Boolean formula for the covering constraints, and u be an upper bound on the assignment cost. Let $G(x_1, \ldots, x_n)$ be a function where $G(\mathbf{x}) = 1$ if and only if the cost of $\mathbf{x}$ is less than or equal to u. Let $T_G = T \cap G$. Then minimum cost assignment for T_G, if one exists with a cost less than or equal to u, is the same as T.*

However, the Boolean formula G can be quite complicated. An alternative approach is to eliminate **paths** from the BDD whose path cost is greater than u. This is in fact what the threshold operator does.



(a) A Binary Decision Diagram for T using a Different Ordering.



(b) A Corresponding Shortest Path Solution.

Figure 2.8: Binary Decision Diagram and Shortest Path Solution using a Different Ordering.

Definition 2.21 *Let D be the BDD of a Boolean formula $T(x_1, \ldots, x_n)$, and u be an upper bound on the input assignment cost. Define $D' = \Lambda(D, u)$ to be a new BDD where there exists no path from root of D to '1' with cost greater than u and the minimum cost, and all paths with less than u are retained. All paths with cost greater than u are redirected to the '0' leaf.*

Theorem 2.6 *Let D be the BDD of a Boolean formula $T(x_1, \ldots, x_n)$ to a set of covering constraints, and u be an upper bound on the input assignment cost. Then the minimum path cost for $D' = \Lambda(D, u)$ is the same as the minimum path cost for D as long as the minimum cost is less than or equal to u.*

Proof: A satisfying assignment is one that leads from the root to the '1' leaf. All paths leading from the root to the '1' leaf of D' also exists in D. Since the only paths in D not in D' are those with a path cost greater than u, it is not possible to find a minimum cost path greater than u in D but not in D'. ■

Essentially, $\Lambda(D, u)$ restricts the assignments under consideration to those with cost less than or equal to u. The threshold operator can be used to discard irrelevant solutions. This is quite powerful since it can greatly reduce the final size of the BDD. However, BDDs are rarely built up at once. It is usually built bottom-up using the **apply** or **ITE** operators. Building the global BDD first before applying the threshold operator is not very effective since it may take a long time and a great deal of storage to build. It is next shown how the threshold operator can be used, in the case when $T = \prod_{i=1}^{m} t_i$, to build the global BDD bottom-up.

Theorem 2.7 *Let $D = \prod_{i=1}^{m} D_i$. Then $\Lambda(D, u) \subseteq \prod_{i=1}^{m} \Lambda(D_i, u)$.*

Proof: Theorem 2.6 states that any satisfying assignment with cost less than or equal to u is retained after the threshold operator. Let $\mathbf{x}$ be a satisfying assignment of D with cost less than or equal to u. By definition, $\mathbf{x} \in D \Rightarrow \mathbf{x} \in D_i, \forall i$. After the threshold operator, $\mathbf{x}$ is retained in $\Lambda(D, u)$ and $\Lambda(D_i, u), \forall i$, which implies $\mathbf{x}$ is also a satisfying assignment in $\prod_{i=1}^{m} \Lambda(D_i, u)$. ■

The application of threshold operator when building the global BDD could help to keep the size of the BDD compact. In many applications, it has been found that it is usually possible to build the BDD for each constraint. The problem arises when combining them

PS	NS,z	
	x=0	x=1
a	a,0	c,0
b	b,0	b,–
c	b,0	a,1

(a) Example FSM.

PS	NS,z	
	x=0	x=1
a	a,0	b,0
b	a or b,0	a,1

(b) Minimized FSM.

Figure 2.9: (a) Example FSM and (b) Minimized FSM with Two Possible Choices for One Next State Entry.

together *viz.* conjunction. The application of Theorem 2.7 *viz.* the threshold operator after each pairwise conjunction of the constraints could help to limit the size of the intermediate BDDs. For example, when constructing the BDD for a binate covering problem from a two-level POS form, a BDD can be easily built for each clause. In fact, The size of the BDD corresponding to a POS clause is linear to the number of literals in the clause. Thus, for binate covering problems in the POS form, the BDD construction for each POS clause is trivial. The problem again arises when performing the conjunction over all the constraints.

2.8 Example and Results

2.8.1 An Illustrative Example

Consider the simple FSM shown in Figure 2.9(a) (taken from [54]). A state-minimized version of this machine can be obtained from the two compatible sets of states $a = \{A, B\}$ and $b = \{B, C\}$. The corresponding flow table is shown in Figure 2.9(b). As

can be seen, there is an alternative for one next-state entry. In order not to preclude any optimization possibility, the flow table in Figure 2.9(b) should be regarded as a symbolic relation. The flow table can be rewritten in the familiar cube-table representation:

0	a	a	0
1	a	b	0
0	b	a,b	0
1	b	a	1

The set of states in this example can be encoded with just one bit ($L = 1$). It should be observed however that this actually makes the problem at hand a special case in terms of relations, since the next-state entry "a,b" can be now replaced by "–" (don't-care). Nonetheless, the example is continued without exploiting this knowledge so that the general procedure can be illustrated. Application of the prime generation procedure yields the following seven primes:

g_1:	0	a	a	0
g_2:	1	a	b	0
g_3:	0	b	a	0
g_4:	0	b	b	0
g_5:	1	b	a	1
g_6:	0	a∪b	a	0
g_7:	–	b	a	0

The corresponding output constraints are given by:

0 a: $(\overline{b}_1 + g_1 + g_6)$
1 a: $(\overline{b}_2 + g_2)$
0 b: $(\overline{b}_1 + g_3 + g_6 + g_7)g_4 + (\overline{b}_2 + g_4)\overline{g}_3\overline{g}_6\overline{g}_7$
1 b: g_5

The disjointness constraints are simply expressed in this case by $(b_1 \oplus b_2)$. There are no non-trivial face embedding constraints in the case of two states only. It can be easily verified now that the constraints for '$a \cup b$', when simplified using the complement of the disjointness constraint as don't-care, result in the tautology. This corresponds to our earlier observation that a "–" (don't-care) could be replaced for the pair of next-states. Carrying on this simplification, the following constraints are left in POS form:

$$(\overline{b}_1 + g_1 + g_6)(\overline{b}_2 + g_2)g_5(b_1 + b_2)(\overline{b}_1 + \overline{b}_2)$$

There are two solutions with two terms and 6 literals. One is:

$$\begin{array}{llll} g_1 = 0 & g_2 = 0 \\ g_3 = 0 & g_4 = 0 \\ g_5 = 1 & g_6 = 1 \\ g_7 = 0 & b_1 = 1 \\ b_2 = 0. \end{array}$$

The other is:

$$\begin{array}{llll} g_1 = 0 & g_2 = 1 \\ g_3 = 0 & g_4 = 0 \\ g_5 = 1 & g_6 = 0 \\ g_7 = 0 & b_1 = 0 \\ b_2 = 1. \end{array}$$

Note that both of these solutions can be derived by solving the binate covering problem.

2.8.2 Experimental Results

An experimental program has been developed. There are two main components to the program: a prime generation procedure, and a binate covering solver. The generation of generalized candidate primes is performed using an extended version of the prime generation procedure developed by Somenzi *et al.* for exact Boolean relation minimization [14]. The *0-hot* conversion scheme proposed by Devadas *et al.* [35] was used as a preprocessing step. A binate covering solver using binary decision diagrams has been developed using Berkeley's BDD package [72]. The shortest path algorithm for finding the minimum cost solution is simply a bottom-up procedure on the BDD data structure. Using this solver, the covering and encoding constraints described in Section 2.6 can be generated directly using BDD operations (*e.g.*, Boolean and, or, not). An advantage of this approach is that the constraints can be built directly from the constraints expressions without first flattening them to a two-level POS form.

In Table 2.1, experimental results collected for some output encoding examples are given. In the table, **in** is the number of inputs, **out** is the number of output symbols, **size** is the size of the constraint BDD, **cost** is the number of primes selected in the solution, and **time** is measured on a DECstation 3100 and expressed in seconds.

The examples that can be completed by experimental exact solver have only been very small. The bottleneck is in the binate covering solver. There are several reasons for

name	in	out	primes	size	cost	time
e1	4	4	13	135	2	0.40
e2	4	5	29	14963	4	21.92
e3	4	4	15	179	2	0.51
e4	4	4	20	679	4	0.88
e5	4	4	19	244	4	0.98
e6	2	4	14	493	2	0.87

Table 2.1: Experimental Results.

in: number of inputs
out: number of output symbols
primes: number of generalized c-primes generated
size: size of the BDD representing the binate covering problem
cost: number of generalized c-primes in the final cover
time: CPU time in seconds

this. One is that the number of generalized candidate primes even for a modest size problem can be extremely large. For example, in an example tested with only 20 symbolic values, the number of generalized candidate primes generated was already over 10,000. This is a problem for the BDD formulation because many variables must be introduced into the constraint formula. Precisely, a variable g_i is required to represent each generalized prime, and a variable b_{i_k} is required to represent each encoding bit of each symbolic value. So for an example with only 20 symbolic values, but 10,000 primes, the number of variables required is $10,000 + (20 * L)$, where L is the code length. Suppose a code length of 5 was chosen. Then a single BDD formula with 10,100 variables must be constructed! In other applications of BDD's, functions typically have no more then a few hundred of variables.

The number of generalized primes may be reduced substantially if essential primes can be detected. Essential primes are those that must be included in any minimal solution. Hence, they can be selected and removed from the covering problem. Also, there may be primes that are completely covered by the essential primes. These primes can be regarded as redundant and be removed from the covering problem. These techniques are widely used in classical two-level Boolean logic minimization [77].

Another problem is the variable ordering. In the experimental program, an arbitrary variable ordering was used. It is well-known that the BDD size is strongly influenced

by the choice of variable ordering. Therefore, it may be possible to significantly reduce the size of the BDD's that represent the constraint expressions if effective variable ordering heuristics can be developed. Then, it may also be possible to extend the applicability of the BDD formulation for solving more difficult binate covering problems.

2.9 Conclusions

In this chapter, the minimization problem of symbolic relations was introduced. It has been shown that the problem of minimizing symbolic relations naturally arises in many contexts in synthesis. A unified framework for finding exact solutions to the two-level minimization of symbolic relations has been developed and described. Based on a generalization of primes and a minimum cost satisfiability formulation of the constrained covering problem, various versions of the minimization problem can be systematically treated. The problem of FSM synthesis using a symbolic relations formulation has also been described. Extensions to the symbolic relations problem were presented to implicitly considered the merging of equivalent states in the synthesis process. Based on these extensions, the traditionally separate problems of state minimization and state assignment were formulated as a single unified optimization problem.

Chapter 3

Multi-Level Symbolic Encoding

3.1 Introduction

Symbolic encoding for two-level implementations (*e.g.*, PLA) is a well-investigated area. A number of viable approaches have been developed [70, 68, 92]. In Chapter 2, the traditional encoding problem was generalized to the case of symbolic relations and it was shown that the solution space is greatly enlarged if multiple output choices and implicit merging are permitted. Incorporating these extra degrees of freedom into the framework of symbolic relation minimization makes it possible to formulate problems such as exact state minimization and state encoding as a single unified optimization problem.

Although a very strong theoretical framework can be built for the two-level case, two-level logic implementations are often inefficient or inappropriate for practical applications. There are many functions for which a two-level realization is known to be explosive in complexity. Even if a two-level implementation is feasible, it is often much more efficient to implement the same behavior in multi-level form. Thus, effective symbolic encoding strategies targeting specifically towards multi-level logic optimization must be developed. Optimal symbolic encoding targeting multi-level implementations is in general more difficult than the two-level case. The main reason for this is that combinational logic optimization of multi-level circuits is itself a developing field.

In the past few years, a number of multi-level encoding strategies have been developed. These multi-level encoding techniques fall into two basic categories: symbolic-minimization-based, and estimation-based. In [56], it was shown how multi-level logic transformations can be extended to symbolically minimize specifications with symbolic inputs.

However, analogous extensions for symbolic outputs have not been developed.

An alternative approach is to determine the encodings such that the encoded circuit description can be better optimized by a multi-level logic optimization system (*e.g.*, the system MIS-II developed at Berkeley [13]). In contrast to the symbolic minimization paradigm, these encodings are determined "prior" to multi-level logic optimization using *static* estimation models. The multi-level encoding program JEDI (written as part of this research) [59], MUSTANG [33], and MUSE [39] fall into this category. These encoding programs have been shown to produce high-quality results for multi-level logic implementations. In all of these programs, the effects of encoding choices on the multi-level logic optimization process are captured by computing a set of pairwise weights (called adjacency relations) between each pair of symbolic values in the encoding problem. The problem of multi-level encoding is then reduced to the solution of the combinational optimization problem defined by the embedding of symbolic values onto a L-dimensional Boolean space. The cost is measured by the total weighted distance, as defined by the Hamming distance metric (*cf.* Section 3.2), between all code assignments. The goal is to find the embedding of symbolic values with the minimum cost. This combinatorial optimization problem is called the *minimum cost graph embedding problem* (or simply the graph embedding problem). Armstrong [3] was the first to formulate the encoding problem for state assignment in this manner. In Armstrong's approach, adjacency relations in terms of Hamming distance between codes of states are defined. These adjacency relations must then be satisfied during the embedding step. However, rather than minimizing the total weighted distance, state pairs are forced to be adjacent to each other whenever possible.

In this chapter, the algorithms and encoding models developed for the multi-level symbolic encoding program JEDI [59] are described. The chapter is organized as follows. Basic definitions are given in the next section. In Section 3.3, the encoding problems considered are defined and a global framework for solving them is proposed. In Section 3.4, new estimation models in JEDI for input, output, and input-output encoding are described. In Section 3.5, a spectrum of algorithms are described for solving the minimum cost graph embedding problem. In particular, a fast heuristic method based on clustering, a simulated annealing based procedure, and an exact formulation based on binate covering (*cf.* Chapter 2) have been developed. These new algorithms represent a tradeoff between CPU time and quality of solutions. Experimental results comparing JEDI with other public domain encoding programs are presented in Section 3.6, and concluding remarks are given in Section

3.7.

3.2 Definitions

A Boolean space B^n can be thought of as a n-dimensional hypercube where each minterm corresponds uniquely to a coordinate in this space. Therefore, a notion of distance can be defined.

Definition 3.1 *Let $\alpha \in B^n$ and $\beta \in B^n$ be two vertices in the space B^n. The* **Hamming distance** *between the two vertices α and β is defined as:*

$$\Delta(\alpha,\beta) = \sum_{i=1}^{n} \mid \alpha_i - \beta_i \mid . \tag{3.1}$$

This corresponds to the number of bit positions where the value of the two vertices differ. The **proximity**, $\mathcal{P}(\alpha,\beta)$, between two vertices is defined as the number of identical bit positions, which can be computed simply as $n - \Delta(\alpha,\beta)$. The Hamming distance and the proximity measure are "dual" metrics that can be used to gauge the relative distance between two vertices in a Boolean space.

The distance between two cubes can also be defined. Specifically, when evaluating $\Delta(\alpha,\beta)$, each component α_i and β_i may be either a 0, 1, or a 2 (a don't-care). The modified Hamming distance metric is defined as follows.

$$\Delta(\alpha,\beta) = \sum_{i=1}^{L} \begin{cases} 0 & \text{if } \alpha_i = \beta_i, \\ \frac{1}{2} & \text{if } \alpha_i = 2 \text{ or } \beta_i = 2, \\ 1 & \text{otherwise} \end{cases} \tag{3.2}$$

3.3 Problem Formulation

In this section, the encoding problems considered are stated and the global formulation is defined. The encoding problems considered can be roughly stated as follows.

1. **Input Encoding:** Given a function $f : B^r \times \Sigma \rightarrow B^n$, where $\Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_{N-1}\}$ and $B = \{0,1\}$, find an encoding $E : \Sigma \rightarrow B^L$ such that L is some code length greater or equal to $\lceil \log_2 N \rceil$, and $\forall \sigma_i, \sigma_j \in \Sigma, E(\sigma_i) \neq E(\sigma_j)$.

2. **Output Encoding:** Given a function $f : B^r \rightarrow \Sigma \times B^n$, where Σ and B are as defined above, find an encoding function of length L for Σ such that each symbolic value is given a unique code.

3. **Input-Output Encoding:** Given a function $f : B^r \times \Sigma \rightarrow \Sigma \times B^n$, where Σ and B are as defined above, find an encoding function of length L for Σ such that each symbolic value is given a unique code. Here, the symbolic input variable and the symbolic output variable both represent the same set of symbolic values. Therefore, they are constrained to be given the same encodings (*e.g.* in state assignment).

In the above encoding problems, only one set of symbolic values is considered for encoding at a time. Multiple symbolic variables can be handled by either by taking their Cartesian product, *i.e.* $\Sigma = \Sigma_1 \times \Sigma_2 \ldots \times \Sigma_m$, to be one symbol set or by considering one symbolic variable at a time and temporarily *one-hot* encoding the remaining symbolic variables for the purpose of determining the encodings.

The basic global strategy used in JEDI follows the same paradigm as MUSTANG [33]. In particular, the encoding procedure proceeds in two steps: a *weight calculation* step and a *minimum cost graph embedding* step. In the weight calculation step, a $N \times N$ integer weight matrix M is derived for the set of N symbolic values $\Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_{N-1}\}$. Each entry $M_{ij} \in \mathbf{Z}$ represents the gains that can be achieved by embedding the symbols σ_i and σ_j as close as possible in the Boolean space. In JEDI, these weights are calculated statically by examining the input-output behavior the symbolic specification directly.

Then, the symbols are encoded using the weight matrix M to provide the cost of an assignment of symbols onto the vertices of the Boolean space. In particular, an optimal solution to the following combinatorial optimization problem is sought.

Problem 3.1 (Minimum Cost Graph Embedding) Given a set of symbolic values $\Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_{N-1}\}$, a weight matrix M, and a code length L, determine an optimal **encoding function** $E : \Sigma \rightarrow B^L$ such that $\forall \sigma_i, \sigma_j \in \Sigma, E(\sigma_i) \neq E(\sigma_j)$ and the objective function

$$cost(E) = \sum_{i=1}^{L-1} \sum_{j=i+1}^{L} M_{ij} \times \Delta(E(\sigma_i), E(\sigma_j)) \tag{3.3}$$

is **minimized**. Here, Δ is the Hamming distance operator defined in Section 3.2. □

Optimizing the objection function (Equation 3.3) will implicitly force pairs of symbols with large weights to be as close together in the Boolean space as possible. This global formulation is then used to solve all of the above encoding problems (*i.e.*, input, output, and input-output encoding). However, depending on the encoding problem, the weights are computed differently.

In [33], the principles behind this basic formulation of the encoding problems were described. Essentially, the formulation attempts to capture the process of common cube extraction in multi-level logic synthesis. Specifically, the goal is to either maximize the sizes of potential common cubes (in input encoding) or the number of occurrences of the largest common cubes (in output encoding). Two key factors influence the effectiveness of this encoding procedure. One is the accuracy of the estimation models, and the other is the robustness of the embedding algorithms. The latter is especially important in evaluating the accuracy of the models. As is presented in the next section, the formulas used for computing the weights between symbolic values are different than those described in [33]. Specifically, the models used in JEDI are not based on a state transition graph model, as in MUSTANG. Effective embedding algorithms are presented in Section 3.5.

3.4 Weight Estimation Models

In this section, estimation models that define a set of pairwise weights between symbolic values are presented. In the program MUSTANG [33], estimation models were presented for the state assignment problem. They are based on analyzing the interaction between states in a state transition graph model of the FSM. In JEDI, the aim is to solve a *generic* symbolic encoding problem rather state assignment specifically, as was the case for MUSTANG. Hence, it does not rely on a state transition graph model for encoding. Instead, estimation models in JEDI are based on analyzing the input-output behavior of the given symbolic function directly. Specifically, a two-level tabular representation of the symbolic function is used. The weight between a pair of symbolic values (σ_i, σ_j) is computed by summing over the "proximity" between all pairs of cubes in the tabular representation of the symbolic function that contain σ_i in one cube and σ_j in the other. This is in contrast to MUSE, which uses models based on analyzing an algebraically factorized *one-hot* coded implementation [39].

There are two main models in JEDI, one for input encoding and one for output encoding. In the case of input-output encoding, a weighted combination of these two models is used.

Input Encoding:

For input encoding, the problem is to encode a symbolic function of the form $f : B^r \times \Sigma \rightarrow B^n$. For the purpose of weight calculation, this function is represented in tabular form as a set of 3-tuples

$$T(I, X, \mathcal{O}).$$

Each entry $(I_j, X_j, \mathcal{O}_i) \in T$ represents a mapping of some input combination $I_j \times X_j$ to an output value $\mathcal{O}_j$. In the input encoding formulation, the goal is to encode the symbolic input values in a manner that will result in many common cubes in the output functions that they assert. In particular, two cubes (with symbolic inputs) that assert similar output patterns can be made to have more common literals by encoding the corresponding symbolic values in the two cubes closer together in the Boolean space. To achieve this, a weight is computed between a pair of symbolic values (σ_i, σ_j) by comparing the output patterns of the subset of cubes that contain σ_i and the subset of cubes that contain σ_j. The subset of cubes that contain σ_i is defined by

$$C_i = \{\mathcal{O}_j \mid \exists I_j : (I_j, \sigma_i, \mathcal{O}_j) \in T\}. \tag{3.4}$$

The subset of cubes containing σ_j is similarly defined. Then a weight between them is computed by summing over the proximity between their output patterns as follows:

$$M_{ij} = \sum_{k=1}^{|C_i|} \sum_{l=1}^{|C_j|} L - \Delta(C_{k,i}, C_{l,j}). \tag{3.5}$$

Here, the term $L - \Delta(C_{k,i}, C_{l,j})$ is the proximity term.

Output Encoding:

The weights for output encoding is computed in a symmetric way. The problem is to encode a symbolic function of the form $f : B^r \rightarrow \Sigma \times B^n$. Similar to the input encoding case, the function f is represented again in tabular form as a set of 3-tuples $T(I, X, \mathcal{O})$. However, each entry $(I_j, X_j, \mathcal{O}_i) \in T$ now represents a mapping of some input I_j to an output combination of $X_j \times \mathcal{O}_j$. In the output encoding formulation, the goal is to find output encodings that will lead to more frequent occurrences of large common cubes. This is achieved by comparing now the *input* patterns of the subset of cubes that contain σ_i and

the subset of cubes that contain σ_j as outputs. The subset of cubes that contain σ_i is now defined by

$$D_i = \{I_j \mid \exists \mathcal{O}_j : (I_j, \sigma_i, \mathcal{O}_j) \in T\}. \tag{3.6}$$

D_j is similarly defined for σ_j. If the input patterns in D_i are very similarly to those D_j, then many common subexpressions can be formed between them. These potential common subexpressions can be made more useful by maximizing their occurrences in the encoded representation. This is achieved by assigning the two corresponding symbolic values with closer codes. Specifically, the weight between a pair of symbolic output values is computed by summing over the proximity between the input patterns that assert them as follows:

$$M_{ij} = \sum_{k=1}^{|D_i|} \sum_{l=1}^{|D_j|} L - \Delta(D_{k,i}, D_{l,j}). \tag{3.7}$$

Input-Output Encoding:

In the case of input-output encoding, the weights are computed by forming a weighted sum of the input weights and the output weights. To compute the input weights, the output symbolic variable is temporarily one-hot encoded. Equation 3.5 is then applied. To compute the output weights, the *input* symbolic variable is temporarily one-hot encoded, and Equation 3.7 is then applied. Let M^I be the input weight matrix and M^O be the output weight matrix. Then each entry of the input-output weight matrix M is simply

$$M_{ij} = \phi \cdot M^I_{ij} + \varphi \cdot M^O_{ij}. \tag{3.8}$$

To give more emphasis on either the input or output effects, the coefficients ϕ and φ can be scaled accordingly. Experimentally, using the value '1' for both coefficients have been found to be effective.

3.5 A Spectrum of Graph Embedding Algorithms

The estimation models presented in the previous section are used to generate a set of weights to guide in the symbolic encoding process. The problem now is to assign the actual binary codes to the symbolic values so that a binary representation of the circuit specification can be derived. The problem was formally stated in Problem 3.1. Given a set of symbolic values $\Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_{N-1}\}$, a weight matrix M, and a code length L,

the goal is to determine an encoding that minimizes Equation 3.3. This objective function accounts for minimal distance codes between symbolic values with large cost relationships.

Clearly, the shortest distance that one can embed two symbolic values in a Boolean space is to assign them adjacent to each other (*i.e.*, Hamming distance of 1). However, since the number of adjacent vertices in a finite Boolean space is extremely limited, this is not possible for all symbol pairs. In fact, only a very small percentage of symbol pairs can be made adjacent to each other. This percentage decreases exponentially with the number of symbols in the encoding set. Therefore, the weights are used to determine the tradeoffs. Unfortunately, the minimum cost graph embedding problem is in the NP [43]. In this section, a number of algorithms for solving the graph embedding problem are presented. First, a fast heuristic procedure based on clustering is presented. Then, a simulated annealing based procedure is described. Finally, an exact formulation based on binate covering (minimum-cost satisfiability) is described.

3.5.1 A Clustering Algorithm

A simple, but effective, algorithm is incremental clustering. The procedure is presented in Figure 3.1. The main idea is to select one symbolic value at a time to encode. For the duration of the procedure, three sets are kept: one representing the set of symbols that have been encoded, simply represented by the set Σ itself, one representing the set of encoded symbols, denoted by A, and one containing the set of already used binary codes, denoted by C. Both A and C are initially empty. The core of the algorithm involves a selection step and an assignment step. At each iteration, the selection step is used to select an unassigned symbolic value σ_i such that it has the strongest cost relationship to the set of already encoded symbolic values. This can be determined by $\sigma_i = \arg\max_{\sigma_i \in \Sigma} \left(\sum_{\sigma_j \in A} M_{ij} \right)$. Once an un-encoded symbolic value σ_i is chosen, an unused binary code must be chosen to represent it. The assignment step tries to assign an unused binary code such that it is *closest* to the binary codes already given to the encoded symbolic values. This can be determined by the expression $\mathbf{x}_i = \arg\min_{\mathbf{x}_i \notin C} \left(\sum_{\sigma_j \in A} M_{ij} \times \Delta(\mathbf{x}_i, \mathbf{x}_j) \right)$. The process is iterated until all symbolic values have been encoded.

This basic algorithm can be improved in efficiency by keeping a priority queue of un-encoded symbolic values and available codes. For the selection step, a gain value γ_i can

```
/*
-- inputs:  Σ (set of symbols), M (weight matrix)
-- output:  E (the encoding)
 */

cluster_encode(Σ,M)
{
    /* initialize */
    A = {}; C = {};
    repeat{
        σ_i = arg max_{σ_i∈Σ} (Σ_{σ_j∈A} M_ij);
        x_i = arg min_{x_i∉C} (Σ_{σ_j∈A} M_ij × Δ(x_i, x_j));
        /* assign code */
        E(σ_i) = x_i;
        /* update sets */
        A = A ∪ σ_i;
        Σ = Σ − σ_i;
        C = C ∪ x_i;
    }until Σ is empty;
    return E;
}
```

Figure 3.1: The Algorithm `cluster_encode` for the Minimum Cost Graph Embedding Problem.

be kept for each un-encoded σ_i. The quantity γ_i is defined as follows:

$$\gamma_i = \sum_J M_{ij}, \text{ where } J = \{j | \sigma_j \in A\}. \tag{3.9}$$

The un-encoded symbolic value with the largest γ_i is selected in the inner loop. A direct pointer to this un-encoded symbolic value is kept so that it can be returned in constant time. After selection, updating each γ_i for the next selection step can be accomplished by simply incrementing it with the cost M_{ij}, where j corresponds to the symbolic value, σ_j, that was just selected. While updating each unselected γ_i, the direct pointer to the highest gain unassigned symbolic value is also updated. The same trick can be applied to keep a

priority queue of unused codes.

3.5.2 Simulated Annealing Formulation

Alternatively, the graph embedding problem can be solved using the technique of *simulated annealing*. Simulated annealing belongs to a class of probabilistic hill climbing algorithms and was first proposed by Kirkpatrick [53] for solving general discrete optimization problems. Simulated annealing has since attracted a great deal of attention for its effectiveness. It has been successfully applied to such CAD problems as placement and floorplanning [82] and module folding [37]. A significant theoretical foundation has also been developed for simulated annealing [76, 85]. Although probabilistic hill climbing algorithms in general may not be well-suited for all combinatorial optimization problems, they have been seen to be extremely effective for the graph embedding problem (*cf.* Section 3.6).

The simulated annealing algorithm for encoding, `anneal_encode`, is shown in Figure 3.2. The core of the simulated annealing algorithm is characterized by a *configuration space*, a *cost function*, and a probabilistic procedure for controlling the search through the solution space. The controlling procedure used here is a simple geometric cooling schedule. The initial configuration is generated by a random assignment. Successive *moves* are generated either by swapping codes or changing the assignment of a symbol with an unused code, as shown in the procedure `new_configuration` in Figure 3.3. The cost of an assignment is evaluated using Equation 3.3. The acceptance condition at the inner loop is determined by the change in cost c and the current temperature T. Specifically, if the cost was improved, then the new configuration is accepted. Otherwise, it is accepted with the probability $e^{-\frac{c}{T}}$. More details on simulated annealing can be found in [53].

3.5.3 Exact Binate Covering Formulation

The above algorithms are based on heuristics. While effective, they cannot guarantee the global optimum solution. A naive approach to the exact graph embedding problem is to enumerate all possible bindings of codes. However, this is not practical and does not contribute to the deeper understanding of the problem.

In this section, an exact solution to the minimum cost graph embedding problem is described. The problem is formulated as a minimum binate covering problem. The graph

```
/*
-- inputs:  Σ (set of symbols), M (weight matrix)
-- output:  E (the encoding)
 */

anneal_encode(Σ,M)
{
    /* initialize */
    E = random_encoding(Σ);
    foreach temperature point T{
        repeat{
            E' = new_configuration(E);
            c = cost(E') - cost(E);
            if(accept(c,T)){
                E = E';
            }
        }until inner loop condition is satisfied;
    }
    return E;
}
```

Figure 3.2: The Algorithm anneal_encode for the Minimum Cost Graph Embedding Problem.

embedding problem can be solved by casting the optimization problem into a set of Boolean constraints. The constraints are then combined together into a single Boolean formula by taking their conjunction. The set of assignments that satisfies the formula corresponds to the set of possible encodings. The problem is then to find a satisfying assignment with the minimum cost. This is similar to the binate covering formulation described in Chapter 2 for the symbolic relation minimization problem. Specifically, two sets of constraints are required. The first set of constraints expresses the condition that no two symbols may have the same code [1]. These constraints are called disjointness constraints.

[1]This restriction can be relaxed if two symbols are *equivalent*, as described in Chapter 2.

```
/*
-- inputs:  E (old encoding)
-- output:  E (modified encoding)
 */

new_configuration(E)
{
    repeat{
        randomly select codes x1 and x2;
    }until code_is_used(x1) or code_is_used(x2);
    if(code_is_used(x1)){
        /* change the code of σi */
        σi = code_assigned_to_symbol(x1);
        E(σi) = x2;
    }else if(code_is_used(x2)){
        /* change the code of σi */
        σi = code_assigned_to_symbol(x2);
        E(σi) = x1;
    }else{
        /* exchange codes */
        σi = code_assigned_to_symbol(x1);
        σj = code_assigned_to_symbol(x2);
        tmp = x1;
        E(σi) = x2;
        E(σj) = tmp;
    }
    /* return modified encoding */
    return E;
}
```

Figure 3.3: The Algorithm new_configuration for Annealing Based Embedding.

Disjointness Constraints: The disjointness constraints are required here are the same as those needed for the symbolic relation minimization problem. Precisely, given N symbolic

values and L bits for encoding, the set of disjointness constraints can be expressed as follows.

$$\prod_{i=1}^{N-1} \prod_{j=i+1}^{N} \sum_{k=1}^{L} (b_{ik} \oplus b_{jk}) = 1. \tag{3.10}$$

The disjointness constraints guarantee that each symbolic value is assigned a distinct code.

Weighting Constraints: A second set of constraints is introduced to capture the cost of an assignment. This is accomplished by introducing a set of Boolean variables $\{w_{ij,1}, w_{ij,2}, \ldots, w_{ij,L}\}$ for every pair of symbols σ_i and σ_j. If a variable $w_{ij,k}$ is assigned a '1', it means that the k-th bit of the codes assigned to σ_i and σ_j are different. The accounting of the weights accumulated for L bits and N symbolic values can be expressed as follows:

$$\prod_{i=1}^{N-1} \prod_{j=i+1}^{N} \prod_{k=1}^{L} [(b_{ik} \oplus b_{jk}) \overline{\oplus} w_{ij,k}] = 1. \tag{3.11}$$

This expression can be expanded into POS (product-of-sum) form as follows.

$$\prod_{i=1}^{N-1} \prod_{j=i+1}^{N} \prod_{k=1}^{L} (b_{ik} + b_{jk} + \overline{w_{ij,k}}) = 1, \tag{3.12}$$

$$\prod_{i=1}^{N-1} \prod_{j=i+1}^{N} \prod_{k=1}^{L} (\overline{b_{ik}} + \overline{b_{jk}} + \overline{w_{ij,k}}) = 1, \tag{3.13}$$

$$\prod_{i=1}^{N-1} \prod_{j=i+1}^{N} \prod_{k=1}^{L} (\overline{b_{ik}} + b_{jk} + w_{ij,k}) = 1, \tag{3.14}$$

$$\prod_{i=1}^{N-1} \prod_{j=i+1}^{N} \prod_{k=1}^{L} (b_{ik} + \overline{b_{jk}} + w_{ij,k}) = 1. \tag{3.15}$$

$$\tag{3.16}$$

Expressing the constraints in POS form makes it possible to use the branch-and-bound strategies for binate covering described in Section 2.7.2 of Chapter 2.

Here, the expression $[(b_{ik} \oplus b_{jk}) \overline{\oplus} w_{ij,k}]$ forces $w_{ij,k}$ to assume a value '1' if the k-th bit of codes for σ_i and σ_j are different and '0' if they are the same. Thus, in essence the $w_{ij,k}$ variables also captures the disjointness of two bit assignments. Therefore, the disjointness constraints can be simplified to:

$$\prod_{i=1}^{N-1} \prod_{j=i+1}^{N} \sum_{k=1}^{L} w_{ij,k} = 1. \tag{3.17}$$

The total cost of an assignment can be determined by summing over all the $w_{ij,k}$ variables. The minimum cost solution can be found using the binate covering techniques described in Chapter 2. To obtain the global optimum solution, the Boolean variables of the overall constraint formula must be weighted. The weights are as follows:

1. weight($w_{ij,k}$) = M_{ij};
2. weight(b_{ij}) = 0;

Theorem 3.1 *The minimum cost satisfying assignment to the above binate covering formulation is also the solution to the minimum cost graph embedding problem.*

Proof: The disjointness constraints guarantee that no two symbols are assigned the same code. The variable $w_{ij,k}$ can only be true if in fact the k-th bit of symbols σ_i and σ_j differs. Hence, the solution would only incur that cost if the encodings are different. By the definition of Hamming distance, this properly accounts for the cost. ■

This formulation of the problem also permits an effective heuristic solution by solving the binate covering problem heuristically.

3.6 Experimental Results

In this section, empirical results from a number of experiments are presented to demonstrate the the effectiveness of the proposed encoding models and optimization algorithms. The examples used for these experiments are the 1989 International Workshop on Logic Synthesis (IWLS'89) finite state machine benchmarks [64]. There are 40 FSM examples in total. The largest of these examples has 121 states. The IWLS'89 benchmark distribution is publicly available from the Micro-electronics Center of North Carolina (MCNC). Some general statistics on the 40 benchmark examples are given in Table 3.1. Specifically, the number of primary inputs (#I) and primary outputs (#O), the number of internal states (#S), and the minimum code length used to encode the states (#L) for each benchmark FSM are indicated. All experiment results presented in this chapter were obtained on a DECstation 3100.

To test the performance of JEDI, three sets of experiments were conducted. These experiments are as follows:

machine	#I	#O	#S	#L
bbara	4	2	10	4
bbsse	7	7	16	4
bbtas	2	2	6	3
beecount	3	4	7	3
cse	7	7	16	4
dk14	3	5	7	3
dk15	3	5	4	2
dk16	2	3	27	5
dk17	2	3	8	3
dk27	1	2	7	3
dk512	1	3	15	4
donfile	2	1	24	5
ex1	9	19	20	5
ex2	2	2	19	5
ex3	2	2	10	4
ex4	6	9	14	4
ex5	2	2	9	4
ex6	5	8	8	3
ex7	2	2	10	4
keyb	7	2	19	5
kirkman	12	6	16	4
lion	2	1	4	2
lion9	2	1	9	4
mark1	5	16	15	4
mc	3	5	4	2
modulo12	1	1	12	4
opus	5	6	10	4
planet	7	19	48	6
s1	8	6	20	5
s1a	8	6	20	5
s8	4	1	5	3
sand	11	9	32	5
scf	27	56	121	7
shiftreg	1	1	8	3
sse	7	7	16	4
styr	9	10	30	5
tav	4	4	4	2
tbk	6	3	32	5
train11	2	1	11	4
train4	2	1	4	2

Table 3.1: Statistics for IWLS'89 FSM Benchmarks.

1. The goal of the first experiment is to determine the effectiveness of JEDI on the problem of multi-level symbolic encoding. For this purpose, three other public domain programs that can perform both input and output encoding are used for comparisons. Specifically, the programs NOVA [92], MUSTANG [33], and MUSE [39] are used. In addition, results using *one-hot* and *random* encodings were also collected.

2. In the second experiment, different *post*-encoding optimization procedures are compared. Since the programs JEDI, MUSE, and MUSTANG are not based on symbolic minimization, logic optimization must be carried out after the replacement of the symbolic values by their binary codes.

3. In the third experiment, the effectiveness of the different graph embedding algorithms presented in Section 3.5 are compared.

3.6.1 Comparing Encoding Programs

The experimental results for the first experiment are given in Table 3.2 for the 40 IWLS'89 benchmark FSM's. The problem being solved is the classical state assignment problem. The experiment was conducted as follows:

- Except for one-hot encoding, minimum code length was always used.
- For each program, all available options were used, and the best results are reported in Table 3.2.

    1. For JEDI, four options were used: "-i" for input encoding, "-o" and "-y" for output encoding, and "-c" for input-output encoding.
    2. For MUSE, three options were used: "-p" for input encoding, "-n" for output encoding, and "-pn" for input-output encoding.
    3. For MUSTANG, four options were used: "-p" and "-pt" for input encoding, and "-n" and "-nt" for output encoding.
    4. For NOVA, 8 options were used: "-ig", "-ih", "-ioh", "-iov", "-ie", "-ia", "-j", and "-y". These options reflect the different optimization algorithms in NOVA for performing input, output, and input-output encoding.
    5. For *one-hot* encoding, a simple translation was performed.

6. For *random* encoding, the results obtained are the best of N runs with different random seeds, where N is the number of states of the benchmark.

- After the encoding has been determined, the optimized logic was obtained as follows:

  1. ESPRESSO [78] was run on the encoded machine using the unused binary codes as don't-cares.
  2. The optimized two-level description was then optimized using MIS-II version 2.2 [13] from Berkeley. The standard script was used once in all cases.

- The results reported are expressed in terms of factored form literals in the optimized encoded representation.

The benchmark examples are listed in alphabetical order. The row labeled **total** indicates the total literal count of the best results over all options for each program. The row labeled **ratio** gives the percentage of total literal count improvement made by JEDI. From the experiment, several observations can be made. First, the program JEDI consistently produces better or comparable results on the examples tested. The overall total is the best among the programs compared. Second, the program is extremely fast. On most examples, the program completes in only a few minutes or less. This makes it possible to apply the program to very large examples. For the program MUSE, better results were reported in [39] when longer than minimum code lengths were used. In the experiments reported here, the same code lengths (minimum) were used for all the programs to obtain a direct comparison.

3.6.2 Comparing Post Encoding Optimization Procedures

Next, the effects of different post encoding optimization procedures are examined. The basic post encoding optimization procedure is to apply ESPRESSO on the initially encoded representation with the unused codes as don't-cares. This is followed by using MIS-II for multi-level logic optimization using the standard script. Here, several subtle variations to this basic procedure are examined. The results are shown in Table 3.3. The programs JEDI and MUSE are used for this comparison.

The default optimization script is shown in Figure 3.5. This option is referred to as the ES script. The command **read_pla -s** reads the PLA representation produced by

machine	JEDI	MUSE	NOVA	MUSTANG	*one-hot*	*random*
bbara	59	60	54	58	73	73
bbsse	99	96	103	99	112	116
bbtas	22	22	22	25	28	26
beecount	33	42	32	37	74	40
cse	173	175	167	189	185	203
dk14	93	96	72	101	116	81
dk15	60	63	61	60	94	58
dk16	210	208	209	231	180	287
dk17	49	51	52	47	66	55
dk27	22	20	25	21	25	23
dk512	50	57	53	59	55	68
donfile	49	84	78	153	152	202
ex1	224	209	220	254	179	276
ex2	124	118	102	121	119	173
ex3	61	59	70	71	68	68
ex4	57	65	57	65	62	70
ex5	54	57	56	63	71	56
ex6	79	84	77	82	72	79
ex7	62	62	69	67	69	66
keyb	184	173	185	191	159	253
kirkman	138	148	129	178	178	175
lion	13	13	13	13	27	13
lion9	15	21	25	17	51	33
mark1	74	74	75	80	86	89
mc	25	22	21	23	31	21
modulo12	30	29	22	36	48	34
opus	71	68	67	72	67	83
planet	481	469	564	513	332	607
s1	81	119	242	218	205	336
s1a	130	87	244	182	168	324
s8	28	22	27	21	57	25
sand	437	430	433	462	450	490
scf	829	810	834	830	527	980
shiftreg	2	2	0	2	28	21
sse	99	96	103	99	112	116
styr	371	370	422	409	342	482
tav	26	26	25	25	24	22
tbk	219	264	241	473	664	535
train11	28	31	34	38	56	35
train4	17	13	14	18	20	13
total	4878	4915	5299	5703	5432	6702
ratio	1.00	1.01	1.09	1.17	1.11	1.37

Table 3.2: Experimental Results for IWLS'89 FSM Benchmarks.

```
E script {
 % espresso < fsm.encode > fsm.espresso;
 % misII
 misII> read_pla fsm.espresso
 misII> source script
 misII> print_stats -f
 misII> write_blif fsm.blif
 misII> quit
}
```

Figure 3.4: The E Post Encoding Optimization Script.

```
ES script {
 % espresso < fsm.encode > fsm.espresso;
 % misII
 misII> read_pla -s fsm.espresso
 misII> source script
 misII> print_stats -f
 misII> write_blif fsm.blif
 misII> quit
}
```

Figure 3.5: The ES Post Encoding Optimization Script.

ESPRESSO and collapses the logic for each output function into a one-level logic. If instead the command **read_pla** is used, then the two-level form is kept. This option is referred to as the E script and is shown in Figure 3.4. The results obtained using this script is less effective by a factor of 12% in the case of JEDI and a factor of about 15% in the case of MUSE. This can be partly explained by the fact that the two-level representation can be recovered by re-extracting the common cubes from the one-level form. Hence, a one-level form has, in some sense, a greater degree of freedom. If ESPRESSO is forced to minimize each function separately, then the overall results are even better. This option is referred to as the ESO script and is shown in Figure 3.6. For this, the **-Dso** option of ESPRESSO is used. This leads to a further reduction of about 3%. The column labeled BEST is the best

```
ESO script {
 % espresso -Dso < fsm.encode > fsm.espresso;
 % misII
 misII> read_pla -s fsm.espresso
 misII> source script
 misII> print_stats -f
 misII> write_blif fsm.blif
 misII> quit
}
```

Figure 3.6: The ESO Post Encoding Optimization Script.

of the three post encoding optimization scripts.

3.6.3 Comparing Graph Embedding Algorithms

Finally, the effectiveness of various graph embedding algorithms are compared. The ability to solve the graph embedding problem effectively is essential to the encoding approach presented in this chapter. First, a comparison of the heuristic algorithms are compared. In particular, the clustering algorithm described in Section 3.5.1 is compared with the simulated annealing algorithm described in Section 3.5.2. The results are tabulated in Table 3.4 and Table 3.5. In Table 3.4, the cost obtained for each benchmark as measured by Equation 3.3 is given. The weight matrix M for each example was obtained using JEDI with the **-o** (default) option. To normalize the comparisons, the results of randoming encodings were also obtained. Relative comparisons are made with the random encoded results. On the examples tested, the clustering algorithm was able to produced results with 8 to 75% reduction in cost. The simulated annealing algorithm is strictly better than clustering in all cases. The last row labeled **total** gives the overall results. The column labeled **ratio** gives the geometric mean. As can be seen, both clustering and simulated annealing were able to obtain substantial reduction over a naive random encoding strategy. Overall, simulated annealing is the most effective. However, the superior results obtained using simulated annealing comes with a price of higher CPU times. Table 3.5 gives the CPU times for both algorithms. On average, clustering is over two orders of magnitude faster. Despite this, with the exception of the largest example **scf**, the CPU times for either

machine	JEDI				MUSE			
	E	ES	ESO	BEST	E	ES	ESO	BEST
bbara	61	59	53	53	59	60	55	55
bbsse	119	99	102	99	114	96	94	94
bbtas	22	22	22	22	22	22	23	22
beecount	32	33	28	28	49	42	32	32
cse	213	173	169	169	218	175	171	171
dk14	99	93	85	85	111	96	79	79
dk15	67	60	57	57	66	63	61	61
dk16	237	210	204	204	232	208	211	208
dk17	52	49	49	49	48	51	49	48
dk27	22	22	20	20	20	20	19	19
dk512	51	50	46	46	68	57	55	55
donfile	47	49	48	47	84	84	80	80
ex1	272	224	231	224	245	209	186	186
ex2	123	124	115	115	125	118	115	115
ex3	60	61	61	60	59	59	54	54
ex4	64	57	50	50	67	65	64	64
ex5	54	54	51	51	57	57	55	55
ex6	114	79	76	76	115	84	88	84
ex7	63	62	60	60	62	62	60	60
keyb	170	184	176	170	226	173	189	173
kirkman	156	138	136	136	157	148	143	143
lion	13	13	12	12	13	13	12	12
lion9	15	15	14	14	21	21	22	21
mark1	87	74	64	64	90	74	66	66
mc	28	25	23	23	25	22	21	21
modulo12	31	30	31	30	29	29	29	29
opus	83	71	67	67	75	68	71	68
planet	523	481	430	430	508	469	434	434
s1	78	81	78	78	166	119	110	110
s1a	161	130	117	117	101	87	86	86
s8	28	28	22	22	22	22	22	22
sand	560	437	441	437	526	430	432	430
scf	812	829	817	812	825	810	817	810
shiftreg	2	2	2	2	2	2	2	2
sse	119	99	102	99	114	96	94	94
styr	471	371	401	371	529	370	398	370
tav	26	26	26	26	26	26	26	26
tbk	277	219	215	215	337	264	261	261
train11	28	28	26	26	31	31	27	27
train4	17	17	15	15	13	13	13	13
total	5457	4878	4742	4681	5657	4915	4826	4760
ratio	1.12	1.00	0.97	0.96	1.16	1.01	0.99	0.97

Table 3.3: Comparisons of JEDI and MUSE with Different Post Optimization Procedures.

clustering or simulated annealing are insignificant. Since simulated annealing produces strictly better results with moderate CPU times, it is used as the default graph embedding algorithm.

The heuristic algorithms are also compared against the exact solution. In particularly, the binate covering formulation of the problem described in Section 3.5.3 has been implemented using a branch-and-bound procedure. The results are given in Table 3.6. Here, a subset of 9 small examples are used for comparisons. Although the binate covering formulation can guarantee the exact result under the chosen cost function, it is at present infeasible for large examples. However, it can be used to measure the robustness of the heuristic algorithms. For the examples that binate covering can complete in reasonable time, the simulated annealing algorithm was able to obtain the same exact result in all cases, but in a substantially shorter time period. The clustering algorithm is also quite robust despite its simplicity.

3.7 Conclusions

Simple, but effective, encoding algorithms for multi-level logic synthesis have been provided. These algorithms have been implemented in the computer program JEDI. The presented algorithms are based on an estimation approach to encoding and are extremely fast. Hence, they can be can applied to relatively large problem instances. The estimation models purposed rely on a symbolic description of functionality rather than a state diagram, as in previous state assignment programs. Therefore, the algorithms can be directly used in other encoding applications, such as encoding hardware description languages for general synthesis. The main weakness of the proposed approach is that it has only very limited analytical correlation with the transformations developed for multi-level logic optimization. Nonetheless, the approach has been shown to perform consistently well on a wide range of benchmarks. This is demonstrated by comparing the program implementation JEDI against the best available programs that can perform both input and output encoding.

Central to the encoding formulation presented is the problem of minimum cost graph embedding. For this problem, a spectrum of optimization algorithms have been devised. The simplest of these is an incremental clustering strategy. This method is extremely fast and has been shown to produce excellent results on the chosen cost function despite its simplicity. A simulated annealing algorithm has also been devised. It also can produce very

machine	random	cluster		annealing	
	cost	*cost*	*ratio*	*cost*	*ratio*
bbara	2762	1828	0.66	1761	0.64
bbsse	1464	992	0.68	947	0.65
bbtas	194	159	0.82	145	0.75
beecount	323	273	0.85	262	0.81
cse	5021	4195	0.84	4150	0.83
dk14	1589	1187	0.75	1171	0.74
dk15	231	179	0.77	179	0.77
dk16	9689	7281	0.75	7285	0.75
dk17	317	293	0.92	273	0.86
dk27	35	26	0.74	21	0.60
dk512	277	155	0.56	149	0.54
donfile	7796	5836	0.75	5420	0.70
ex1	26384	14930	0.57	14490	0.55
ex2	4227	2020	0.48	1958	0.46
ex3	824	346	0.42	346	0.42
ex4	223	202	0.91	192	0.86
ex5	652	268	0.41	250	0.38
ex6	269	281	1.04	201	0.75
ex7	621	335	0.54	299	0.48
keyb	13921	5420	0.39	5273	0.38
kirkman	332451	287839	0.87	280010	0.84
lion	14	8	0.57	8	0.57
lion9	429	166	0.39	166	0.39
mark1	349	272	0.78	241	0.69
mc	8	4	0.50	4	0.50
modulo12	206	144	0.70	150	0.73
opus	365	255	0.70	221	0.61
planet	4985	2494	0.50	2423	0.49
planet1	4985	2494	0.50	2423	0.49
s1	7844	4926	0.63	4420	0.56
s1a	7844	4926	0.63	4420	0.56
s8	322	200	0.62	200	0.62
sand	24394	16997	0.70	15340	0.63
scf	36407	32621	0.90	32380	0.89
shiftreg	64	16	0.25	16	0.25
sse	1464	992	0.68	947	0.65
styr	31534	26075	0.83	25530	0.81
tav	384	384	1.00	384	1.00
tbk	5075263	1511030	0.30	1496000	0.29
train11	424	226	0.53	200	0.47
train4	36	24	0.67	24	0.67
total	5606591	1938299	0.66	1910280	0.62

Table 3.4: Comparing the Effectiveness of Different Graph Embedding Algorithms.

machine	cluster	annealing	
	time	*time*	*ratio*
bbara	0.01	0.35	35.00
bbsse	0.01	1.66	166.00
bbtas	0.01	0.11	11.00
beecount	0.01	0.15	15.00
cse	0.01	1.67	167.00
dk14	0.01	0.16	16.00
dk15	0.01	0.05	5.00
dk16	0.01	6.18	618.00
dk17	0.01	0.23	23.00
dk27	0.01	0.21	21.00
dk512	0.01	1.09	109.00
donfile	0.01	4.43	443.00
ex1	0.01	2.60	260.00
ex2	0.01	2.27	227.00
ex3	0.01	0.41	41.00
ex4	0.01	1.03	103.00
ex5	0.01	0.26	26.00
ex6	0.01	0.20	20.00
ex7	0.01	0.38	38.00
keyb	0.02	2.52	126.00
kirkman	0.01	1.30	130.00
lion	0.01	0.06	6.00
lion9	0.01	0.25	25.00
mark1	0.01	1.25	125.00
mc	0.01	0.05	5.00
modulo12	0.01	0.66	66.00
opus	0.01	0.40	40.00
planet	0.12	40.69	339.08
planet1	0.11	40.78	370.73
s1	0.02	2.55	127.50
s1a	0.01	2.61	261.00
s8	0.01	0.07	7.00
sand	0.02	10.23	511.50
scf	1.34	551.93	411.89
shiftreg	0.01	0.19	19.00
sse	0.01	1.39	139.00
styr	0.02	8.03	401.50
tav	0.01	0.06	6.00
tbk	0.03	9.63	321.00
train11	0.01	0.44	44.00
train4	0.01	0.05	5.00
total	2.01	698.58	142.22

Table 3.5: CPU Expenditures of `cluster_encode` *vs.* `anneal_encode`.

machine	cluster		annealing		exact	
	cost	*time*	*cost*	*time*	*cost*	*time*
bbtas	159	0.01	145	0.11	145	164.81
beecount	273	0.01	262	0.15	262	1992.64
dk14	1187	0.01	1171	0.16	1171	1554.58
dk15	179	0.01	179	0.05	179	0.17
dk27	26	0.01	21	0.21	21	1557.96
lion	8	0.01	8	0.06	8	0.15
mc	4	0.01	4	0.05	4	0.17
s8	200	0.01	200	0.07	200	17.30
tav	384	0.01	384	0.06	384	0.17
train4	24	0.01	24	0.05	24	0.18

Table 3.6: Comparisons with Exact Graph Embedding.

good results. Although simulated annealing is typically thought of as a time-consuming algorithm that should only be used as a last resort, it has been shown in this chapter that a carefully implemented annealing strategy can in fact produce excellent results with only modest CPU times, at least for the graph embedding problem. This algorithm is consistently better than the clustering algorithm, and hence it is used as the default in JEDI. An exact formulation of the graph embedding problem based on binate covering has also been developed. While it is only feasible for small examples, it can be used to measure the robustness of the heuristic algorithms, which according to the comparisons on a small set of examples, they do surprisingly well. It is expected that larger examples can be handled by the exact formulation when better binate covering strategies become available.

Chapter 4

Optimization of Sequential Circuits

4.1 Introduction

In the previous two chapters, algorithms were presented for encoding a symbolic specification of hardware, possibly starting from a HDL description [71, 50], to a circuit-level description consisting of Boolean logic gates and synchronous latches. In this chapter, the problem of optimizing the encoded sequential circuit before mapping it into a final physical implementation is addressed. Over the past few years, synthesis methods for optimizing combinational logic have achieved a significant level of maturity. This can be seen by the number of successful multi-level logic optimization systems that have been developed from both universities [13, 10] and industry [29, 45]. Although these systems provide powerful optimization techniques for reducing area and improving circuit performance, to date they have only tackled the combinational portions of the sequential circuit and have not consider interactions between combinational logic blocks separated by latches.

In contrast with methods for combinational circuits, techniques for optimizing sequential circuits are considerably less developed. Previous methods have been proposed for optimizing a state diagram model before encoding it to a logic-level description for optimization. State minimization [90, 44] is one such method. The major drawback of methods that optimize at the state diagram level (such as trying to minimize the number of states or edges) is the remoteness of the state diagram model from the actual cost of the final

implementation. This makes it difficult to evaluate such key figures of merit as area and performance during the optimization process. A more critical shortcoming is that many sequential circuits of practical interest cannot be represented with state diagrams, either because there are too many states or the combinational blocks cannot be flattened into a two-level form. Alternatively, recent research efforts [69, 66, 28, 57] have focussed on methods that operate directly at the structural level. These methods are capable of detecting interactions between combinational blocks separated by latches, and hence can potentially optimize across latch boundaries, either by temporarily re-positioning some latches [66] or by establishing don't-care conditions across latches [28]. These techniques, while potentially powerful, make only limited use of *global state-space* information during their optimization step.

Global state-space information, such as knowledge about invalid states and equivalent states, can be used to optimize sequential circuits at the structural level by considering them as *sequential don't-care conditions.* Don't-cares have long been recognized as very important in synthesizing combinational [7] and sequential logic [34]. In effect, information on invalid states and equivalent states can be used to provide "bounds" on how much the combinational portions of the sequential circuit can be "perturbed" or optimized without modifying the overall sequential behavior. For example, if a state is invalid, then the output or next-state behavior of the sequential circuit starting from this state is not important. This represents an important sequential don't-care condition that can very often be found. Equivalent states are also quite important. Given an input condition, the next state response can be any one of the equivalent states. Invalid and equivalent state information can be computed by examining the underlying state-space of the circuit. A key advantage of using global state-space information is that well-developed algorithms for combinational logic synthesis are directly applicable. Global state-space information can potentially be used in conjunction with structural level transformations such as retiming. In addition to reducing area, and possibly improving performance, an intimate relationship exists between the optimal exploitation of don't-cares and the combinational [7] and sequential [34] testability of the resulting circuit [7, 34]. In [34], it was recognized that the optimal use of don't-cares in minimizing a finite state machine will lead to a fully non-scan testable sequential circuit.

Traditional techniques for state-space exploration have relied on the use of state diagram models. Unfortunately, these techniques are not applicable to a substantial class of sequential circuits that contain many memory elements and data path circuitry. This

limitation is due either to a state explosion or a logic explosion (when collapsed to two-level logic form). Recently, more efficient procedures based on enumerating *covers* or *cubes* have been proposed that do not require explicit construction of state diagrams [42]. Although these procedures can be applied to larger circuits, their applicability is still limited to circuits whose state-space can be represented compactly in two-level form. This is because covers and cubes are essentially two-level representations.

To extend the applicability of global state-space information in synthesis, new efficient procedures for computing global state-space information must be developed so that much larger state-spaces can be handled. In this chapter, new algorithms using binary decision diagrams (BDD's [20]) are proposed for extracting global state-space information. These algorithms are based on the concept of breadth-first implicit state enumeration recently developed by Coudert *et al.* [25], and are capable of computing sequential don't-care conditions like invalid and equivalent states for very large sequential circuits. The extracted sequential don't-care conditions are then used to optimize circuit-level representation directly by using multi-level logic optimization techniques.

The remainder of this chapter is organized as follows. In the next section, the basic method of state-space analysis and its application to sequential optimization is presented. Then in Section 4.3, techniques for implicit state enumeration using binary decision diagrams are reviewed. These techniques can be directly used for computing the set of invalid states by performing reachability analysis. The computation of equivalent states is, however, more difficult. Algorithms for computing equivalent states are described in Section 4.4. Experimental results are given in Section 4.5, and finally, concluding remarks are given in Section 4.6.

4.2 State Space Analysis for Sequential Optimization

An example of a sequential circuit is shown in Figure 4.1. The behavior of such a circuit can be modeled with a state diagram. The state diagram corresponding to the above circuit is shown in Figure 4.2. Each of possible binary vectors that can be stored in the latches corresponds to a "state" of the sequential circuit. Since there are two latches, there can be $4 = 2^2$ possible states, namely 00, 01, 10, and 11. In general, there can be as many as 2^n states for a circuit with n latches. Suppose a reset state-code of 00 is given. Then from this state, only a subset of the state-space may be reachable. In the example

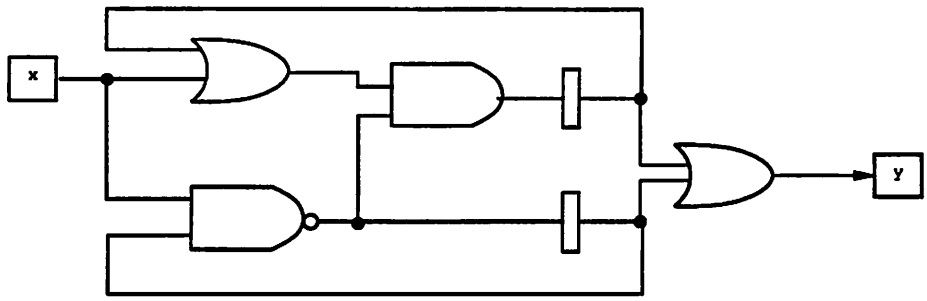


Figure 4.1: The Original Circuit.

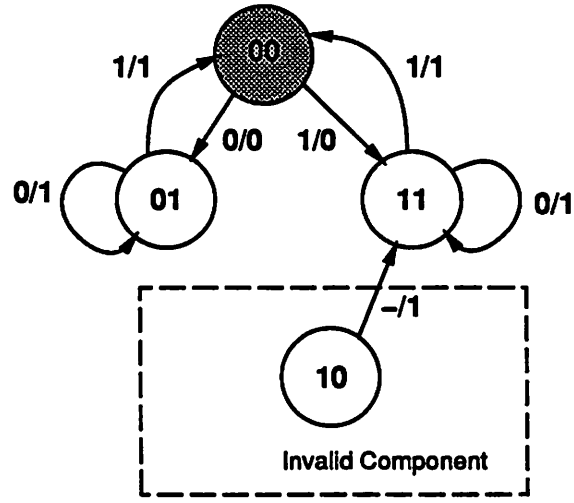


Figure 4.2: The State Diagram Corresponding to the Above Example.

of Figure 4.2, the state 10 cannot be reached with any input sequence from 00. Therefore, it is called an *invalid* state. The other states that can be reached from the reset state by some input sequence are called *valid* states. The corresponding input sequence is termed a *justification sequence.* The portion of the state diagram consisting only of invalid states and the associated edges emanating from them is called the *invalid component* of the state diagram. The remainder part of the state diagram is called the *valid component.*

A pair of states (s_1, s_2) are considered *equivalent* if no input sequence exists that

can be applied to the two states such that a different output is produced. An input sequence that can differentiate a state pair (s_1, s_2) is called a *differentiating sequence* for the state pair. Referring to the state diagram shown in and Figure 4.2, the state pair (01,11) are equivalent, but all other state pairs, *e.g.*, (00,10), are differentiable. The invalid states and equivalent states information can be used to advantage in logic optimization. In a sense, the invalid states and equivalent states information provide "bounds" on how much the combinational portions of the circuit can be perturbed or optimized without modifying the overall sequential behavior. Invalid states can be used as traditional *external don't-care conditions* in combinational logic synthesis. Combinational logic optimization techniques such as those described in [7, 81] can be applied to optimize a circuit under external don't-cares. Equivalent states can also be used in logic optimization by considering output equivalence classes or *Boolean relations*. The main idea is to permit the output of the next state functions to be any one of the equivalent next-states. This is the same idea as described in Chapter 2 for giving more degrees of freedom to the encoding process. Recently, techniques have been proposed to optimize a multi-level logic circuit under a Boolean relation specification [80, 40]. In this work, the authors proposed to add an *observability function* on top of the original network to capture the degrees of freedom permitted by a Boolean relation. Current logic synthesis algorithms can then freely modify the original network as long as the overall behavior at the output of the observability function remains unchanged. The main point that should be emphasized here is that existing combinational logic optimization methods are capable of exploiting both of these kinds of sequential don't-care conditions.

Considering again the example shown in Figure 4.1. It can easily be verified that the circuit cannot be simplified any further using only combinational techniques. However, if the state 10 is used as an invalid state don't-care and the equivalent state pair (00,10) is used as an output equivalence class in logic optimization, then the circuit can be further simplified to a much more cost-effective version shown in Figure 4.3. Note that this minimized circuit can be implemented with *one* simple gate rather than *four*. The corresponding modified state diagram is shown in Figure 4.4(a). Note that one of the two next state functions degenerates to a constant "0". Hence, the corresponding latch can be removed. Doing so will leave a circuit with only one simple gate and one latch. The resulting state diagram is shown in Figure 4.4(b).

It should be clear from the above example that the use of global state space information in logic optimization is extremely important from the point of view of producing

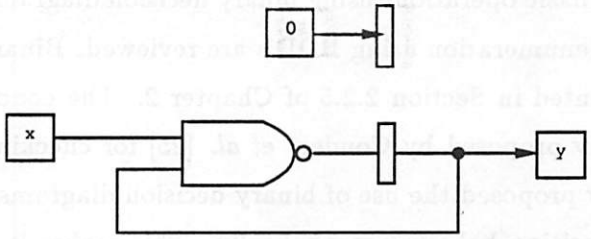


Figure 4.3: The Simplified Circuit under Invalid and Equivalent States.

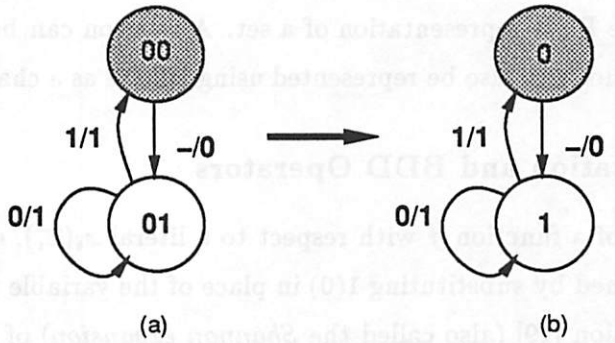


Figure 4.4: The Corresponding State Diagrams.

a more cost effective solution. In addition to reducing area, it is well known that the optimal use of invalid states and equivalent states don't-care conditions in logic minimization will lead to a fully-testable sequential circuit without memory access [34]. The main bottleneck is the computation of invalid states and equivalent states. In practice, these computations are very difficult: either because the circuit contains data-path-like components such as adders and comparators, which causes logic explosion, or the circuit contains many latches, which causes state-space explosion. Circuits with these properties would render traditional techniques based on state diagrams ineffective. Therefore, more sophisticated algorithms based on binary decision diagrams are proposed in the sequel.

4.3 Efficient State Enumeration

In this section, basic operations using binary decision diagrams (BDD's) and techniques for implicit state enumeration using BDD's are reviewed. Binary decision diagrams have already been presented in Section 2.2.5 of Chapter 2. The concept of implicit state enumeration was recently proposed by Coudert *et al.* [25] for checking the equivalence of sequential circuits. They proposed the use of binary decision diagrams for representing the state-space and the transition behavior symbolically. This makes it possible to perform breadth-first state enumeration by traversing a set of states at a time. For the remainder of the chapter, all Boolean and set operations are assumed to be implemented using BDD's. Also, unless otherwise noted, no distinction will be made between a set, the *characteristic function* of a set, or the BDD representation of a set. A relation can be viewed as a set of n-tuples. Hence, a relation can also be represented using BDD's as a characteristic function.

4.3.1 Set Computation and BDD Operators

The *cofactor* of a function f with respect to a literal $x_i(\overline{x_i})$, denoted by $f_{x_i}(f_{\overline{x_i}})$, is a new function obtained by substituting 1(0) in place of the variable x_i in the function f [12]. The Boole expansion [19] (also called the *Shannon expansion*) of a Boolean function f with respect to a variable x_i is

$$f = x_i f_{x_i} + \overline{x_i} f_{\overline{x_i}}.$$

Binary decision diagram [20] is a compact representation of a nested Boole decomposition.

In addition to the basic propositional logic operations, the existential operator ($\exists$) and the universal operator ($\forall$) are required for implicit enumeration. These Boolean quantifiers can also be computed directly on a BDD representation as follows. The existential quantification (also called *smoothing*) of a set of Boolean variables $X = \{x_1, \ldots, x_n\}$ with respect to the Boolean formula f can be evaluated as

$$\begin{aligned} \exists X(f) &= \exists x_1(\exists x_2(\ldots \exists x_{n-1}(\exists x_n(f))\ldots)), \\ \exists x_i(f) &= f_{x_i} + f_{\overline{x_i}}, \end{aligned}$$

where $f_{x_i}(f_{\overline{x_i}})$ denotes the *cofactor* of formula f with respect to a literal $x_i(\overline{x_i})$. $f_{x_i}(f_{\overline{x_i}})$ is a new formula obtained by substituting 1(0) in place of the variable x_i in the formula f.

Likewise, the universal quantifier (also called the *consensus*) can be evaluated as

$$\begin{aligned} \forall X(f) &= \forall x_1(\forall x_2(\ldots \forall x_{n-1}(\forall x_n(f))\ldots)), \\ \forall x_i(f) &= f_{x_i} \cdot f_{\overline{x_i}}. \end{aligned}$$

Any quantified propositional formula can be rewritten into a propositional formula using the above rewriting rules.

Let $f : B^r \rightarrow B^n$ be a Boolean function and $\xi \subseteq B^r$ a subset of the input space. The *image* of ξ by f is the set $f(\xi) = \{y \in B^n \mid y = f(x),\ x \in \xi\}$. If $\xi = B^r$, the image of ξ by f is also called the *range* of f. Let $\psi \subseteq B^n$ be a subset of the output space. The *inverse-image* (or the reverse-image) of ψ by f is the set $f^{-1}(\xi) = \{x \in B^r \mid y = f(x),\ y \in \psi\}$.

The cofactor operation for single literals or cubes was previously defined [12]. However, cofactoring with respect to an arbitrary function in an important operation for image computation and simplification. Let $f : B^r \rightarrow B^n$ be a Boolean function and $c : B^r \rightarrow B$ be a *care set*. The *generalized cofactor* of f with respect to c, written f_c or **gen_cofactor**(f, c), is the new function

$$f_c(x) = \begin{cases} f(x) & \text{if } c(x) = 1 \\ y & \text{such that } y \in f(c) \end{cases}$$

This operator was initially proposed by Coudert *et al.* in [25] and was called the *constraint* operator. In Coudert's implementation, f_c is dependent on the variable ordering.

4.3.2 Representation of States and State Relations

Let M be a sequential circuit with r primary inputs, m latches, and n primary outputs. States, state relations, and output relations can be represented as follows.

Definition 4.1 *Let Q be a set of states in B^m. The* **characteristic function** *of Q is a function such that $Q(\mathbf{x}) = 1$ if and only if $\mathbf{x}$ is a state in Q.*

For example, the set of possible reset states may be represented by the characteristic function $I(\mathbf{x})$.

Definition 4.2 *The* **next-state transition relation** *for sequential circuit M is $T \subseteq B^r \times B^m \times B^m$ such that $T(i, \mathbf{x}, \mathbf{y}) = 1$ if and only if the state $\mathbf{y}$ can be reached in exactly one state transition from state $\mathbf{x}$ when input i is applied.*

The transition relation can be interpreted as a set of state transitions. It can also be represented as a binary relation $T(\mathbf{x},\mathbf{y}) = \exists i\ T(i,\mathbf{x},\mathbf{y})$, where $T(\mathbf{x},\mathbf{y}) = 1$ if and only if state $\mathbf{y}$ can be reached in one transition from state $\mathbf{x}$ under some input. In this case, T represents a set of un-labeled transitions.

In general, the transition relation can be non-deterministic or incompletely specified. However, for the algorithms developed in this chapter, a *deterministic* transition relation is assumed.

Definition 4.3 *The* **output relation** *for sequential circuit M is $\mathcal{O} \subseteq B^r \times B^m \times B^n$ such that $\mathcal{O}(i,\mathbf{x},z) = 1$ if and only if the output pattern z can be produced from state $\mathbf{x}$ when input i is applied.*

In the algorithms used for extracting global state-space information (*e.g.*, equivalent states calculation), a product machine is required. A product machine for a sequential circuit M is simply a new network constructed by duplicating itself, connecting the same primary inputs to both copies of the circuit, and connecting the primary outputs of the two copies to a set of bit-wise equivalence gates:

$$(z_{1_1} \overline{\oplus} z_{2_1})(z_{1_2} \overline{\oplus} z_{2_2}) \cdots (z_{1_n} \overline{\oplus} z_{2_n}).$$

This is shown in Figure 4.5. The state-space is defined by the Cartesian product $B^m \times B^m$. A set of states of the product machine is as represented before. The *product* transition relation can also be implicitly represented as follows.

Definition 4.4 *The* **product transition relation** *of M is $P \subseteq B^r \times B^{2m} \times B^{2m}$ such that $P(i,\mathbf{X},\mathbf{Y}) = 1$ if and only if the product state $\mathbf{Y} = (\mathbf{y},\mathbf{y}')$ can be reached in exactly one state transition from the product state $\mathbf{X} = (\mathbf{x},\mathbf{x}')$ when input i is applied. Each product state corresponds to a state pair of the original circuit M.*

For a product transition relation, the primary inputs can be abstracted away to produce $P(\mathbf{X},\mathbf{Y}) = \exists i\ P(i,\mathbf{X},\mathbf{Y})$.

Definition 4.5 *The output of the product machine is a single-output* **output equivalence function** $\mathcal{Z}(i,\mathbf{X})$ *that evaluates to a* **1** *if the output patterns produced under input i by the circuit M are the same starting from either the state $\mathbf{x}$ or the state $\mathbf{x}'$. Otherwise, it evaluates to a* **0**.

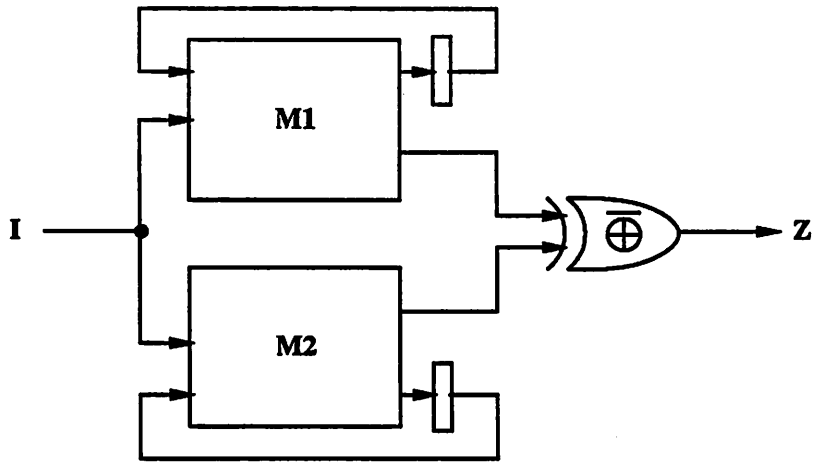


Figure 4.5: Product Machine.

In equivalent states calculation, the problem is to find the complete set of state pairs $(\mathbf{x}, \mathbf{x}')$ such that the output equivalence function $\mathcal{Z}$ always remains a 1 under all possible input sequences.

4.3.3 Implicit Enumeration and Fixed Point Computation

Given a BDD representation of the transition relation, forward (image) and backward (inverse-image) computations can be done simply by performing a simple conjunction followed by variable quantifications. Given a set of states $c(\mathbf{x})$, the set of next-states that can be reached in one transition is simply

$$c'(\mathbf{y}) = \exists i, \mathbf{x} \ (T(i, \mathbf{x}, \mathbf{y}) \cdot c(\mathbf{x})).$$

Similarly, given a set of states $c(\mathbf{y})$, the set of previous states that can be reached in one step back is

$$c'(\mathbf{x}) = \exists i, \mathbf{y} \ (T(i, \mathbf{x}, \mathbf{y}) \cdot c(\mathbf{y})).$$

Recently, the idea of representing the global transition relation T as a conjunction of simpler *partitioned* transition relations was proposed [89]. This technique makes it possible

to perform state enumeration when the global transition relation cannot be built. The method relies on the fact that many variables can be abstracted away before taking the full conjunction. Coudert *et al.* [25] also proposed efficient algorithms based on transition functions for image computation. Although their method may be more efficient on some examples, it cannot be used for backward traversal, which is required for equivalent states computation. Since the technique of transition relation, in particular *partitioned relations* [89], is as efficient in almost all cases, it is used to develop the algorithms in the remainder of this chapter.

The last concept required to describe the algorithms is the notion of fixed point computation. The notation and terminology used here are same as those found in [88, 21]. Let S be a finite set of elements. Let Z be a function that maps the power set $\mathcal{P}(S)$ to itself, *i.e.*, $Z : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$, and assume Z is *monotone*, *i.e.*, $A \subseteq B$ implies that $Z(A) \subseteq Z(B)$. The *least fixed point* of Z, $\mathrm{lfp}(Z)$, and the *greatest fixed point* of Z, $\mathrm{gfp}(Z)$, are defined as follows:

$$\begin{aligned} \mathrm{lfp}(Z) &= \sum_{A=Z(A)} A \\ \mathrm{gfp}(Z) &= \prod_{A=Z(A)} A \end{aligned}$$

Since Z is monotone, it can be checked [21] that there is always some integer $j \geq 0$ such that $Z_j(S) = Z_{j+1}(S) = \mathrm{lfp}(Z)$, and $Z_j(\emptyset) = Z_{j+1}(\emptyset) = \mathrm{gfp}(Z)$, where S denotes the complete set and $\emptyset$ denotes the empty set. Here Z_j refers to the set obtained after the j-th iteration of the fixed point computation. The check for termination is simply a tautology check, for which the use of BDD's is particularly well-suited.

A simple example is the computation of reachable states $R(\mathbf{x})$ given a set of initial states $I(\mathbf{x})$. It can be easily computed as the least fixed point computation

$$\begin{aligned} R_0(\mathbf{x}) &= I(\mathbf{x}) \\ R_{j+1}(\mathbf{x}) &= R_j(\mathbf{x}) + (\exists i)(\exists \mathbf{y})\,(T(i,\mathbf{y},\mathbf{x}) \cdot R_j(\mathbf{y})). \end{aligned} \tag{4.1}$$

The set of *invalid states* $U(\mathbf{x})$ is simply the complement of R.

4.4 Computing the Equivalent States

The problem of computing invalid states was already shown in the previous section. The computation of equivalent states is more complicated. Classical equivalent states

identification procedure requires the construction of merger graphs where each state must be explicitly represented [54]. However, the algorithm is known to be $O(N \log_2 N)$, where $N = 2^r$ is the number of states and r is the number of state-bits. Hence, this algorithm is not feasible for a large class of circuits.

In this section, algorithms are given for computing this information implicitly using binary decision diagrams and fixed point computations. Given a sequential circuit, the goal is to compute the complete set of equivalent state pairs. This set can be represented as a characteristic function using BDD's as follows.

Definition 4.6 *The* **equivalent state relation** *for a sequential circuit M is a characteristic function $E(\mathbf{x}, \mathbf{x}')$ such that $E(\mathbf{x}, \mathbf{x}') = 1$ if and only if state $\mathbf{x}$ is equivalent to state $\mathbf{x}'$ in M.*

To perform the equivalent state relation, a product machine is first constructed, as shown in Figure 4.5. The product transition relation $P(i, \mathbf{X}, \mathbf{Y})$ and the output equivalence function $\mathcal{Z}(i, \mathbf{X})$ are then built using binary decision diagrams. These relations can be kept as partitioned relations whenever possible to reduce the storage requirements.

Recall that the condition for a pair of states $(\mathbf{x}, \mathbf{x}')$ to be equivalent is that under all input sequences starting from either state, the output response is the same. This corresponds to the set of "product" states in the product machine such that under all input sequences, the output equivalence function $\mathcal{Z}$ evaluates to a **1**. Thus, the problem is reduced to finding the set of product states, $E(\mathbf{X})$, that satisfy this requirement. This equivalence relation can be computed using a backward traversal of the state-space [1].

In this section, two BDD-based proof procedures are described for computing this equivalence relation. One is to find a set of product states such that $\mathcal{Z}$ evaluates to **1** under all (single) inputs i, and all the possible next product states are also equivalent. This can be computed as a greatest fixed point computation. The other is to instead determine a set of product states that are differentiable, *i.e.* there exists some input sequence such that $\mathcal{Z}$ eventually evaluates to a **0**. This can be computed as a least fixed point computation.

[1] A forward traversal algorithm for equivalent states computation has not been explored.

4.4.1 Equivalence-Based Analysis

In the first proof method, the following greatest fixed point computation can be used for computing the set of equivalent states:

$$\begin{aligned} E_0(\mathbf{X}) &= (\forall i)\mathcal{Z}(i,\mathbf{X}) \\ E_{j+1}(\mathbf{X}) &= E_j(\mathbf{X}) \cdot (\forall i)(\exists \mathbf{Y})\,[P(i,\mathbf{X},\mathbf{Y}) \cdot E_j(\mathbf{Y})]. \end{aligned} \tag{4.2}$$

Here, a deterministic transition relation is assumed. Given i, $\mathbf{X}$, $\mathbf{Y}$ is unique such that $P(i,\mathbf{X},\mathbf{Y}) = 1$. A similar fixed point computation was independently proposed in [73]. However, this formula, as written, has several notable disadvantages. In the term

$$(\forall i)(\exists \mathbf{Y})\,[P(i,\mathbf{X},\mathbf{Y}) \cdot E_j(\mathbf{Y})],$$

the primary input variables cannot be abstracted away before the complete expression $(\exists \mathbf{Y})\,[P(i,\mathbf{X},\mathbf{Y}) \cdot E_j(\mathbf{Y})]$ is computed since the universal and existential quantifiers do not commute. Although the partitioned transition relation idea introduced in [89] can abstract away some of the existentially quantified variables, such as the $\mathbf{Y}$ variables here, computing the complete expression without abstracting the primary input variables in the process may be too memory intensive (in terms of the number of BDD nodes). Therefore, the computation may, and does fail on larger examples. Indeed, this problem has been the cause for failure on many examples. Also, an important trick proposed by Coudert *et al.* [25] for reducing the cost of computing least fixed points can not be applied here. The main idea of their trick is to perform a backward (or forward) computation on only a subset of states at each iteration. Specifically, only new states generated at each iteration need to considered rather than the total set. They proposed the use of the generalized cofactor operator to heuristically select a set *between* those two extremes such that the BDD representation is small. It turns out that this trick is crucial for some computations but unfortunately it cannot be applied directly here.

4.4.2 Differentiation-Based Analysis

An alternative proof procedure is to compute the complete set of state pairs, $D(\mathbf{x},\mathbf{x}')$ or $D(\mathbf{X})$, that are *differentiable*. The set of equivalent state pairs can simply be obtained by taking the complement. This can be computed with the following least fixed

point computation:

$$\begin{aligned} D_0(\mathbf{X}) &= (\exists i)\overline{\mathcal{Z}(i,\mathbf{X})} \\ D_{j+1}(\mathbf{X}) &= D_j(\mathbf{X}) + (\exists i)(\exists \mathbf{Y})\,[P(i,\mathbf{X},\mathbf{Y}) \cdot D_j(\mathbf{Y})]. \end{aligned} \tag{4.3}$$

This formula, as written, can be evaluated much more efficiently than Equation 4.2. Since this is a least fixed point computation, the trick of Coudert *et al.* [25] applies, which is essential for many large examples. Since existential quantifications can commute, one can order the computation in the term $(\exists i)(\exists \mathbf{Y})\,[P(i,\mathbf{X},\mathbf{Y}) \cdot D_j(\mathbf{Y})]$ such that the variables in i and $\mathbf{Y}$ can be abstracted away as soon as possible using the technique presented in [89]. This is also essential for large examples.

Equations 4.2 and 4.3 are in fact dual of each other. At each iteration, $D_j(\mathbf{X})$ is the complement of $E_j(\mathbf{X})$. Both algorithms require the same number of iterations to reach their fixed points, and the final sets are complementary to each other.

Theorem 4.1 *For all j, $E_j(\mathbf{X}) = \overline{D_j(\mathbf{X})}$.*

To prove Theorem 4.1, the following lemmas are first presented.

Lemma 4.2 $E_0(\mathbf{X}) = \overline{D_0(\mathbf{X})}$.

Proof: Trivial. $(\forall i)\mathcal{Z}(i,\mathbf{X}) = \overline{(\exists i)\overline{\mathcal{Z}(i,\mathbf{X})}}$. ■

Lemma 4.3 *Given i, $\mathbf{X}$, if $\mathbf{Y}$ is unique such that $P(i,\mathbf{X},\mathbf{Y}) = 1$, then*

$$(\exists \mathbf{Y})\,[P(i,\mathbf{X},\mathbf{Y}) \cdot E_j(\mathbf{Y})] = (\forall \mathbf{Y})\left[\overline{P(i,\mathbf{X},\mathbf{Y})} + E_j(\mathbf{Y})\right].$$

To prove Lemma 4.3, a more general lemma is posed.

Lemma 4.4 *Let $F : B^r \rightarrow B^n$ be any function and let $\mathcal{F} : B^r \times B^n \rightarrow B$ be its characteristic function. Let $C \subseteq B^n$ such that $\mathbf{y} \in C$ if and only if $C(\mathbf{y}) = 1$. Then,*

$$(\exists \mathbf{y})\,[\mathcal{F}(\mathbf{x},\mathbf{y}) \cdot C(\mathbf{y})] = (\forall \mathbf{y})\left[\overline{\mathcal{F}(\mathbf{x},\mathbf{y})} + C(\mathbf{y})\right].$$

Proof: To prove the above, the following should first be noted:

1. $\forall \mathbf{x} \in B^r$, $\mathbf{y} = \mathcal{F}(\mathbf{x})$ is unique and well-defined.

2. $(\forall \mathbf{y})\left[\overline{\mathcal{F}(\mathbf{x},\mathbf{y})} + C(\mathbf{y})\right] = (\forall \mathbf{y})\left[\overline{\mathcal{F}(\mathbf{x},\mathbf{y})} + \mathcal{F}(\mathbf{x},\mathbf{y})C(\mathbf{y})\right]$.

Let

$$S(\mathbf{x}) = (\exists \mathbf{y})[\mathcal{F}(\mathbf{x},\mathbf{y}) \cdot C(\mathbf{y})] \tag{4.4}$$

$$\hat{S}(\mathbf{x}) = (\forall \mathbf{y})\left[\overline{\mathcal{F}(\mathbf{x},\mathbf{y})} + \mathcal{F}(\mathbf{x},\mathbf{y})C(\mathbf{y})\right]. \tag{4.5}$$

The prove will now proceed in two steps. First, $\mathbf{x} \in S(\mathbf{x})$ implies $\mathbf{x} \in \hat{S}(\mathbf{x})$ is shown. In Equation 4.4, $\mathbf{x} \in S(\mathbf{x})$ implies $(\mathbf{y} = \mathcal{F}(\mathbf{x})) \in C(\mathbf{y})$. In Equation 4.5, $\mathbf{Y} = \overline{\mathcal{F}(\mathbf{x})}$ for the same $\mathbf{x}$ is the set of elements $\mathbf{Y} = B^n - \mathbf{y}$. Therefore, $\hat{S}(\mathbf{x}) = (\forall \mathbf{y})\left[\overline{\mathcal{F}(\mathbf{x},\mathbf{y})} + \mathcal{F}(\mathbf{x},\mathbf{y})C(\mathbf{y})\right]$ will also contain $\mathbf{x}$.

Next, $\mathbf{x} \in \hat{S}(\mathbf{x})$ implies $\mathbf{x} \in S(\mathbf{x})$ is shown. Since $\mathbf{y} = \mathcal{F}(\mathbf{x})$ is unique and well-defined, $(\forall \mathbf{y})\left[\overline{\mathcal{F}(\mathbf{x},\mathbf{y})}\right]$ is always empty. Therefore, $\mathbf{x} \in \hat{S}(\mathbf{x})$ implies that there exists at least one $\mathbf{y}$ such that $\mathcal{F}(\mathbf{x},\mathbf{y})C(\mathbf{y})$ is true. This implies that $\mathbf{x}$ is also contained in $S(\mathbf{x})$. ■

Proof: [Proof of Lemma 4.3] $P(i,\mathbf{X},\mathbf{Y})$ is a characteristic function corresponding to a deterministic function. Therefore, $\forall i, \mathbf{X}$, $\mathbf{Y} = P(i,\mathbf{X})$ is unique and well-defined. Lemma 4.3 then follows from Lemma 4.4. ■

Proof: [Proof of Theorem 4.1] The prove is by induction on k.

Induction Hypothesis: Assume the statement in the theorem is true for all j up to k.

Induction Basis: The statement in the theorem was proven in Lemma 4.2 for $j = 0$.

Induction Step: It is now proven for $j + 1$.

$$\begin{aligned}
E_{j+1}(\mathbf{X}) &= \overline{D_{j+1}(\mathbf{X})} \\
&= \overline{D_j(\mathbf{X}) + (\exists i)(\exists \mathbf{Y})[P(i,\mathbf{X},\mathbf{Y}) \cdot D_j(\mathbf{Y})]} \\
&= \overline{D_j(\mathbf{X})} \cdot \overline{(\exists i)(\exists \mathbf{Y})[P(i,\mathbf{X},\mathbf{Y}) \cdot D_j(\mathbf{Y})]} \\
&= E_j(\mathbf{X}) \cdot (\forall i)(\forall \mathbf{Y})\overline{[P(i,\mathbf{X},\mathbf{Y}) \cdot D_j(\mathbf{Y})]} \\
&= E_j(\mathbf{X}) \cdot (\forall i)(\forall \mathbf{Y})\left[\overline{P(i,\mathbf{X},\mathbf{Y})} + \overline{D_j(\mathbf{Y})}\right] \\
&= E_j(\mathbf{X}) \cdot (\forall i)(\forall \mathbf{Y})\left[\overline{P(i,\mathbf{X},\mathbf{Y})} + E_j(\mathbf{Y})\right]
\end{aligned}$$

From Lemma 4.3,

$$E_{j+1}(\mathbf{X}) = E_j(\mathbf{X}) \cdot (\forall i)(\exists \mathbf{Y})[P(i,\mathbf{X},\mathbf{Y}) \cdot E_j(\mathbf{Y})]$$

■

Therefore, the computations given in Equations 4.2 and 4.3 produces exactly the dual results. Note that although the formulas given in Equation 4.2 and Equation 4.3 can be rewritten from one form to the other, Equation 4.3 is much more efficient to evaluate using state-of-the-art computation techniques.

4.4.3 Computing Single Cycle State Equivalence

Although current state enumeration techniques based on binary decision diagrams are very powerful, there are still many difficult examples where the necessary fixed point computations for computing equivalent states cannot be performed. While current trends seem to dictate that more efficient techniques will likely be developed in the near future, they may still not be sufficient to tackle all realistic examples in practice. In the reachable state computation, only the state-space of the circuit needs to be examined. However, in the equivalent state computation, it is the state-space of the product machine that must be examined. This state-space is much more complicated in most cases.

Depending on the sequential circuit, relatively large sets of equivalent state pairs can be detected *without* a full state-space traversal. In some cases, surprisingly, a significant percentage of the equivalent state pairs are in fact *single-cycle* equivalent, meaning that the output response and the next-state transitions are exactly the same under all primary input conditions (single vector). Therefore, these equivalent state pairs can all be derived in only one backward computation step. For this computation, a new function on the product machine is defined.

Definition 4.7 *Given a product machine (as shown in Figure 4.5),* **a next-state equivalence function** $\mathcal{L}(i, \mathbf{X})$ *is defined by connecting the corresponding next-state functions of the two copies of the machine to a set of bit-wise equivalence gates:*

$$(t_{1_1} \overline{\oplus} t_{1_2})(t_{2_1} \overline{\oplus} t_{2_2}) \cdots (t_{m_1} \overline{\oplus} t_{m_2}),$$

where t_{i_1} *and* t_{i_2} *are corresponding next-state functions.* $\mathcal{L}(i, \mathbf{X})$, $\mathbf{X} = (\mathbf{x}, \mathbf{x}')$, *evaluates to a* **1** *if the next-states produced under input* i *by the circuit* M *are the same, starting from either the state* $\mathbf{x}$ *or the state* $\mathbf{x}'$. *Otherwise, it evaluates to a* **0**.

This function is analogous to the output equivalence function $\mathcal{Z}(i, \mathbf{X})$. Given this function,

the set of single-cycle equivalent state pairs can be computed as follows:

$$\begin{aligned} S(\mathbf{X}) &= (\forall i)[\mathcal{Z}(i,\mathbf{X}) \cdot \mathcal{L}(i,\mathbf{X})], \\ \overline{S(\mathbf{X})} &= (\exists i)\left[\overline{\mathcal{Z}(i,\mathbf{X})} + \overline{\mathcal{L}(i,\mathbf{X})}\right]. \end{aligned} \tag{4.6}$$

This computation requires only one step. Larger subsets can be generated by traversing backwards on the product relation.

4.4.4 Computing State Equivalence in the Valid Component

The fixed point computations in Equation 4.2 and Equation 4.3 can be used to compute the complete set of equivalent state pairs. However, one is often only concerned about the equivalent states in the valid component of the state-space (*i.e.* the reachable states). Let M be the finite state machine under consideration. Let $R(\mathbf{x})$ be the characteristic function representing the set of reachable states in M and let $E(\mathbf{x},\mathbf{x}')$ be the characteristic function representing the set of equivalent state pairs of M. The set E contains both valid and invalid states. If only the set of equivalent state pairs in the valid component of the state-space is required, then it can be computed simply as follows:

$$E_V(\mathbf{x},\mathbf{x}') = E(\mathbf{x},\mathbf{x}') \cdot R(\mathbf{x}) \cdot R(\mathbf{x}'). \tag{4.7}$$

However, this is unnecessarily expensive. In fact, the set E is usually much more expensive to compute than the set R for most examples. Hence computing E first is a crucial bottleneck.

If the eventual goal is to compute only the set of equivalent state pairs in the valid component, then it is possible to first simply the transition relations representing the state machine using the invalid states as *don't-cares* before computing the equivalent states. Doing so almost always makes the equivalent states computation much easier. Specifically, the product transition relation $P(i,\mathbf{X},\mathbf{Y})$ and the output equivalence function $\mathcal{Z}(i,\mathbf{X})$ can first be reduced with respect to the reachable states R. The *generalized cofactor* can be used for this purpose as a heuristic minimizer to minimize the BDD representation as follows:

$$R(\mathbf{X},\mathbf{Y}) = R(\mathbf{x}) \cdot R(\mathbf{x}') \cdot R(\mathbf{y}) \cdot R(\mathbf{y}'),$$

$$R(\mathbf{X}) = R(\mathbf{x}) \cdot R(\mathbf{x}'),$$

$$P_R(i,\mathbf{X},\mathbf{Y}) = \textbf{gen\_cofactor}(P(i,\mathbf{X},\mathbf{Y}), R(\mathbf{X},\mathbf{Y})), \tag{4.8}$$

$$\mathcal{Z}_R(i,\mathbf{X}) = \textbf{gen\_cofactor}(\mathcal{Z}(i,\mathbf{X}), R(\mathbf{X})). \tag{4.9}$$

In Equation 4.8, $P_R(i, \mathbf{X}, \mathbf{Y})$ is actually computed by applying **gen_cofactor** with R on each of the next-state functions of the product machine before building the product transition relation. This ensures that $\mathbf{Y}$ is still unique for each combination of i and $\mathbf{X}$ in the reduced product transition relation $P_R(i, \mathbf{X}, \mathbf{Y})$.

It can be shown that the set of equivalent states in the valid component using the reduced product transition relation, $P_R(i, \mathbf{X}, \mathbf{Y})$, and the reduced output equivalence function, $\mathcal{Z}_R(i, \mathbf{X})$, is in fact the same as the equivalent states in the valid component using $P(i, \mathbf{X}, \mathbf{Y})$ and $\mathcal{Z}(i, \mathbf{X})$.

Theorem 4.5 *Let $P_R(i, \mathbf{X}, \mathbf{Y})$ and $\mathcal{Z}_R(i, \mathbf{X})$ be the reduced product transition relation and the reduced output equivalence function, respectively. Let $E_R(\mathbf{x}, \mathbf{x}')$ be the set of equivalent states using $P_R(i, \mathbf{X}, \mathbf{Y})$ and $\mathcal{Z}_R(i, \mathbf{X})$, and let*

$$E_{R,V}(\mathbf{x}, \mathbf{x}') = E_R(\mathbf{x}, \mathbf{x}') \cdot R(\mathbf{x}) \cdot R(\mathbf{x}')$$

be the set of equivalent states in the valid component. Then

$$E_V(\mathbf{x}, \mathbf{x}') = E_{R,V}(\mathbf{x}, \mathbf{x}').$$

Proof: Simplifying the product transition relation and output equivalence function using invalid states as don't-cares does not change the behavior of the valid component of M. Therefore, if two reachable states $\mathbf{x}$ and $\mathbf{x}'$ were equivalent in M, they will remain equivalent. ■

4.5 Experimental Results

In this section, empirical results on the techniques described in this chapter. All results presented were collected on a DECstation 5000 and the CPU times are reported in seconds. The algorithms were developed using MIS-II [13] version 2.2 as a package, which provided extensive support for experimentation. For the experiments, the following benchmark examples were used: a set of sequential circuits from the 1989 ISCAS sequential benchmark set, a 32-bit carry-bypass adder called `cbp.32.4`, a 32-bit MINMAX benchmark from IMEC called `minmax32`, a key encryption circuit called `key` from a data encryption standard (DES) chip [91], and a simplified version of `key` called `tkey`. The example `tkey`

Circuit	#L	states	iter.	time
s344	15	2625	7	24.0
s349	15	2625	7	23.5
s382	21	8865	151	56.8
s400	21	8865	151	56.6
s420	16	18	18	2.0
s444	21	8865	151	65.6
s641	19	1544	7	9.9
s713	19	1544	7	9.8
s838	32	18	18	5.1
tkey	228	1.35e+68	17	1325.22
key	228	1.35e+68	17	1325.22
cbp.32.4	32	4.29e+9	2	4.00
minmax32	96	1.32e+28	4	45.50

Table 4.1: Computation of Invalid States.

#L: number of latches
states: number of reachable states
iter.: number of iterations in the FSM forward traversal
time: CPU time in seconds for computing the invalid states

was obtained from key by simply considering only a subset of outputs. The purpose of the experiments was to determine the computational efficiency of the sequential don't-care extraction algorithms and the effectiveness of the don't-cares in optimization.

In Table 4.1 and Table 4.2, results are given for the computing reachable and equivalent states. For reachable state computation, the number of latches (**#L**), the number of reachable states (**set**), the number of iterations in the fixed point computation (**iter.**), and the required CPU time (**time**) are presented. For equivalent state computation, the number of equivalent state pairs (**pairs**), the size of the BDD's representing the set of equivalent states, the number of iterations in the backwards fixed point computation (**iter.**), and the required CPU time (**time**) are in Table 4.2. The results presented in Table 4.2 are for the **complete** set of equivalent state pairs, including both valid and invalid states. This is much more difficult in general to compute than just those state pairs for the valid component of the circuit. For non-scan sequential testability, the complete set of equivalent states is required [34].

Once the invalid and equivalent states have been computed, they can be used in

Circuit	pairs	size	iter.	time
s344	2.59e+05	68268	5	57.79
s349	2.59e+05	68780	5	56.75
s382	5.35e+07	11928	93	541.16
s400	5.35e+07	11928	93	541.77
s420	1.68e+08	2040	8	3.76
s444	5.35e+07	12010	93	534.63
s641	1.18e+06	371	1	22.44
s713	1.18e+06	371	1	22.36
s838	4.42e+17	31985	8	56.19
tkey	1.06e+124	226	7	177.83
key	4.31e+68	1135	5	517.85
cbp.32.4	4.29e+9	95	1	1.51
minmax32	7.29e+28	1855	1	149.58

Table 4.2: Computation of Equivalent States.

pairs: number of equivalent state pairs
size: size of the BDD representation for the equivalent pairs
iter.: number of iterations in the FSM backwards traversal
time: CPU time in seconds for computing the invalid states

conjunction with combinational logic optimization techniques. At the time of this experiment, techniques for exploiting Boolean relations in multi-level logic optimization were not fully implemented. However, it should be noted that they are forthcoming [80]. In the absence of these techniques, only invalid states were used in this experiment. Invalid states were used as external don't-cares in MIS-II. The results for the ISCAS examples are shown in Table 4.3. **start** is the count of factored form literals of the initial circuit. Each circuit is then optimized using MIS-II with a new script called `script.rugged` without the invalid states don't-cares. This script contains the new don't-care simplification procedure `full_simplify` described in [81]. `script.rugged` also uses improved factorization procedures for area optimization. The results without sequential don't-cares are reported under the column labeled **script1**. Then the invalid states were used as external don't-cares with the command `full_simplify` to further reduce the size of the circuit. The factored form literal counts after this step are reported under the column labeled **script2**. In general, the amount of simplification gained by using the sequential don't-cares depends directly on the specific example and the effectiveness of the logic minimization technique.

Circuit	#L	start	script1	script2
s344	15	269	150	140
s349	15	273	151	139
s382	21	306	162	151
s400	21	320	157	146
s420	16	336	159	46
s444	21	352	152	142
s641	19	539	185	151
s713	19	591	186	151
s838	32	670	320	47
tkey	228	3865	3662†	3661†
key	228	3865	3669†	3661†
cbp.32.4	32	480	416	400
minmax32	96	1874	1594	1534

Table 4.3: Results on Sequential Optimization.

#L: number of latches
start: starting point (literals in factored form)
script1: results after MIS-II with script.rugged
script2: further optimization with `full_simplify` [81] using sequential don't-cares
†: `full_simplify` or script did not complete

In Table 4.4, results are given for computing only the single-cycle equivalent state-pairs using Equation 4.6. In most cases, the CPU time required for computing only the single-cycle equivalent state-pairs is considerably less. For example in s444, computing all equivalent state-pairs requires 534.63 seconds whereas the single-cycle equivalent state-pairs can be computed in only 2.77 seconds. As can be seen from the table, a reasonable subset of equivalent state-pairs can be derived using only single-cycle equivalence. In some cases, the computation is actually more expensive because the BDD representation for the single-cycle equivalent state-pairs is larger than the BDD representation for all equivalent state-pairs. tkey is such an example.

Circuit	all pairs		single-cycle pairs	
	pairs	time	pairs	time
s344	2.59e+05	57.79	9.42e+04	4.04
s349	2.59e+05	56.75	9.42e+04	3.06
s382	5.35e+07	541.16	4.39e+07	5.09
s400	5.35e+07	541.77	4.39e+07	5.12
s420	1.68e+08	3.76	1.65e+08	1.47
s444	5.35e+07	534.63	4.39e+07	2.77
s641	1.18e+06	22.44	1.18e+06	7.14
s713	1.18e+06	22.36	1.18e+06	7.02
s838	4.42e+17	56.19	4.23e+17	18.95
tkey	1.06e+124	177.83	6.84e+84	260.04
key	4.31e+68	517.85	4.31e+68	313.01
cbp.32.4	4.29e+09	1.51	4.29e+09	1.23
minmax32	7.29e+28	149.68	7.29e+28	116.01

Table 4.4: Computation of Single-Cycle Equivalent States.

pairs: number of equivalent state pairs
time: CPU time in seconds for computing the invalid states

4.6 Conclusions

Efficient algorithms for extracting global state-space information from a logic-level description of a sequential circuit were presented in this chapter. These algorithms are based on a symbolic representation of the state-space using binary decision diagrams. In particular, algorithms have been developed for computing invalid and equivalent state information for large circuits such that the information can be used as don't-care conditions for combinational logic synthesis. In contrast with previous methods based on explicit state diagram construction or cube enumeration, these algorithms can be used on much larger circuits that may contain both control and data-path circuitry.

Chapter 5

Implicit Manipulation of Equivalence Classes

5.1 Introduction

In Chapter 4, it was shown how binary decision diagrams (BDD's) [20] and the concept of implicit state enumeration [25] can be used to extract global state-space information (*e.g.* invalid and equivalent states) for sequential optimization. The use of BDD's and implicit state enumeration makes it possible to develop efficient algorithms that can be applied to very large circuits. Recently, it has become apparent that many problems in synthesis, testing, and verification are intimately related in that they are often fundamentally dependent on the same set of basic logic manipulations. Therefore, efficient techniques developed in this area can be viewed as "core" technologies that can be applied to a wide spectrum of problems in all three areas. For example, binary decision diagrams can be used to represent and manipulate logic functions and relations very efficiently and the concept of implicit state enumeration can be used to solve many problems requiring spatial or temporal analysis of a large state-space. Since many algorithms in synthesis, testing, and verification make use of such basic core computations extensively, new developments in this area are extremely important.

While BDD's and implicit enumeration provide the basic machinery and concepts to manipulate functions, relations, and state-spaces efficiently, analogous machinery for representing and manipulating *equivalence classes* efficiently (other than straightforward

truth-table enumeration) have not been developed. In many interesting applications, the ability to represent and manipulate equivalence classes is in fact the fundamental bottleneck. Example applications where efficient methods for representing and manipulating equivalence classes are crucial include *communication complexity calculation* and manipulation of sequential machines. In communication complexity calculation, the problem is to compute the amount of "real" information being transmitted between two combinational logic blocks. This calculation forms the core (and the most expensive) computation in many problems, such as functional decomposition for logic synthesis [49, 51] and logic partitioning [8]. The problem of calculating communication complexity is equivalent to the problem of computing the number of equivalence classes across the partition. Consider the case of a two block partition with n signals going from one block to the other. A number of binary patterns from the first block may be considered "equivalent" with respect to the observable behavior of the second block. Each set of these equivalent binary patterns form an equivalence class. In effect, these equivalent binary patterns partition the space into a number of (disjoint) equivalent classes. Using implicit enumeration and characteristic function techniques, only pairwise equivalence relationships can be derived. However, techniques have not been developed for deriving or representing the equivalence classes efficiently. One approach is to represent each equivalence class separately with a characteristic function in BDD form. Although each characteristic function may be reasonably compact, there can be an exponential number (*i.e.* 2^n) of equivalence classes in the worst case. Since the number of signals is usually large, this severely limits the analysis to only small problems. Similar problems arise in the manipulation of state machines under equivalent states.

In this chapter, new methods based on binary decision diagrams are presented for representing and manipulating equivalence classes efficiently. A new concept, the *equivalence class characterization function*, is introduced to implicitly represent equivalence classes. Informally, an equivalence class characterization function is effectively an *encoding function* that encodes all equivalent binary patterns into a unique binary pattern. An equivalence class characterization function has several interesting properties that make it very useful for representing and manipulating classes. One is that all equivalence classes can be implicitly represented with a single multi-output function with at most n output variables rather than possibly 2^n individual functions if each equivalence class is represented by a separate characteristic function. This makes it possible to represent the all equivalence classes for very large Boolean spaces implicitly, since the number of output variables required to represent

the equivalence class characterization function is bounded by the number of dimensions of the Boolean space considered. Another property of the equivalence class characterization function is that the number of equivalences classes can be readily determined by computing the size of the range (which can be easily determined by a simple traversal of the BDD representation). To compute the equivalence class characterization function, a new Boolean operator called the *compatible projection operator* is introduced. Conceptually, the compatible projection operator is used to select a single member from each equivalence class to "characterize" the class. In manipulating equivalence classes symbolically, the compatible projection operator is used to derive implicitly an encoding function from the equivalence relation that encodes the equivalence class information symbolically. The resulting function produced by the compatible projection operator is well-defined for any fixed variable ordering. The compatible projection operator can also be seen as a basic mechanism for selecting a compatible mapping corresponding to any relation. An efficient algorithm using binary decision diagrams has been developed for implementing the compatible projection operator.

The remainder of this chapter is organized as follows. In the next section, basic terminology for equivalence relations and equivalence classes are given. In Section 5.3, the importance of representing and manipulating equivalence classes efficiently is motivated using the problem of state minimizing large finite automata (*e.g.* over 10^{50} states) as an example application. Then the concept of an equivalence class characterization function is described as a means for representing equivalence classes efficiently. In Section 5.4, the compatible projection operator is described. In Section 5.5, two example applications are presented to demonstrate the use of the compatible projection operator and the concept of equivalence characterization function. In particular, the problem of reducing finite automata is further developed and the problem of computing communication complexity is described. Experimental results are given in Section 5.6 and concluding remarks are given in Section 5.7.

5.2 Definitions and Notation

In Chapter 2, relations were introduced as a means to represent logic behavior. In general, a relation is any subset over some Cartesian product of (possibly non-Boolean) finite domains $\mathcal{R} \subseteq D_1 \times D_2 \times \ldots \times D_n$. The cardinality of a finite set D is denoted by $|D|$.

Without loss of generality, all domains can be assumed to be over Boolean values for the purpose of analysis. Non-Boolean domains (*i.e.*, symbolic relations, *cf.* Chapter 2) can be handled by encoding elements of D with unique Boolean vectors of at least length k, where $k \geq \lceil \log_2 \mid D \mid \rceil$. For most applications considered, the choice of encodings and code lengths are usually pre-determined. For the sake of exposition, it is often convenient to speak in terms of a *binary* relation which is a subset of some Cartesian product $B^r \times B^n$, denoted as $\mathcal{R} \subseteq B^r \times B^n$. Henceforth, unless otherwise noted, relation and binary relation are used interchangeably. A important special case is the equivalence relation, as defined as follows.

Definition 5.1 *A relation* $\mathcal{R} \subseteq B^n \times B^n$ *is an* **equivalence relation** *on* B^n *provided* $\mathcal{R}$ *satisfies the following three properties:*

1. **reflexive:** $(\mathbf{x}, \mathbf{x}) \in \mathcal{R}, \forall \mathbf{x} \in B^n$;
2. **symmetric:** $(\mathbf{x}, \mathbf{y}) \in \mathcal{R} \Rightarrow (\mathbf{y}, \mathbf{x}) \in \mathcal{R}$;
3. **transitive:** $[(\mathbf{x}, \mathbf{y}) \in \mathcal{R}$ and $(\mathbf{y}, \mathbf{z}) \in \mathcal{R}] \Rightarrow (\mathbf{x}, \mathbf{z}) \in \mathcal{R}$;

$\mathbf{x}$ *is* **equivalent** *to* $\mathbf{y}$ *under* $\mathcal{R}$, *written* $\mathbf{x} \sim \mathbf{y}$, *if* $(\mathbf{x}, \mathbf{y}) \in \mathcal{R}$. *The set of possible mappings for an element* $\mathbf{x}$, $\mathcal{R}(\mathbf{x}) = \{\mathbf{y} \mid (\mathbf{x}, \mathbf{y}) \in \mathcal{R}\}$, *is also called the* **equivalence class** *of* $\mathbf{x}$. *This is also denoted as* $[\mathbf{x}]$.

An equivalence relation $\mathcal{R}$ $(\sim)$ on B^n induces a **partition** π on B^n, $\pi(B^n) = \{S_1, S_2, \ldots, S_q\}$. *i.e.*,

1. $\bigcup_i S_i = B^n$;
2. $S_i \cap S_j = \emptyset, \forall i \neq j$.

Each S_i is an equivalence class of $\mathcal{R}$.

A function can be expressed in a multi-output form $f : B^r \rightarrow B^n$, or it can be expressed in a characteristic function form $F \subseteq B^r \times B^n$. Given a function in multi-output from, its characteristic function can be derived as follows:

$$F = \prod_{i=1}^{n} (y_i \overline{\oplus} f_i) \tag{5.1}$$

where $f_1 \ldots f_n$ corresponds to the individual output functions of the multi-output function f. When $f_1 \ldots f_n$ correspond to the next-state functions of a finite state machine, then the characteristic function F is also referred to as the transition relation (*cf.* Chapter 4). A

function $F \subseteq B^r \times B^n$ expressed as a characteristic function can be converted to a multi-output form $f : B^r \rightarrow B^n$ if and only if $\forall \mathbf{x} \in B^r, \mid F(\mathbf{x}) \mid = 1$. The conversion can be done as follows:

$$f_i = (\exists Y)(F \cdot y_i) \tag{5.2}$$

where $Y = \{y_1, \ldots, y_n\}$ are the variables of the co-domain.

5.3 Efficient Representation of Equivalence Classes

In Chapter 4, efficient algorithms were developed for computing equivalent state relation E where

$$E \subseteq B^n \times B^n$$

and B^n represents the state-space. $E(\mathbf{x}, \mathbf{x}') = 1$ if and only if the states $\mathbf{x}$ and $\mathbf{x}'$ are equivalent. It was shown that the set of all equivalent state-pairs for machines with as much as 10^{50} states can be computed. However, this equivalence relation only provides information about pairwise relationships. While this is sufficient for the purpose of optimizing the combinational portions of the sequential circuit, not enough information is available to solve other problems like state minimization. Efficient machinery for computing and representing *equivalence classes* efficiently has not been developed.

Given a set of equivalence classes $[S_1], \ldots, [S_p]$, one possible approach is to represent each of the p equivalence classes *explicitly* with a separate characteristic function $f_{[S_1]}, \ldots, f_{[S_p]}$. Although characteristic function $f_{[S_i]}$ can be represented in compact form using BDD's, the number of equivalence classes in general can be exponential in the number of variables n (*i.e.* $p = 2^n$). Instead, a new concept is proposed for representing the equivalence classes *implicitly*.

Definition 5.2 *Let $E \subseteq B^n \times B^n$ be an equivalence relation. An* **equivalence class characterization function** *is any (multi-output) function $\xi : B^n \rightarrow B^m$ that satisfies the property $\xi(\mathbf{x}) = \xi(\mathbf{x}')$ if and only if $\mathbf{x} \sim \mathbf{x}'$.*

An equivalence class characterization function is effectively an *encoding function* that encodes all equivalent vertices to a unique image point in B^m such that two non-equivalent vertices have different images.

An equivalence class characterization function has the following obvious properties.

Property 5.1 *Let $E \subseteq B^n \times B^n$ be an equivalence relation with p equivalence classes. An equivalence class characterization function $\xi : B^n \rightarrow B^m$ for B^n can be constructed for any value $m \geq \lceil log_2 p \rceil$. In the worst case, $m \leq n$. For any such ξ, $\mid range(\xi) \mid = p$.*

This is important since p equivalence classes can always be represented with at most $\lceil \log_2 p \rceil$ functions, each representable in BDD form, rather than possibly 2^n functions. Further, given ξ, the number of equivalence classes can be simply determined by computing the size of the range(ξ), an easy computation with BDD's. An effective mechanism is now given to compute an equivalence class characterization function given E.

5.4 The Compatible Projection Operator

5.4.1 Definition and Properties

To derive an equivalence class characterization function that uniquely encodes each equivalence class, a new Boolean operator called the **compatible projection** operator is defined. When applied to an equivalence relation E, it uniquely selects a single element from each equivalence class to "characterize" the class. This is performed by finding a compatible function for the relation E with a special property reflecting the requirements imposed by equivalence relations.

Informally, the compatible projection operator is defined as follows. Given an equivalence relation $E \subseteq B^n \times B^n$, the compatible projection of E is the compatible function, written in characteristic function form:

$$F = \{(\mathbf{x},\mathbf{y}) \mid (\mathbf{x},\mathbf{y}) \in E, \mathbf{y} = SEL(\mathbf{x})\}$$

where $SEL(\mathbf{x})$ is any selection function that uniquely selects a member from the equivalence class of $\mathbf{x}$.

One SEL function is obtained by treating each binary vector in the Boolean space B^n as an integer, encoded in the traditional binary form, and choosing, for each equivalence class, the vector with the lowest integer value. However, a different ordering of the vectors in B^n can be defined. This can be achieved by using a *geometric distance* metric.

Definition 5.3 *Let B^n be a n-dimensional Boolean space. The* **geometric distance** *(or simply distance) between two vertices $\mathbf{x} \in B^n$ and $\mathbf{y} \in B^n$ for a given variable ordering is*

defined as

$$d(\mathbf{x},\mathbf{y}) = \sum_{i}^{n} \mid x_i - y_i \mid 2^{n-i}$$

Lemma 5.1 *Given a reference vertex $\alpha \in B^n$ and any variable ordering, the geometric distance metric induces a total ordering on the elements in B^n.*

Proof: It is sufficient to prove that $\mathbf{x} \neq \mathbf{x}'$ implies $d(\alpha,\mathbf{x}) \neq d(\alpha,\mathbf{x}')$. Each of the expressions $d(\alpha,\mathbf{x})$ and $d(\alpha,\mathbf{x}')$ can be interpreted as the binary representation of some integer. Since the binary representation of integers is unique, the integer value for each minterm $\mathbf{x} \in B^n$ is also unique. Therefore, the elements in B^n are ordered by their integer values. ■

Using the distance operator, an ordering on the vertices of a Boolean space relative to some reference vertex α can be defined.

Definition 5.4 *Given $\alpha \in B^n$, $C \subseteq B^n$, the* **closest interpretation** *of α in C for a given variable ordering is defined as*

$$\perp(\alpha,C) = \arg\min_{\mathbf{x}\in C} d(\alpha,\mathbf{x})$$

Lemma 5.2 *The definition of closest interpretation, $\perp$, relative to a reference vertex α, is unique for any given variable ordering.*

Proof: Since the distance metric maps each element in C to a unique integer, using α as a reference point, the element in C with the lowest integer mapping is also unique. ■

The compatible projection operator can now be formally defined.

Definition 5.5 *Let $E \subseteq B^n \times B^n$ be an equivalence relation and $\alpha \in B^n$ be a reference vertex. The* **compatible projection**, *or simply* **cprojection**, *of E relative to α, denoted as* **cprojection**(E,α), *is the compatible function F defined as follows:*

$$F = \textbf{cprojection}(E,\alpha) = \{(\mathbf{x},\mathbf{y}) \mid \mathbf{y} = \perp(\alpha, E(\mathbf{x}))\}$$

Lemma 5.3 *Given $E \subseteq B^n \times B^n$, $\alpha \in B^n$, and any variable ordering, the compatible mapping $F =$* **cprojection**(E,α) *is* **well-defined.**

Proof: For every $\mathbf{x} \in B^n$, $\perp(\alpha, E(\mathbf{x}))$ is unique. Therefore, the compatible function F using the $\perp$ operator is well-defined. ■

This result says that a unique function F is derived. In fact, it is an equivalence class characterization function for E.

Theorem 5.4 *Let $E \subseteq B^n \times B^n$ be an equivalence relation. Then $F =$* **cprojection**(E, α) *with respect to any reference vertex α is an equivalence class characterization function for E.*

Proof: $\mathbf{x} \sim \mathbf{x}'$ in E implies $E(\mathbf{x}) = E(\mathbf{x}')$. Given that $\perp(\alpha, E((\mathbf{x})) = \perp(\alpha, E((\mathbf{x}'))$, then $F(\mathbf{x}) = F(\mathbf{x}')$. ■

5.4.2 An Efficient Algorithm using BDD's

An efficient recursive algorithm for computing **cprojection**(E, α), which exploits recently developed techniques for efficient BDD implementation, is now given. In the description of the algorithm, E is an equivalence relation with variables $\mathbf{x} = \{x_1, \ldots, x_n\}$ and $\mathbf{y} = \{y_1, \ldots, y_n\}$, and α is the characteristic function for the reference vertex (minterm) in B^n with variables $\mathbf{y} = \{y_1, \ldots, y_n\}$. The algorithm traverses the BDD graph structure directly and uses caching techniques to remember intermediate results. This enables the algorithm to be performed in a bottom-up fashion on large BDD's. The pseudo code for the recursive procedure is shown in Figure 5.1. The procedure takes two arguments: the equivalence relation E and a reference vertex α. In Line 1, if $E = 1$, it means that $\forall \mathbf{x} \in B^n, E(\mathbf{x}) = 1$. Therefore, the output mapping is simply α. In Lines 2 and 3, if either $\alpha = 0$ or $E = 0$, the relation E is returned. In Line 4, if $\alpha_{y1} = 0$, it means that the "reference" literal is $\alpha_1 = \overline{y_1}$ (recall that α is a minterm). Otherwise, the reference literal is $\alpha_1 = y_1$. The term E_{α_1} is E cofactored with the reference literal α_1. $\gamma = (\exists \mathbf{y}) E_{\alpha_1}$ represent the set of $\mathbf{x}'s$ with at least one output choice $\mathbf{y}$ where the value of the variable y_1 is α_1. For the set of $\mathbf{x}$ not in γ, an output mapping with literal $\overline{\alpha_1}$ at variable y_1 must be chosen. This can be computed with the following expression:

$$\begin{aligned} \textbf{cprojection}(E, \alpha) = {} & \alpha_1 \cdot \textbf{cprojection}(E_{\alpha_1}, \alpha_{\alpha_1}) + \\ & \overline{\alpha_1} \cdot \overline{\gamma} \cdot \textbf{cprojection}(E_{\overline{\alpha_1}}, \alpha_{\alpha_1}) \end{aligned} \tag{5.3}$$

Theorem 5.5 *Given an equivalence relation $E \subseteq B^n \times B^n$, a reference vertex $\alpha \in B^n$, and a variable ordering on* $\mathbf{y}$, *Equation 5.3 computes the* **cprojection** *of E with respect to α.*

Proof: The prove is by induction. Assume the statement is true for all j up to k. For the base case, $j = 0 \Rightarrow \alpha = 0 \Rightarrow \mathbf{cprojection}(E, \alpha) = E$. By the induction hypothesis, $\alpha_1 \cdot \mathbf{cprojection}(E_{\alpha_1}, \alpha_{\alpha_1})$ returns a unique mapping $\mathbf{y}$ closest to α for every $\mathbf{x}$ in E with the variable $y_1 = \alpha_1$ if such a mapping exists. $\gamma = (\exists \mathbf{y}) E_{\alpha_1}$ computes exactly the set of $\mathbf{x}$ where such a mapping exist. $\overline{\gamma}$ is the set of $\mathbf{x}$ that does not have an output mapping with $y_1 = \alpha_1$. Similarly, $\overline{\alpha_1} \cdot \mathbf{cprojection}(E_{\overline{\alpha_1}}, \alpha_{\alpha_1})$ returns a unique mapping $\mathbf{y}$ closest to α for every $\mathbf{x}$ in E with the variable $y_1 = \overline{\alpha_1}$ if such a mapping exists. $\overline{\gamma} \cdot \overline{\alpha_1} \cdot \mathbf{cprojection}(E_{\overline{\alpha_1}}, \alpha_{\alpha_1})$ deletes from $\overline{\alpha_1} \cdot \mathbf{cprojection}(E_{\overline{\alpha_1}}, \alpha_{\alpha_1})$ exactly those $\mathbf{x}$ that are already in $\alpha_1 \cdot \mathbf{cprojection}(E_{\alpha_1}, \alpha_{\alpha_1})$. Therefore, the union of these sets will produce a unique mapping closest to α. ■

In the actual implementation, the procedure begins at the root of the BDD. It then recursively computes the cprojection of its left branch and its right branch. In the BDD package developed at Berkeley [72], a unique cache table [11] is used for caching ITE operations. For the cprojection operation, two additional auxiliary caches are used. The first one caches previously computed results for $\gamma = (\exists \mathbf{y}) E_{\alpha_1}$ at each BDD node. The second auxiliary cache is used to remember intermediate cprojection results so that previous computations of the cprojection operator with the same arguments are not repeated. These caches use the similar management scheme as the unique cache table for the ITE operation.

5.5 Example Applications

5.5.1 Communication Complexity

In the example in Figure 5.2, a hierarchically-defined combinational network implemented in two separate combinational blocks $N1$ and $N2$ is shown. $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$ are the primary inputs to $N1$. The signals going from $N1$ to $N2$ are $\mathbf{y} = \{y_1, y_2, \ldots, y_p\}$. These signals are used to encode the patterns of $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$ for use by $N2$. $N2$ additionally has other inputs $i = \{i_1, i_2, \ldots, i_m\}$. From the point of view of the block $N2$, two vertices (binary patterns) $\mathbf{x} \in B^n$ and $\mathbf{x}' \in B^n$ over the variables $x_1 x_2 \ldots x_n$ (the primary inputs of $N1$) are *equivalent* ($\mathbf{x} \sim \mathbf{x}'$) at the primary outputs $\mathbf{z} = \{z_1, z_2, \ldots, z_q\}$ if $\forall i, \mathbf{z}(i, \mathbf{x}) = \mathbf{z}(i, \mathbf{x}')$. The number of equivalence classes is bounded by 2^n. The *communica-*

```
function cprojection(E, α) {
1.    if (E = 1) return Eα;
2.    if (E = 0) return E;
3.    if (α = 0) return E;
4.    if (α_{y_1} = 0) let α_1 = ȳ_1;
5.    else if (α_{ȳ_1} = 0) let α_1 = y_1;
6.    else assert(failure);
7.    let γ = (∃y)E_{α_1};
8.    if (γ = 1) return α_1 cprojection(E_{α_1}, α_{α_1});
9.    else if (γ = 0) return ᾱ_1 cprojection(E_{ᾱ_1}, α_{α_1});
10.   else return α_1 cprojection(E_{α_1}, α_{α_1})
          + ᾱ_1 γ̄ cprojection(E_{ᾱ_1}, α_{α_1});
}
```

Figure 5.1: A Recursive Algorithm for the Compatible Projection Operator.

tion complexity from $N1$ to $N2$ is the minimum number of variables required to encode the equivalence classes. For this calculation, an equivalence relation

$$E \subseteq B^n \times B^n$$

can be computed. If all the inputs to the block $N2$ come from $N1$ (*i.e.*, $i = \{\}$), then the equivalence relation is

$$\begin{aligned} E(\mathbf{x},\mathbf{x}') &= (z_1(\mathbf{x})\overline{\oplus}z_1'(\mathbf{x}'))(z_2(\mathbf{x})\overline{\oplus}z_2'(\mathbf{x}'))\ldots(z_q(\mathbf{x})\overline{\oplus}z_q'(\mathbf{x}')) \\ &= \prod_{j=1}^{q}(z_j(\mathbf{x})\overline{\oplus}z_j'(\mathbf{x}')) \end{aligned} \tag{5.4}$$

Here, $\mathbf{x}' = \{x_1', x_2', \ldots, x_n'\}$ is a set of duplicated variables corresponding to the variables in $\mathbf{x}$. If the set of variables i is not empty, meaning there are primary inputs that come into $N2$ directly, then the expression for computing the equivalence relation can be modified as follows:

$$\begin{aligned} E(\mathbf{x},\mathbf{x}') &= (\forall i)(z_1(i,\mathbf{x})\overline{\oplus}z_1'(i,\mathbf{x}'))\ldots(z_q(i,\mathbf{x})\overline{\oplus}z_q'(i,\mathbf{x}')) \\ &= (\forall i)\prod_{j=1}^{q}(z_j(i,\mathbf{x})\overline{\oplus}z_j'(i,\mathbf{x}')) \end{aligned} \tag{5.5}$$

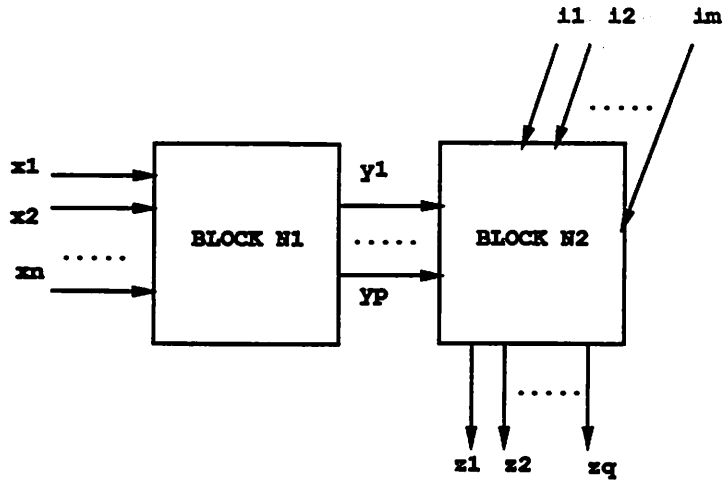


Figure 5.2: Calculating Communication Complexity.

$$= \prod_{j=1}^{q} (\forall i)(z_j(i,\mathbf{x}) \overline{\oplus} z'_j(i,\mathbf{x}'))$$

Since Boolean AND and universal quantification commute, the $(\forall i)$ can be moved inside the formula for more efficient computation. An equivalence class characterization function ξ using any reference vertex $\alpha \in B^n$ is therefore

$$\xi = \mathbf{cprojection}(E, \alpha). \tag{5.6}$$

The number of equivalence classes from $N1$ to $N2$ is simply

$$\mathtt{classes} = \mid (\exists \mathbf{x})\xi(\mathbf{x},\mathbf{x}') \mid . \tag{5.7}$$

The communication complexity is simply

$$\mathtt{com\_complexity} = \lceil \log_2 C \rceil. \tag{5.8}$$

5.5.2 Reduction of Finite Automata

In this section, the problem of state minimization for large sequential machines is considered. In practice, large sequential circuits may not be state minimal. This is especially

true when the sequential circuits are automatically compiled from high-level descriptions. There are well known algorithms for state minimization when a state transition graph model can be extracted, but these are only feasible for machines with at most a few hundred states. Using the machinery developed in the previous sections, an exact algorithm has been devised that can feasibly state minimize finite state machines of practically any size (*e.g.*, over 10^{50} states) as long as the equivalent state-pairs can be computed. The outline of the minimization algorithm is as follows:

1. Given a finite state machine M, construct a transition relation $T \subseteq B^r \times B^n \times B^n$ using BDD's such that $T(i, \mathbf{x}, \mathbf{y}) = 1$ if and only if the state $\mathbf{y}$ can be reached in exactly one state transition from state $\mathbf{x}$ when input i is applied. Construct also an output relation $\mathcal{O} \subseteq B^r \times B^n \times B^m$ such that $\mathcal{O}(i, \mathbf{x}, o) = 1$ if and only if the output pattern o can be produced when the input i is applied at present-state $\mathbf{x}$.

2. Compute the equivalence relation $E \subseteq B^n \times B^n$ corresponding to the set of all equivalent state-pairs. The relation E is computed by building a product machine $M^* = M_1 \otimes M_2$ and using algorithms described in Chapter 4.

3. Use the compatible projection operator to find an equivalence class characterization function $\xi : B^n \rightarrow B^n$, in relation form $\xi(\mathbf{x}, \mathbf{y})$, that maps all equivalent states to the same state. Modify $T(i, \mathbf{x}, \mathbf{y})$ using the characterization function ξ.

In the second step, the equivalence relation E is represented in BDD form. Although there may be a very large number of equivalent state-pairs, as can be seen in Chapter 4, equivalent state-pairs can be computed for very large sequential circuits. The efficiency is dependent on the regularity of the underlying structure. On the third step, ξ is computed as

$$\xi = \mathbf{cprojection}(E, \mathbf{x}_0), \tag{5.9}$$

where $\mathbf{x}_0$ is the reset state. $\mathbf{x}_0$ is used as the reference vertex to guarantee that the reset state-code is retained (*i.e.*, all states equivalent to the reset state will be re-assigned the reset state-code). However, this is not a necessary condition. The reduced set of states can be computed as

$$Q = \text{set of reduced states} = (\exists \mathbf{x}) \xi(\mathbf{x}, \mathbf{y}), \tag{5.10}$$

It is the *range* of ξ. Using ξ, the transition relation of the finite state machine can be simplified as follows:

$$T_{min}(i, \mathbf{x}, \mathbf{y}) = (\exists \mathbf{x}'\mathbf{y}')[T(i, \mathbf{x}', \mathbf{y}') \cdot \xi(\mathbf{x}', \mathbf{x}) \cdot \xi(\mathbf{y}', \mathbf{y})]. \tag{5.11}$$

Similarly, the state minimized output relation can be obtained as follows:

$$\mathcal{O}_{min}(i, \mathbf{x}, o) = (\exists \mathbf{x}')[\mathcal{O}(i, \mathbf{x}', o) \cdot \xi(\mathbf{x}', \mathbf{x})] \tag{5.12}$$

5.6 Experimental Results

The capabilities of the proposed techniques are demonstrated by applying the techniques to the problems of communication complexity analysis for large combinational circuits and exact state minimization of large sequential circuits. The algorithms make use of the concept of the equivalence class characterization function and the compatible projection operator described in this chapter, which have been efficiently implemented using BDD's. All experimental results presented were measured on a DECstation 5000 and the CPU times presented are quoted in seconds.

For communication complexity analysis, a number of large MCNC logic benchmarks were used. The experiments were designed to show the limitation of conventional explicit manipulation methods and expose the need for representing and manipulating equivalence classes efficiently. For each benchmark, a cut through the network was found starting at the gate-level description. The set of all equivalent pairs of vector patterns along the cut was computed using Equation 5.5. This equivalence relation, which corresponds to the set of equivalent pairs of bit patterns, is represented using a characteristic function in BDD form. In Table 5.1, some results for the benchmarks are given. In some examples, the number of equivalent pairs is quite large (over 10^{32}). An equivalence class characterization function is then derived implicitly using the compatible projection operator. In Table 5.2, the number of equivalence classes is reported under the column labeled **classes**. As can be seen, the compatible projection operation is extremely fast. Using the compatible projection operator, the number of equivalence classes can be computed for all benchmarks tested.

For the minimization of state machines, the benchmarks used were obtained from various sources in the form of gate-level descriptions. The examples **s208** and **s298** are

Circuit	I/O	vars	elem.	pairs	size	time
alu1a	50/5	13	8192	571776	424	51.36
duke2	44/29	11	2048	4096	31	4.46
misex1	16/7	4	16	32	10	0.24
misex2	50/18	13	8192	5.92e+7	16	0.42
misex3	28/14	7	128	5288	249	9.94
misex3c	28/14	7	128	224	28	3.48
sao2-hdl	20/4	5	32	334	22	0.84
seq	82/35	21	2.1e+6	2.0e+12	773	223.01
alu4	28/8	7	128	128	22	3.15
ampbsm	150/66	38	2.75e+11	1.22e+19	64	9.28
amppint2	170/66	43	8.8e+12	4.5e+15	103	6.31
ampxhdl	124/40	31	2.15e+9	3.52e+13	52	1.64
apex6	270/99	68	2.95e+20	7.76e+32	439	7.25
dflgrcbl	216/65	54	1.80e+16	1.76e+29	3150	5.76
fconrcbl	124/35	31	2.15e+9	1.65e+17	339	1.79
k2	90/45	23	8.39e+6	1.57e+11	158	31.61
kcctlcb3	162/44	41	2.20e+12	1.44e+20	230	3.22
sbiucbl	80/35	20	1.05e+6	9.52e+8	110	2.72
tfaultcbl	154/35	39	5.50e+11	1.63e+21	776	2.55
vda	34/39	9	512	1232	49	5.25

Table 5.1: Computing and Representing Equivalent Pairs.

I/O: number of primary inputs and outputs
vars: number of variables along the cut
elem.: number of possible bit patterns along the cut
pairs: number of equivalent pairs of vertices
size: number of nodes in BDD of corresponding equivalence relation
time: CPU time in seconds for computing the set of equivalent pairs

from the ISCAS sequential benchmark set. The example `tlc` is a traffic light controller. The examples `vit3` and `viterbi` are control circuits from the VITERBI speech recognition processor chip [86]. The example `key` is derived from a control circuit that implements the key encryption algorithm in the data encryption standard (DES) chip [91]. It contains a large number of state-bits. The examples `mkey` and `tkey` are derived from `key` by considering a subset of outputs (namely the control outputs). These examples vary in complexity with the largest one having 228 latches and over 10^{68} states. Since BDD's are used, the number of state-bits is no longer the bottleneck; the "real" complexity of the circuit is actually

Circuit	elem.	classes	size	time
alupla	8192	338	322	0.09
duke2	2048	1024	32	0.01
misex1	16	8	12	0.01
misex2	8192	2	27	0.01
misex3	128	33	98	0.03
misex3c	128	80	29	0.01
sao2-hdl	32	8	30	0.01
seq	2.1e+6	163	498	0.42
alu4	128	128	22	0.01
ampbsm	2.75e+11	18432	90	0.01
amppint2	8.8e+12	1.72e+10	112	0.01
ampxhdl	2.15e+9	131072	67	0.01
apex6	2.95e+20	1.42e+9	2174	0.60
dflgrcbl	1.80e+16	655360	1490	0.69
fconrcbl	2.15e+9	80	225	0.17
k2	8.39e+6	2664	166	0.03
kcctlcb3	2.20e+12	1.69e+7	296	0.04
sbiucbl	1.05e+6	2560	118	0.02
tfaultcbl	5.50e+11	2624	634	0.22
vda	512	272	51	0.01

Table 5.2: Computing the Communication Complexity.

elem.: number of possible bit patterns along the cut
classes: number of equivalence classes
size: number of nodes in BDD of corresponding equivalence class characteristic function
time: CPU time in seconds for performing the compatible projection operation

dependent on the structure and regularity of the problem.

The experiment for state minimization was undertaken as follows. For each benchmark sequential circuit, starting at the gate-level, the set of all equivalent state-pairs is computed using the algorithms described in Chapter 4. These algorithms find all possible equivalent state-pairs and represent them as a characteristic function in BDD form.

An implicit encoding function is then derived using the compatible projection operator. The transition functions of the reduced machine are accordingly constructed. The number of reduced states after state minimization is usually much less than the number of states in the initial form. Thus, the reduced states can potentially be re-encoded with

Circuit	I/O/Lits(fac)	states	pairs	bits	CPU1
s208	11/21/166	256	3310	8	0.45
s298	3/6/244	16384	510000	14	35.88
tlc	3/5/324	1020	25400	10	5.41
vit3	11/4/880	512	18400	9	5.08
viterbi	11/34/1372	4100	10700	12	17.50
mkey	258/10/3676	4.31e+68	1.11e+130	228	146.16
tkey	258/20/3686	4.31e+68	1.06e+124	228	177.83
key	258/193/3865	4.31e+68	4.31e+68	228	517.85

Table 5.3: Computation of Equivalent State Pairs.

I/O/Lits(fac): number of primary inputs and outputs, and literals of the circuit
states: number of states in the initial machine
pairs: number of equivalent state-pairs
bits: initial number of state-bits
CPU1: CPU time for computing the set of equivalent state-pairs

Circuit	states	classes	bits	lower	CPU2
s208	256	40	8	6	0.02
s298	16384	8060	14	13	1.58
tlc	1020	254	10	8	0.14
vit3	512	15	9	4	0.01
viterbi	4100	3120	12	12	0.05
mkey	4.31e+68	1.68e+07	228	24	1.72
tkey	4.31e+68	1.76e+13	228	44	2.19
key	4.31e+68	4.31e+68	228	228	2.47

Table 5.4: State Minimization Results.

states: number of states in the initial machine
classes: number of equivalence classes and states after state minimization
bits: initial number of state-bits
lower: lower bound on minimum code length for re-encoding reduced machine
CPU2: CPU time for performing state minimization

significantly fewer number of state bits.

Tables 5.3 and 5.4 show the results of the experiment. Table 5.3 shows the results of computing the equivalent state-pairs. Here, all states, both reachable and unreachable states are considered. This is more general but equivalent states analysis for the reachable subset is a trivial extension. The largest examples are mkey, tkey, and key, each of which

has 4.31×10^{68} possible states. The example mkey has over 10^{130} equivalent state-pairs. The CPU times reported used efficient implementations [62, 89] of the algorithms.

Using the compatible projection operator, it was possible to implicitly compute the number of equivalence classes corresponding to the equivalence relation and merge all equivalent states in the finite state machine. The column labeled **EQV.classes** indicates the number of equivalence classes for each finite state machine example. This is also the number of states after state minimization. For example, the machine mkey was state minimized from 4.31×10^{68} states to only 1.68×10^{7} states. The results of state minimization are shown in Table 5.4. The CPU times for computing the equivalence classes (using compatible projection) and performing state reduction are indicated in the column labeled **CPU2**. The CPU time required for performing symbolic class manipulation is modest relative to the equivalent state-pair calculation.

Because the number of state patterns after state minimization may be considerably less, it is possible to encode the reduced states with fewer state-bits. For example, the minimum code length to re-encode the reduced machine tkey is 44 state-bits, but the original number of state-bits is 228.

In all cases, the CPU time for computing the equivalence relation, in both "communication complexity" analysis and minimization of state machines, *strictly dominates* the overall time for computing the characterization function.

5.7 Conclusions

New concepts and machinery for representing and manipulating equivalence classes efficiently have been presented in this chapter. Specifically, a new representation called an equivalence class characterization function and a new operator called compatible projection were presented. The results of this work can be applied towards a number of applications in synthesis where there is a need to manipulate equivalence classes efficiently. Two example problems were used to demonstrate the usefulness of this work, namely communication complexity calculation and finite state machine reduction. The latter problem has direct application to sequential optimization. On some example circuits, the state-space can be greatly reduced if state minimization was performed. However, previous techniques for state minimization are only applicable to sequential circuits with small state spaces. It should be possible to apply the concepts described in this chapter to a number of other applications;

the capabilities of these concepts have just begun to be exploited.

Chapter 6

Minimization of State Latches

6.1 Introduction

In this chapter, another symbolic manipulation technique is presented. Specifically, the problem of removing redundant state-bits, unnecessary to distinguish the state-codes of the reachable states, is considered. A sequential circuit description, as an interconnection of combinational logic gates and synchronous latches, is given. A reset bit-pattern is also given. This technique can be used to reduce the number of encoding variables in situations where minimal code length is crucial. It can also be used as an optimization technique for sequential circuits.

Given a sequential circuit with N latches, there are 2^N possible state patterns (or simply states). However, only a subset of these states are reachable from the reset state. Using recently developed concepts from sequential verification [27, 89, 24] that are based on binary decision diagrams and symbolic execution techniques, the set of reachable sets can be computed for sequential circuits with extremely large state-spaces. The main limitation is the ability of BDD's to implicitly represent the set of reachable states as a characteristic function. Note that the real bottleneck is the irregularity of the state-space rather than its size.

In many sequential circuits, especially those with a large number of latches, the subset of states that are reachable is often significantly smaller then 2^N (where N is the number of latches). This is the case with many of the circuit examples from the ISCAS-89 sequential test benchmark set. Therefore, it is possible to re-encode the set of reachable states with fewer state-bits. Surprisingly, this property is not uncommon in realistic designs.

Sequential logic optimization algorithms for area and performance optimization, such as retiming [57] and retiming-and-resynthesis [66], are known to increase the number of state-bits dramatically. Too many state-bits can be quite costly in terms of area.

One powerful strategy for reducing the number of latches (code length) is to remove state-bits that are not necessary to distinguish the different state-codes. These are called *redundant* state-bits. Consider the following trivial example. Suppose $Q = \{001, 100, 111\}$ is the set of reachable states. The first state-bit is redundant since the remaining state-bits can uniquely determine the state-codes: $\hat{Q} = \{01, 00, 11\}$. In effect, the states have been re-encoded using a subset of their bit patterns. This kind of re-encoding will almost always lead to a more efficient representation or implementation of the sequential circuit. The problem is to find a maximum (or maximal) subset of these state-bits to eliminate such that the state-codes induced by the remaining state-bits are unique. Berthet *et al.* [9] give conditions for detecting and removing these redundant state-bits, but no algorithms or results are given.

A straightforward, but naive, approach is to test and remove one state-bit at a time. Unfortunately, this is suboptimal. The problem is that the removal of one redundant state-bit may preclude the removal of other state-bits. In this chapter, a new BDD-based branch and bound algorithm for finding the maximum set of redundant state-bits is presented. The algorithm is guaranteed to produce an *exact* solution upon completion. As with most branch and bound algorithms, effective heuristics for branching and bounding are crucial. Here, a branching heuristic, the selection of the most *unate* (the least binate) redundant state-bit for removal, is used. This heuristic is based on the observation that the most unate state-bit is least effective in distinguishing state pairs. An effective bounding technique based on the concept of *maximal removable set* has also been developed. Empirically, it has been found that this bounding technique is extremely effective in pruning the search space. Because the proposed algorithm is based on the use of BDD's, and it examines implicitly the states rather than explicitly, it is applicable to any set of reachable states that can be computed and represented using BDD's. Hence, it is applicable to a very broad class of sequential designs.

The algorithms have been applied to a variety of sequential benchmark circuits. After the maximum set of redundant state-bits for removal is computed, the information can be used to reduce both the BDD representation of the transition relation for verification and the gate-level representation of the circuit for implementation. For design verification,

a reduction in the BDD representation of the transition relation can significantly reduce the verification time when checking for multiple design properties. This is important since many design properties may be checked during a verification session. Removal of redundant state-bits can also lead to much more efficient implementations. This is partly due to the savings in latches. However, since a subset of state bits are no longer needed, the hardware of the corresponding next state functions can be also discarded. This represents a more significant savings in area. Although some additional logic may be incurred for adapting to the new state-codes, the overall area is almost always reduced. Also, with a significant reduction in latch count, the resulting sequential designs tend to be more easily testable.

The remainder of the chapter is organized as follows. In Section 6.2, the redundant state-bit removal problem is analyzed. In Section 6.3, an exact algorithm for finding the maximum removable set of state-bits is presented. In Section 6.4, experimental results are presented on a variety of sequential benchmark circuits using the described techniques. In a surprising number of cases, the number of state-bits is significantly reduced. Results are also given on the effects of latch removal on reducing the size of transition relations in BDD form and the area of the circuit-level implementation.

6.2 Redundant Encoding Variable Removal

For greater generality, the set of elements under consideration need not be states. Instead, any generic set of encoded elements A over some finite Boolean domain B^n can be handled. The lower bound on the number of variables required to re-encode A with a unique code, $\lceil \log_2 \#A \rceil$, is often less than n. One technique for reducing the number of encoding variables is to eliminate variables that are not necessary to distinguish the codes given to the elements. These variables are called redundant variables.

Definition 6.1 *Let $A \subseteq B^n$ be a set of binary patterns over the Boolean space B^n and $a_1 \ldots a_n$ be the corresponding encoding variables. A variable a_i is said to be* **redundant** *if each element after its removal is still unique. It is also called a* **free** *variable.*

Definition 6.2 *Let $A \subseteq B^n$ be a set of binary patterns over the Boolean space B^n and $a_1 \ldots a_n$ be the corresponding variables. A* **permissible removal set** *is a set of redundant variables $\{a_i, a_j, \ldots, a_k\}$ such that their removal does not affect the uniqueness of each code.*

Definition 6.3 *A permissible removal set $\{a_i, a_j, \ldots, a_k\}$ is said to be* **maximal** *if it is not fully contained in any other permissible removal set. A permissible removal set is said to be the* **maximum** *if it is the largest permissible removal set.*

Example 6.1 Let A be $\{0011, 1100, 0000, 1001\}$ and let a_1, a_2, a_3 and a_4 be the encoding variables. Then a_1 can be eliminated since the resulting set of codes are still unique: $\hat{A} = \{011, 100, 000, 001\}$. Since no remaining variables can be eliminated, $\{a_1\}$ is also a maximal removal set. However, $\{a_1\}$ is not a maximum set since $\{a_2, a_3\}$ is also a permissible, but larger, removal set. □

The goal is to remove the largest set of variables. The maximum variable elimination problem is stated as follows:

Problem 6.1 (Maximum Variable Removal Problem) Given a set of binary patterns $A \subseteq B^n$, find the maximum removal set such that the resulting set of codes induced by the remaining encoding variables remain unique. □

To handle redundant variable removal for very large sets of elements (*e.g.*, 10^{50} elements and beyond), the set must be represented symbolically as a characteristic function in BDD form. Depending on the regularity of the elements, BDD's can represent very large sets. Once captured in BDD form, redundant variables can be detected very easily as follows:

Lemma 6.1 *A variable a_i is redundant if and only if*

$$(\forall a_i)A = 0. \tag{6.1}$$

Proof: Suppose a_i is irredundant. This is equivalent to the existence of two elements, A_j and A_k, in A such that they are only distinguishable by a_i, meaning the value is identical for both elements in all other variables . This requires either $a_i = 0$ for A_j and $a_i = 1$ for A_k, or $a_i = 1$ for A_j and $a_i = 0$ for A_k. In either case, $\forall a_i A$ will be non-empty. ■

The new set of codes after removing the redundant variable a_i can be derived as follows:

$$\hat{A} = (\exists a_i)A. \tag{6.2}$$

The problem of finding the maximum removal set is NP-complete so that any algorithm which solves this problem exactly can be expected to have exponential worst-case complexity

in the number of variables even if BDD operations are used. In practice, significantly larger removal sets than obtained by a greedy solution can often be found if more intelligent algorithms are devised.

Example 6.2 Consider the following set of elements:

	a_1	a_2	a_3	a_4	a_5
q_1	0	0	0	1	1
q_2	0	1	0	0	1
q_3	1	1	1	0	1
q_4	1	0	0	0	0

The characteristic function for the above set can be expressed as

$$\bar{a_1}\bar{a_2}\bar{a_3}a_4a_5 + \bar{a_1}a_2\bar{a_3}\bar{a_4}a_5 + a_1a_2a_3\bar{a_4}a_5 + a_1\bar{a_2}\bar{a_3}\bar{a_4}\bar{a_5}$$

In this example, the greedy solution obtained by removing variables in order is $\{a_1, a_2\}$. However, $\{a_3, a_4, a_5\}$ is a larger permissible removable set. □

6.3 BDD-Based Branch-and-Bound Algorithm

A basic branch-and-bound algorithm for finding the maximum removal set involves the following steps:

1. Determine an upper bound on the number of variables that can still be eliminated. If the size of the current selected set plus the upper bound is less than or equal to a bound (*e.g.*, the size of the best solution seen so far), return from this level of recursion. If there are no more free variables, declare the current solution as the best solution recorded so far.

2. Select a free variable to eliminate. The test for a free variable is given by Equation 6.1.

3. Add the variable to the selected set and solve the subproblem by obtaining the new set of patterns using Equation 6.2. Then, solve the subproblem by not selecting this variable.

Constant Variables: If the value of the variable a_i is always 0 or always 1 in every element of A, then a_i is said to be a *constant* variable. A constant variable can be removed unconditionally since it cannot distinguish any pairs of codes in A.

Choice of Redundant Variable: At each level of recursion, there may be many redundant variables. Good heuristics for choosing the branching redundant variable are crucial in speeding up the branch and bound algorithm. One effective heuristic is to choose the most *unate* variable. The most unate variable is determined as follows:

$$\text{weight}(a_i) = \text{absolute}(|A_{a_i}| - |A_{\overline{a_i}}|) \tag{6.3}$$

$$\text{select}(a_i) = \arg\max_{a_i} (\text{weight}(a_i)) \tag{6.4}$$

Here, $|A_{a_i}|$ is the number of elements where a_i is 1, and $|A_{\overline{a_i}}|$ is the number of elements where a_i is 0. The weight(a_i) is the absolute difference. For example, if $|A_{a_i}| = 2$ and $|A_{\overline{a_i}}| = 5$, then weight(a_i) = absolute($2 - 5$) = absolute(-3) = 3. The reason for removing the most unate variables first is because they are the least effective in distinguishing pairs of codes.

Maximal Removable Set: An important feature of the proposed branch and bound algorithm is the use of maximal removable set. This routine is used to compute a simple upper bound that corresponds to the maximum number of removable variables. Here, the number of remaining free variables is used as a heuristic upper bound. A simple bounded look-ahead scheme can be applied for greater accuracy. The purpose of this upper bound is to bound the recursion early so that inferior parts of the search space may be discarded as early as possible.

When the above branch and bound procedure terminates, it is guaranteed to return the **maximum** removal set of variables that can be eliminated.

The search space is pruned significantly using the described bounding technique. All operations are performed using BDD's so that the set of elements is never explicitly enumerated. A fast heuristic algorithm can be derived from the above procedure by terminating the recursion early. Experiments show that the first leaf solution is usually quite good. A number of test cases have been found where greedy method (removal in the order of occurrence) is considerably less effective than the procedures proposed here. The effectiveness of the proposed BDD-based algorithms are dependent on the regularity of the set considered. Using the heuristics described, substantial reduction in the number of encoding variables has been found for many cases.

circuit	states	bits	exact	reduction	minimum	CPU
s208	17	8	5	37.5	5	0.01
s298	218	14	12	14.3	8	3.40
s344	2625	15	15	0.0	12	0.85
s349	2625	15	15	0.0	12	0.84
s382	8865	21	18	14.3	14	29.44
s386	13	6	6	0.0	4	0.01
s400	8865	21	18	14.3	14	29.45
s444	8865	21	17	19.0	14	34.38
s510	47	6	6	0.0	6	0.01
s526	8868	21	19	9.5	14	30.31
s641	1544	19	14	26.3	11	0.83
s713	1544	19	14	26.3	11	0.84
s820	25	5	5	0.0	5	0.01
s832	25	5	5	0.0	5	0.01

Table 6.1: Exact Redundant State Register Removal Results.

states	number of reachable states
bits	number of state-bits in original circuit
exact	number of state-bits after exact removal
reduction	percentage reduction in the number of state variables
minimum	lower bound on the number of state variables
CPU	CPU times in seconds for exact redundant state-bit removal

6.4 Experimental Results

The techniques described in the previous sections have been implemented using Berkeley's implementation of a BDD package originally described in [11]. In this section, results are presented on experiments based on exact redundant state-bit removal. All experimental results presented were measured on a DECstation 3100 workstation. The primary set of test cases used were obtained form the ISCAS-89 sequential test benchmark set.

The results of exact redundant state-bit removal are presented in Table 6.1. The set of reachable states is computed using BDD-based implicit enumeration techniques [26]. In particular Berkeley's implicit enumeration routines [89] and some auxiliary routines from [62] were used. In most cases, the number of reachable states is much less than 2^N, where N is the number of state-bits. In performing exact redundant state-bit removal, the maximal removal set bounding heuristic was found to be very powerful in reducing the depth of

circuit	original relation	reduced relation	reduction
s208	51	35	31.4
s298	461	413	10.4
s344	377	377	0.0
s349	377	377	0.0
s382	4129	3890	5.8
s386	140	140	0.0
s400	4129	3890	5.8
s444	654	557	14.8
s510	287	287	0.0
s526	462	414	10.4
s641	3280	2917	11.1
s713	3280	2917	11.1
s820	217	217	0.0
s832	217	217	0.0

Table 6.2: Comparisons of Transition Relation Sizes.

original relation	BDD size of original transition relation
reduced relation	BDD size of transition relation after state-bit removal
reduction	percentage reduction in the BDD size of the transition relations

recursion. The heuristics for choosing the branching variables were also quite important. Surprisingly, the circuits from the ISCAS-89 benchmark set contain a very large number of redundant state-bits. The percentage reduction is shown in column **reduction**. Using the exact BDD-based algorithms, it was possible to obtain exact results in all cases with modest CPU times.

In the second experiment, the effects of redundant state-bit removal on the size of the BDD representation of the transition relation were examined. Reducing the size of the transition relation can have a dramatic impact on the verification, especially design verification, since properties may be checked on the transition relation repeatedly. Table 6.2 shows the BDD size of the transition relations before and after redundant state-bit removal. For cases where no redundant state-bits were identified, the size of the transition relation remained the same. However, in a significant number of examples tested, the size of the transition relation reduced by a significant amount. For example, the transition relation of

circuit	literals only			with bits		
	area1	area2	ratio	area1	area2	ratio
s208	76	47	0.62	132	82	0.62
s298	112	85	0.76	210	169	0.80
s344	141	141	1.00	246	246	1.00
s349	146	146	1.00	251	251	1.00
s382	152	135	0.89	299	261	0.87
s386	132	132	1.00	174	174	1.00
s400	156	135	0.87	303	261	0.86
s444	150	136	0.91	297	255	0.86
s510	247	247	1.00	289	289	1.00
s526	190	146	0.77	337	279	0.83
s641	189	219	1.16	322	317	0.98
s713	198	219	1.11	331	317	0.96
s820	252	252	1.00	287	287	1.00
s832	258	258	1.00	293	293	1.00

Table 6.3: Comparisons of Gate-Level Implementations.

literals only	literals in factored form using MIS-II and standard script
with bits	literals plus latch costs (7 literals per state-bit)
area1	optimized area cost for original sequential circuit
area2	optimized area cost of sequential circuit after state-bit removal and simplification
ratio	area cost ratio between modified and original circuit

s208 was reduced by over 31%.

In the third experiment, the effects of redundant state-bit removal on the size of the gate level implementation were analyzed. The multi-level logic optimization system MIS-II Version 2.2 [13] is used with the standard script to obtain an optimized result. Under the section labeled **literals only**, the area measured in terms of literal counts in factored form for each test case is reported. **area1** is the area of the optimized original circuit, and **area2** is the area of the optimized modified circuit. The area results reported are in terms of the **total area**, including both the next-state and output functions. The area of the original circuit was obtained by running the standard script. The area of the modified circuit was obtained in three steps: first attach necessary re-encoding circuitry to the original circuit; then perform some partial collapsing; then optimize using the standard script.

Depending on the example, significant reduction in area can be achieved (between

circuit	latch-removal		script.rugged	
	lits.	bits	lits.	bits
s208	47	5	46	8
s298	85	8	97	14
s382	135	18	151	21
s400	135	18	144	21
s444	136	17	140	21
s526	146	19	156	21
s641	219	14	185	17
s713	219	14	186	17

Table 6.4: Comparisons of State-Bit Removal with `script.rugged`.

lits. number of literals
bits number of state-bits

20 – 38% on several examples). Note that much of the reduction comes about because the next-state functions corresponding to removed state variables did not need to be implemented. However, the procedure cannot guarantee that the area will be reduced since the circuit structure itself must be modified. In examples s641 and s713, the area actually increased if the cost of state-bits is not considered. Since some state-bits have been removed, it would be more accurate to compare area with the costs of state-bits accounted. Under the section labeled **with bits**, the area results are reported in terms of literal count by giving each state-bit the cost of 7 literals. This corresponds to the relative size of a standard cell implementation versus a similar 7 literal logic gate. With the state-bit cost considered, the total area actually decreased for the examples s641 and s713.

State-bits may also be removed implicitly by using the unreachable states as external don't cares. If an intense optimization script is applied with these don't cares, some next-state functions can be simplified to a constant **1** or **0**. To determine the effectiveness of this method, the optimization script **script.rugged** was applied to the benchmarks with the unreachable states as external don't cares. `script.rugged` contains the command `full_simplify` that has been shown to be quite effective in exploiting unreachable states don't care. This script in general produces much more area efficient implementations than the standard script of MIS-II. These results are shown in Table 6.4. Only benchmarks where state-bits can be removed by the state-bit removal procedure are included. For the

benchmarks s641 and s832, the number of state-bits did indeed reduce from 19 bits to 17 bits by applying `script,rugged`. The number of literals is also less. However, this is not always effective.

6.5 Conclusions

In this chapter, the problem of removing redundant state-bits that are not needed to differentiate the state-codes was addressed. A new BDD-based algorithm was then presented for this problem that is guaranteed to return the maximum set of removable state-bits. It makes use of new branching heuristics and a new bounding technique called maximal removable set. The size of sequential circuits that can be handled using the techniques described here is limited by the size of the BDD representation of the reachable states, which can be very large dependingly on the regularity of the state-space. For the examples tested, the exact solutions were found in all cases with modest CPU times. It has been shown how the technique of redundant state-bit removal can be used to simplify both the BDD representation of the transition relation for verification as well as the size of the gate-level implementation for synthesis. For the examples tested, significant reduction in both the BDD size as well as implementation area were achieved.

Chapter 7

Conclusions

The goal of this dissertation has been to develop new synthesis algorithms for automating the design process from a symbolic specification of hardware to an optimized VLSI design. Specifically, the synthesis of sequential designs was considered. The starting point of synthesis can be a textual description written in a hardware description language (*e.g.*, VHDL [50] and ELLA [71]) and using symbolic data abstraction. Towards this end, techniques for symbolic encoding and optimization of sequential circuits were explored in this dissertation.

The main contributions from this research are as follows. The first part of this dissertation was concerned with the problem of encoding symbolic specifications. In particular, encoding techniques targeting both two-level and multi-level logic implementations were developed. The second part of this dissertation was concerned with the problem of optimizing sequential circuits encoded from symbolic hardware descriptions. The goal was to develop efficient techniques that can be used to optimize large sequential circuits that may contain both control and data-path circuitry.

In Chapter 2, the concept of symbolic relation was introduced. Symbolic relation is a generalization of traditional symbolic specification with the added freedom of having multiple output choices. The minimization problem is more difficult since the selection of encodings as well as the output mappings must both be optimally determined. A unified framework for finding exact solutions for two-level minimization has been developed. The approach is based on a novel binate covering formulation that combines the processes of prime selection and constrained encoding into a single optimization step. It was shown that the binate covering problem can be solved using binary decision diagrams in linear time if

the corresponding BDD for the covering constraints can be built. The same technique can be used to solve the minimum cost satisfiability problem in the same way. The symbolic relation concept has been further generalized to solve the problems of state minimization and state assignment problems simultaneously. An exact formulation of the problem has been developed for finding optimum merging and encoding of states such that the resulting two-level implementation has minimum area.

Although a theoretical framework for exact minimization of symbolic relations has been developed, the algorithms in the framework are not practical for real-life examples. However experience has shown that effective heuristic algorithms can be developed based on an incremental improvement strategy. The basic heuristic minimization strategy of ESPRESSO [12] seems like the most likely candidate. Recent work in this direction has already seen some progress [94]. Also, the state-of-the-art in binate covering must also be advanced significantly before practical solutions can be achieved for large problem instances.

In Chapter 3, new algorithms for multi-level encoding were presented. An important goal of this research was to develop fast, but effective, algorithms for this problem so that they can be used to encode large symbolic descriptions compiled from hardware description languages. A general symbolic encoding program called JEDI has been developed for this purpose based on an estimation approach. The main optimization problem to be solved is the minimum cost graph embedding problem. A spectrum of new algorithms, both heuristic and exact procedures, have been developed for solving this problem. The most significant developments here are the generalization of the encoding problem to general symbolic specifications, improved estimation models, and new graph embedding strategies.

While the overall results are very good, more insights are required as to why estimation-based techniques, like the ones described here, work as effectively as they do in practice. Some theoretical and intuitive justifications have been given in [33, 39], but much mystery still remains. An important open problem that must be addressed is the issue of code length. All current multi-level encoding tools require the user to specify a desired code length. It is fair to say that there is no known technique for exploring different code lengths optimally, at least not for the multi-level case. In practice, real-life complex controller designs by humans often make use of many more latches than minimum code length. However, when these designs are re-extracted to the state transition graph level and re-encoded with the state-of-the-art symbolic encoding tools, the re-encoded results are often two or three times larger than the original implementations. This is partly because

current symbolic encoding tools do not explore different code lengths effectively and do not work well with long code lengths even when given. Effective techniques for solving this problem can potentially give substantial gains over current symbolic encoding approaches.

One of the main sources of optimization is the use of sequential don't-care conditions in logic optimization. These don't-care conditions can be derived by examining the underlying global state-space of the corresponding sequential circuit. It has been shown that these sequential don't-care conditions are very important for reducing area and improving testability. The main issue is the efficient derivation of these sequential don't-care sets starting from a multi-level sequential logic representation. In Chapter 4, new algorithms for deriving these sequential don't-care sets efficiently were described. These algorithms are based on the use of binary decision diagrams and implicit analysis techniques. Using these techniques, sequential don't-care sets for very large sequential circuits have been computed exactly. In particular, the complete set of invalid and equivalent states can be computed directly from a gate-level description.

Additional work needs to be done in two areas. First, much more efficient state-space traversal algorithms must be developed to increase practical applicability of the developed techniques. Many practical test cases still cannot be traversed by state-of-the-art implicit enumeration techniques. For example, several of the large ISCAS sequential benchmark examples currently cannot be handled by any known techniques. Some of these examples contain over sixteen thousand state latches. The main limitation appears to be the memory requirement for storing the intermediate computations. The second is to apply the invalid states and equivalent states computation algorithms to other problem areas other than sequential optimization. For example, equivalent state information may be used to improve the performance of sequential test pattern generation algorithms or to extend the practicality of synthesis-for-testability methods.

The last two chapters of this dissertation were concerned with basic symbolic computation techniques based on binary decision diagrams for manipulating sequential circuits. New methods for representing and manipulating equivalence classes efficiently were proposed in Chapter 5. These techniques make it possible to develop an exact implicit state minimization algorithm for large sequential circuits. In Chapter 6, another symbolic technique was proposed. The problem addressed was the removal of redundant encoding variables that are not necessary to distinguish pairs of codes. This can be viewed as a restricted form of re-encoding. An exact algorithm based on binary decision diagrams has

been developed. One application of this technique is in the optimization of sequential circuits. Although some applications have been described using the basic core computation techniques described in Chapter 5 and Chapter 6, more work should to be done to determine the capabilities and applicability of these concepts in other problem areas.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[2] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, pages 509–516, June 1978.

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

[4] P. Ashar, S. Devadas, and A. R. Newton. Irredundant interacting sequential machines via optimal logic synthesis. *IEEE Transactions on Computer-Aided Design*, 10:311–325, March 1991.

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

[6] M.R. Barbacci. Instruction set processor specifications (ISPS): The notation and its applications. *IEEE Transactions on Computers*, C-30(1):24–40, January 1981.

[7] K. Bartlett, R. Brayton, G. Hachtel, R. Jacoby, C. Morrison, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level logic minimization using implicit don't cares. *IEEE Transactions on Computer-Aided Design*, CAD-7(6):723–740, June 1988.

[8] M. Beardslee. Private communications. 1990.

[9] C. Berthet, O. Coudert, and J.C. Madre. New ideas on symbolic manipulations of finite state machines. In *Proceedings of the IEEE International Conference on Computer Design*, October 1990.

[10] 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*, pages 62–65, 1987.

[11] K.L. Brace, R.L. Rudell, and R.E. Bryant. Efficient implementation of a BDD package. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 9–16, June 1990.

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

[13] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1062–1081, November 1987.

[14] R. K. Brayton and F. Somenzi. An exact minimizer for Boolean relations. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 316–319, November 1989.

[15] R.K. Brayton, G.D. Hachtel, and A. Sangiovanni-Vincentelli. Multi-level logic synthesis. *Proceedings of the IEEE*, 72(2):264–300, February 1990.

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

[17] R.K. Brayton and F. Somenzi. Minimization of Boolean relations. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 739–743, May 1989.

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

[19] F.M. Brown. *Boolean Reasoning: The Logic of Boolean Equations.* Kluwer Publisher, 1990.

[20] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[21] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Logic in Computer Science*, June 1990.

[22] Raul Camposano and Wayne Wolf. *High-Level Synthesis.* Kluwer Publisher, 1991.

[23] E. Cerny and M. A. Marin. An approach to unified methodology of combinational switching circuits. *IEEE Transactions on Computers*, C-26(8):745–756, August 1977.

[24] H. Cho, G.D. Hachtel, S.W. Jeong, B. Plessier, E. Schwarz, and F. Somenzi. ATPG aspects of FSM verification. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 134–137, November 1990.

[25] O. Coudert, C. Berthet, and J.C. Madre. Verification of sequential machines using Boolean function vectors. In L.J.M. Claesen, editor, *Formal VLSI Correctness Verification*, pages 111–128. Elsevier Science Publishers B.V., North Holland Press, 1990.

[26] O. Coudert, C. Berthet, and J.C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems.* Springer-Verlag, June 1990.

[27] O. Coudert and J.C. Madre. A unified framework for the formal verification of sequential circuits. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 126–129, November 1990.

[28] M. Damiani and G. De Micheli. Synchronous logic synthesis: Circuit specifications and optimization algorithms. In *Proceedings of the Synthesis and Simulation Meeting and International Interchange*, May 1990.

[29] J. Darringer, D. Brand, J. Gerbi, W. Joyner, and L.Trevillyan. LSS: A system for production logic synthesis. *IBM Journal of Research and Development*, 28(5):326–328, September 1984.

[30] H. de Man, F. Catthoor, G. Goossens, J. Vanhoof, J.L. Van Meerbergen, S. Note, and J.A. Huisken. Architecture-driven synthesis techniques for VLSI implementation of DSP algorithms. *Proceedings of the IEEE*, 72(2):319–335, February 1990.

[31] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Technology mapping in MIS. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 116–119, November 1987.

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

[33] S. Devadas, H.-K. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. MUSTANG: State assignment of finite state machines targeting multi-level logic implementations. *IEEE Transactions on Computer-Aided Design*, CAD-7(12):1290–1300, December 1988.

[34] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. Irredundant sequential machines via optimal logic synthesis. *IEEE Transactions on Computer-Aided Design*, CAD-9(1):8–18, January 1990.

[35] S. Devadas and A. R. Newton. Exact algorithms for output encoding, state assignment and four-level Boolean minimization. *IEEE Transactions on Computer-Aided Design*, January 1991.

[36] S. Devadas and A.R. Newton. Decomposition and factorization of sequential finite state machines. *IEEE Transactions on Computer-Aided Design*, 8:1206–1217, November 1989.

[37] Srinivas Devadas and A. Richard Newton. Topological optimization of multiple-level array logic. *IEEE Transactions on Computer-Aided Design*, pages 915–942, November 1987.

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

[39] X. Du, G. Hachtel, B. Lin, and A.R. Newton. MUSE: A multi-level state encoding algorithm for state assignment. *IEEE Transactions on Computer-Aided Design*, 10:28–38, January 1991.

[40] M. Fujita, Y. Tamiya, Y. Matsunaga, and K.C. Chen. Multi-level logic synthesis for boolean relations. In *Submitted to VLSI'91*, August 1991.

[41] A. Ghosh, S. Devadas, and A.R. Newton. Heuristic minimization of Boolean relations using testing techniques. In *Proceedings of the IEEE International Conference on Computer Design*, pages 277–281, October 1990.

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

[43] R.L. Graham and H.O. Pollak. On embedding graphs in squashed cubes. In *Graph Theory and Applications 303*. Springer Verlag, 1972.

[44] A. Grasselli and F. Luccio. A method for minimizing the number of internal states in incompletely specified sequential networks. *IEEE Trans. Elec. Comp.*, EC-14:350–359, June 1965.

[45] D. Gregory, K. Bartlett, A. DeGeus, and G. Hachtel. SOCRATES: A system for automatically synthesizing and optimizing combinational logic. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 79–85, June 1986.

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

[47] J. Hartmanis and R.E. Stearns. Some dangers in the state reduction of sequential machines. In *Information and Control*, pages 252–260, September 1962.

[48] R. W. House and D. W. Stevens. A new rule for reducing CC tables. *IEEE Transactions on Computers*, C-19:1108–1111, November 1970.

[49] T. Hwang, R.M. Owens, and M.J. Irwin. Exploiting communication complexity for multi-level logic synthesis. *IEEE Transactions on Computer-Aided Design*, 9(10):1017–1027, October 1990.

[50] IEEE Inc., 345 East 47th Street, New York, NY, 10017. IEEE Standard 1076-1987. *IEEE Standard VHDL Language Reference Manual*, March 1982.

[51] R.M. Karp. Function decomposition and switching circuit design. *Journal of Society of Industrial Applied Mathematics*, 11(2), June 1963.

[52] K. Keutzer. DAGON: Technology binding and local optimization. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 341–347, June 1987.

[53] S. Kirkpatrick, C. Gelatt, Jr., and M. vecchi. Optimization by simulated annealing. *Science*, 220:671–680, May 1983.

[54] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw Hill, 1978.

[55] E.S. Kuh and T. Ohtsuki. Recent advances in vlsi layout. *Proceedings of the IEEE*, 72(2):237–263, February 1990.

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

[57] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing synchronous circuitry by retiming. In R. E. Bryant, editor, *Advanced Research in VLSI: Proceedings of the Third Caltech Conference*, pages 86–116. Computer Science Press, 1983.

[58] B. Lin. Synthesis of finite state machines by combined machine restructuring and state assignment. In *Ph.D Qualifying Examination Proposal*, April 1989.

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

[60] B. Lin and A.R. Newton. Restructuring state machines and state assignment: Relationship to minimizing logic across latch boundaries. In *Proceedings of the MCNC International Workshop on Logic Synthesis*, May 1989.

[61] B. Lin and F. Somenzi. Minimization of symbolic relations. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 88–91, November 1990.

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

[63] B. Lin, G. S. Whitcomb, and A. R. Newton. Symbolic don't cares and equivalence in high-level synthesis. In *IFIP International Working Conference on Logic and Architecture Synthesis*, May 1990.

[64] R. Lisanke. Logic synthesis and optimization benchmarks user guide bersion 2.0. In *Technical Report, MCNC, P.O Box 12889, Research Triangle Park, North Carolina 27709*, December 1988.

[65] S. Malik. *Combinational Logic Optimization Techniuques in Sequential Logic Synthesis.* PhD thesis, University of California, Berkeley, November 1990.

[66] S. Malik, E.M Sentovich, R.K. Brayton, and A. Sangiovanni-Vincentelli. Retiming and resynthesis: Optimizing sequential circuits using combinational techniques. *IEEE Transactions on Computer-Aided Design*, 10(1):74–84, January 1991.

[67] E. J. McCluskey. Minimization of Boolean functions. *Bell Laboratory System Technical Journal*, 35:1417–1444, April 1956.

[68] 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.

[69] G. De Micheli. Synchronous logic synthesis: Algorithms for cycle-time minimization. *IEEE Transactions on Computer-Aided Design*, 10(1):63–73, January 1991.

[70] 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):269–285, July 1985.

[71] J.D. Morrison. ELLA: Hardware description or specification. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, November 1984.

[72] Berkeley's BDD Package. MIS-II logic synthesis system. 1990.

[73] C. Pixley. A computational theory and implementation of sequential hardware equivalence. In *Proc. of CAV Workshop, Rutgers University*, June 1990.

[74] W. Quine. The problem of simplifying truth functions. *American Math. Monthly*, 59:521–531, 1952.

[75] J. Rho, G. Hachtel, F. Somenzi, and R. Jacoby. Exact and heuristic minimization of incompletely specified finite state machines. In *Proceedings of the IEEE European Design Automation Conference*, February 1990.

[76] Fabio Romeo and Alberto Sangiovanni-Vincentelli. Probabilistic hill-climbing algorithms: Properties and applications. In *Chapel Hill Conference on Very Large Scale Integration*, 1985.

[77] R. Rudell. *Logic synthesis for VLSI Design.* PhD thesis, University of California, Berkeley, April 1989.

[78] R. L. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):727–750, September 1987.

[79] T. Sasao. Multiple-valued decomposition of generalized boolean functions and the complexity of programmable logic arrays. *IEEE Transactions on Computers*, C-30:635–643, September 1981.

[80] H. Savoj and R. K. Brayton. Observability functions and observability don't cares. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, November 1991.

[81] H. Savoj, H. J. Touati, and R. K. Brayton. Extracting local don't cares for network optimization. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, November 1991.

[82] Carl Sechen and Alberto Sangiovanni-Vincentelli. The TimberWolf placement and routing package. In *Proceedings of the Custom Integrated Circuit Conference*, May 1984.

[83] Russell B. Segal. BDSYN: Logic description translator; BDSIM: Switch-level simulator. In *Research Report, Electronics Research Laboratory, University of California, Berkeley*, May 1987.

[84] M. C. McFarland S.J., A. C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 72(2):301–318, February 1990.

[85] G. Sorkin. Combinatorial optimization, simulated annealing, and fractals. *IBM Journal of Research and Development*, April 1988.

[86] A. Stolzle. A VLSI wordprocessing subsystem for a real time large vocabulary continuois speech recognition system. In *MS Thesis*, September 1989.

[87] H. J. Touati. *Performance Oriented Technology Mapping.* PhD thesis, University of California, Berkeley, November 1990.

[88] H. J. Touati, R. K. Brayton, and R. Kurshan. Testing language containment for ω-Automata using BDD's. In *Formal Methods in VLSI, Miami*, January 1991.

[89] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 130–133, November 1990.

[90] S. H. Unger. *Asynchronous Sequential Switching Circuits.* Wiley Interscience, 1969.

[91] National Bureau of Standards U.S. Department of Commerce. Data encryption standard. In *Federal Information Processing Standards Publication (FIPS PUB 46)*, January 1977.

[92] T. Villa and A. Sangiovanni-Vincentelli. NOVA: State assignment of finite state machines for optimal two-level logic implementations. In *Proceedings of the ACM/IEEE Design Automation Conference*, pages 327–332, June 1989.

[93] Y. Watanabe and R.K. Brayton. Heuristic minimization of Boolean relations. In *Proceedings of the MCNC International Workshop on Logic Synthesis*, May 1991.

[94] Y. Watanabe and R.K. Brayton. Minimization of multiple-valued relations. In *Research Report, Electronics Research Laboratory, University of California, Berkeley*, May 1991.

[95] S. Yang and M. 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.